A JML-Based Strategy for Incorporating Formal Specifications into the Software Development Process

Author:

João Miguel Alves Pestana, 2046403

Advisor:

Néstor Cataño

Abstract

This thesis presents a JML-based strategy that incorporates formal specifications into the software development process of object-oriented programs. The strategy evolves functional requirements into a “semi-formal” requirements form, and then expressing them as JML formal specifications. The strategy is implemented as a formal-specification pseudo-phase that runs in parallel with the other phase of software development. What makes our strategy different from other software development strategies used in literature is the particular use of JML specifications we make all along the way from requirements to validation-and-verification.

Keywords

Software Development

Software Correctness

Formal Methods

Design by Contract

Java Modelling Language (JML)

JML-based Strategy

Informal Functional Requirements

Semi-Formal Functional Requirements

Invariants

Formal Specifications

Abstract Variables

Java

Acknowledgments

First of all, I would like to thank Professor Néstor Cataño, my thesis advisor, without whose guidance I would surely have been lost. I'm grateful for his patience, support and specially his enthusiasm on this work. I'm also grateful for the documents he provided to me and his knowledge, as well as his advice on formal methods and the Java Modelling Language.

I would like also to thank Ricardo Rodrigues, my master's mate, for his collaboration on this thesis and being by my side while we worked together. His work helped to validate the work proposed in this thesis, and in some aspects had a strong influence in its precise definition. Our two thesis works complement each other. I'm grateful for his support in those hard times we had while working.

Finally, I give many thanks to my family and friends who gave me support and strength by being on my side each day. Words are not enough to express all my gratitude.

Contents

1. Introduction 1
2. Preliminaries 2
2.1. Software Correctness 2
2.1.1. Design by Contract 3
2.2. Formal Methods in the Software Development Process 10
3. The Java Modelling Language (JML) 13
3.1. The JML Specifications 13
3.1.1. JML Expressions 14
3.2. Abstract Variables 16
3.2.1. JML Abstract Data Types 17
3.3. The JML Common Tools 18
4. Related Work 18
4.1. Formal Methods Strategies for Software Development Processes 18
4.2. SOFL 19
4.3. ConGu 21
5. The JML-based Strategy for Software Development of Java Programs 22
5.1. Requirements Analysis 24
5.2. From Informal Functional Requirements to Semi-Formal Specifications 25
5.2.1. Semi-Formal Functional Requirements 26
5.2.2. Class Invariants 27
5.2.3. System Invariants 27
5.3. Design 28
5.4. Implementation 29
5.4.1. Writing JML Abstract Variables 30
5.4.2. Writing JML Class Invariants 31
5.4.3. Writing JML Method Functional Specifications 32
5.4.4. Coding the applications 33
5.5. Validation and Verification 33
6. A Running Example 34
6.1. Introduction of an Application to be Formally Developed ..... **Erro! Marcador não definido.**

6.1.1. The HealthCard Application .... **Erro! Marcador não definido.**
6.1.2. HealthCard Formal Development 35
6.1.3. HealthCard System Architecture 36
6.2. Smart cards and Java Card 38
6.2.1. Elements of a Java Card Application 38
6.2.2. Accessing the Smart Card (Communication in Java Card) 40
6.2.3. Java Card Remote Method Invocation (JCRMI) 41
6.3. JML-based Formal Development of the HealthCard 43
6.3.1. Getting the Informal Requirements 43
6.3.2. Getting the Semi-Formal Functional Requirements 44
6.3.3. Getting the Class and System Invariants 45
6.3.4. Design and Implementation 45
6.3.5. JML Formal Specification Pseudo-Phase 46
7. Conclusion 47
Bibliography 51
List of Figures 54
List of Code 54
List of Tables 54

1. Introduction

Although software engineering methods provide a disciplined approach to software development, it is still quite common to find flawed software systems. An approach to tackle the problem of constructing correct programs is through the use of mathematical formalisms and mathematically based tools as part of software engineering practices. Formal specifications allow the capture of requirements unambiguously as part of a software engineering methodology. A formal specification can be used to generate a collection of documents describing the expected behaviour of a system. This documentation can be used to resolve any differences regarding the expected behaviour of the system between members of the quality assurance team, the programmers and the client. An interesting effort towards the development and use of tools based on a common specification language is JML (short for Java Modelling Language), which is the standard language for formally specifying the behaviour of Java classes. The JML is a tool that provides support to B. Meyer's design by contract principles. (Leavens G. , 2008) (Leavens & Cheon, 2006) (Meyer, 1997) JML makes possible to use run-time and static checkers for checking program correctness.

In this thesis we propose a software development strategy in the style of Bertrand Meyer's design-by-contract principles, which makes use of JML specifications for writing contracts (see Section 3). JML (Leavens) is used as the formal specification language for writing specifications to support the correct implementation of Java programs. Our strategy consists in evolving informal functional software requirements (written in English) into formal specifications, through an intermediate stage in which semi-formal requirements are written, i.e., requirements written in English but in a more mathematical style. The informal functional requirements are suggested by the client (or stakeholders). Often these requirements are ambiguous, inconsistent and incomplete, due to the use of natural language. Our strategy involves the transformation of informal functional requirements into semi-formal requirements so as to provide functional requirements a structure. Having semi-formal functional requirements is halfway to obtain formal specifications. Hence, the client is not only capable of following and supporting the software development process, but also the obtained formal specifications can be used by the software development team to solve differences. For the transformation process between informal functional requirements and formal specifications, we propose to write the informal functional requirements as semi-formal functional requirements of the form *if <event/condition> then <restriction/rule>*, or as semiformal class and system invariants respectively specifications imposing system/class properties restrictions in small and large scale. Later, these semi-formal requirements are to be ported into JML method specifications, and JML class invariants respectively. These JML specifications are written within the Java implementation of the application. Some of the benefits of using JML specifications in software development process are inherent documentation aspect of JML specifications, and the existence of tools that make the specifications executable or allow making runtime assertion checks of the JML specifications against their respective implementation code. With JML as a tool for code documentation, one can apply the principles of design by contract while programming the client applications/classes that will request services on the supplier side (JML specified classes and methods). That is, by having the supplier applications specified with JML, one has to respect the conditions when coding a call for a supplier method. This code programming style helps the reduction of redundant code, like unnecessary data validations. By using JML specifications on the supplier side, one specifies the conditions of a method and invariants, so there is no need for validating code of the passing arguments or another conditions validation because it

becomes the responsibility of the client applications to make validations. Therefore, we can have a lightweight supplier application/class.

The structure of this document is as it follows. First, in Section 2, we describe the preliminaries of this work, i.e., we describe the background of software correctness, design by contract and the usage of formal methods in software development processes. In Section 3, we give an overview of JML, presenting some specification expressions and the notion of abstract variables. Further, in Section 4 we present some related work, including types of strategies for incorporating formal specifications into the software development process. In Section 5, we present our proposed strategy of incorporating formal methods into software development processes, including a description for each phase of the development process. Section 6 shows an example of development of a Smart Card application using our strategy. The application was originally written by Ricardo Rodrigues as part of his Master thesis work (Rodrigues, 2009)

2. Preliminaries

In this section we present topics on the software correctness and formal methods in software development processes. The topics described here provide the minimal background that is needed to understand the matter of the thesis. The introduction of a strategy for incorporating formal specifications into the software development process is the key aspect of in this thesis. Section 2.1 presents software correctness, including the Design by Contract design methodology, which is the basis of the strategy, and next we focus on the use of formal methods in the development process.

2.1. Software Correctness

To determine if a software program is correct, first we must specify what the software is intended to do. We can't check correctness of a software program in isolation, but only with respect to some specification. Even an incorrect program can perform some processing correctly, although it could be a different processing to the one the developers (or clients) have in mind. Obtaining the requirement specifications is vital as first step in the process of developing a correct software system. (Priestley) To help us assess the correctness of a software program we can express these requirement specifications through the use of assertions. To prove the correctness of a software program's routine body or instruction, these assertions must be checked against it. This proof can be explained here by a correctness formula (also called *Hoare triples*) as an expression of the form denoting the following property (Meyer, 1997) Notice that this formula is a mathematical notation, not a programming construct. It serves only to explain how we can prove the software correctness of a program's routine:

$$\{P\}\ A\ \{Q\}$$

- Where, ***A*** is some operation (for example, an instruction or a routine body); and
- ***P*** is an assertion called precondition; and
- ***Q*** is an assertion called postcondition.

The formula shown above denotes that ***A*** as an operation which requires ***P*** to assure ***Q***, where this must hold to ***A*** be correct. The general meaning of a total correctness formula is: -

"*Any execution of* ***A****, starting in a state where* ***P*** *holds, will terminate in a state where* ***Q*** *holds*." (Meyer, 1997)

As an example, let's use a mathematical expression. Considering ***x*** as an integer value, the arithmetic operation ***x := x + 2***, the precondition ***{x >= 5}*** and the postcondition ***{x >= 6}***, we have the correctness expression:

{x >= 5} x := x + 2 {x >= 6}

Assuming a correct implementation of the integer arithmetic operation, the above expression holds: – if ***x >= 5*** is true when calling the instruction ***x := x + 2***, then ***x>= 6*** will be true afterwards. And of course, if the precondition were false, then the integer arithmetic operation couldn't assure nothing, i.e., the postcondition would be neither true nor false. However, now assuming an incorrect implementation of the above correctly specified expression (i.e. assuming that the instruction violates its specification), if the precondition is true and the postcondition is false, then we could conclude that the integer arithmetic operation was wrongly implemented according to what is specified, i.e., the tester would know that something was wrong with the implementation against the specifications. These preconditions and postconditions can be strengthen or weaken.

- **Stronger preconditions are better**: If we have a strong precondition, that means that the routine must handle a limited set of cases, making easier the routine's job. However, a weaker precondition makes the routine's job harder, as it has to consider several cases not specified by the precondition. A *false* precondition is the strongest possible assertion, since it's never satisfied by any state. By this, any request to execute the routine will be incorrect, as the fault is of the client (i.e., obviously he will never satisfy the preconditions). Whatever the routine's result, it may be useless, but it will be always correct, as it is consistent with the specifications. (Meyer, 1997) However, the least restrictive precondition is the weakest precondition.

- **Weaker postconditions are better**: In postconditions, the situation is reversed. A strong postcondition means that a harder job by the routine must be made to assure all the postconditions. By this, the routine's result has to respect a bigger set of conditions. However, the weaker a postcondition is the better for the routine's job, which means that its result will be satisfied by more states. Asserting a postcondition as *true* is the weakest possible assertion, because it is satisfied by all states. (Meyer, 1997)

The design by contract is a software correctness methodology that has its roots in Hoare logic. Like the Hoare triples formula: *{P} A {Q}* , the design by contract has the concept of preconditions *{P}* and postconditions *{Q}* to document the change in state caused by a piece of a program *A*. These pre- and postconditions are used to strengthen the conditions of a contract between a caller and a supplied routine. (Meyer, 1997) Further, in Section 3, it is presented the Java Modelling Language (JML) which is a design by contract tool. In JML, the *Hoare* logic is applied through the use of *requires* (precondition - *{P}*) and *ensures* (postcondition - *{Q}*) expressions that specifies some Java method's body behaviour (*A*).

2.1.1. Design by Contract

The essence of the *design by contract* methodology is that a *contract* exists between a routine class (*supplier* of certain services) and its callers (*clients* of those services). Some

documents refer the routine classes (serving some services to others) as suppliers, server or server side, while callers can be referred as clients or client side. The design by contract makes the Hoare logic (see beginning of the Section 2.1), a vital component in a program development strengthening the notion of contract. Like in the *Hoare* logic, in design by contract we may specify the routine task's contract with two associated assertions: - *precondition* and a *postcondition*. The precondition defines the properties that must hold whenever the routine is called and the postcondition defines the expected return properties. These two assertions are a way to define a *contract* between the routine and its callers. (Meyer, 1997) The design by contract, as a tool for a software development process can lead to the construction of more reliable object-oriented systems, provides a mechanism through assertions for checking the conformance of the code against its specification. (Meyer, 1997)

Before discussing further the design by contract we'll show below an example of pseudo-specification of a contract. Let's supposed we have a Medicines class for managing a list of medicines. In the following Code 1 we are presented with a Medicines operation specified with pseudo-specification that demonstrates how assertions are used in practice for describing a contract for a routine (Eiffel Software). Here, preconditions and postconditions are represented respectively by **require** and **ensure** keywords. (Meyer, 1997) In JML, these two keywords are actually *requires* and *ensures*, with "s".

```
class MEDICINES create
   make
feature
   quantity: INTEGER
   name_length: INTEGER is 20
...
   addMedicine (medicine: STRING) is
         -- Adds a medicine into the list of medicines.
      require
         medicine.length <= name_length
      do
               insert(medicine)
      ensure
         quantity = old(quantity) + 1
end
...
end -- class MEDICINES
```

Code 1. Example of a Medicines class specified with pseudo-specification

In the example above, the precondition states that a client who calls the *addMedicine* routine must assure that the medicine's name length must be lesser or equal to the constant value of *name_length* which is 20. The postcondition states that the post-state of the method must verify that the quantity is updated and higher by 1 than the old medicine quantity. Note that when we say "client", it refers to a routine that calls another, that is, the contract between a client and a supplier is made by a communication of software-software. (Meyer, 1997) In a contract, both clients and suppliers have obligations and benefits.

2.1.1.1. *Obligations and Benefits*

The precondition is related to the client in a way that it defines the conditions under which is legitimate for the client to call a method, i.e., it's an *obligation* for the client and a *benefit* for that supplier (server). The postcondition is related to the class, which defines the conditions that must be ensured by the class routine on return, i.e., it's a *benefit* for the client and an *obligation* for the supplier. That is, from the previous statements we can say that the benefits are, for the client, the guarantee that he will get what he expects after the call, and for the supplier, the guarantee that certain assumptions will be satisfied when the routine is called, while the obligations are, for the client, to satisfy the requirements as defined by the precondition, and for the supplier, to produce results as defined in the postcondition. (Meyer, 1997) The following example taken from (Tucker & Noonan, 2001) shows how *design by contract* plays out for a factorial computation in respect for client/suppliers' benefits and obligations.

Table 1. A design by contract example (Tucker & Noonan, 2001)

	Obligations	**Benefits**
Client	(*Satisfy precondition* :) Pass $n \geq 0$	(*From postcondition* :) Receive $n!$ computed
Supplier	(*Satisfy postcondition* :) Compute $n!$	(*From precondition* :) Can assume that $n \geq 0$

When an assertion fails, we can assign blame to the party that did not fulfil its responsibilities: if the precondition is violated then the supplier won't be benefited and the client is to blame, and if the postcondition is violated then the client won't be benefited and the routine implementation is to blame. (Meyer, 1997) In any of these cases, part of the contract won't be fulfilled.

Following these obligations and benefits' convention a developer can simplify its programming style while developing an application. Having specified preconditions that clients must respect when calling a routine, the developers may assume when writing the routine's body that the preconditions are satisfied, i.e., the developer do not need to validate them in the routine's body. It helps to clear redundancy in the code as under no circumstances shall the body of a routine ever test for the routine's precondition. This is called the principle of non-redundancy. (Meyer, 1997) By this principle, we add the responsibility of validating the preconditions to the client, reducing the code on the supplier side (server side). For instances, in the previous Table 1, the routine computing the factorial has a precondition that specifies *n* as a positive value or equal to zero, so in its body we haven't to validate if *n* is respecting that condition.

2.1.1.2. *Clearing redundancy*

By following the non-redundancy principle we are clearing out the redundancy in our code. One of the main advantages of clearing redundancy is that it reduces considerably the quantity of lines of code when programming, and thus its complexity. Having been specified as preconditions, the constraints that must be respected for calling a routine, we may assume that those constraints are satisfied when writing the routine body, and also we do not need to test them in the body. (Meyer, 1997) So if a factorial computation meant to produce a positive integer as result, is of the form seen in Code 2:

```
fact(n: INTEGER): INTEGER is
                -- Factorial of n
    require
                n >= 0
    do ... end
```

Code 2. Pre-condition example for a factorial computation (Tucker & Noonan, 2001) (1997)

We may write the "*do … end*" algorithm for computing the factorial without concerning whether n is negative or not. This concern is taken care by the precondition which becomes the clients' responsibility. (Meyer, 1997) If the "*do*" clause was on the form as seen in Code 3:

```
if n < 0 then
    "Handle this erroneous case!"
else
    "Proceed with normal factorial computation"
end
```

Code 3. A redundant test ()

Then the test "$n < 0$ " is not just unnecessary but unacceptable, because it violates the non-redundancy principle. This is a characteristic of the *defensive programming* in which it states that to obtain reliable software one should design every component of a system to protect itself as much as possible. The defensive programming technique is advocated by many software engineering books, but this technique causes redundancy in the code when following the design by contract methodology. The more redundant checks added to a software application, more complexity to the software will be added. This may cause problems to obtain reliability[1] and may imply a performance penalty. (Meyer, 1997) By applying the principle of non-redundancy, we are light weighting the supplier operations. In case of an application developed in Java Card, to be supported by smart cards, this may be a benefit for memory saving on the card side, due its limited small capacity (see Section 6.2 for a description on smart cards and Java Card). When an external client makes a remote call on the card, it is assumed that the preconditions of remote methods in the card side are valid. These preconditions validations are made in the client side, so there is no need of validations in the card side.

The notion of a contract in design by contract can be extended down to the method/procedure level besides the concepts of preconditions and postconditions. A contract can also be strengthened by concepts like invariants, inheritance and exceptions.

2.1.1.3. Invariants

Besides having preconditions and postconditions, we can have invariants to express global properties of routine's contracts between suppliers and clients. Preconditions and postconditions only describe properties of single routines. There is a necessity of expressing global properties of instances of a class, which must be preserved by all routines. We may

[1] Reliability is the ability of a system or component to perform its required functions under stated conditions for a specified period of time.

consider an invariant as being an extension for both preconditions and postconditions of every class's routines. (Meyer, 1997) For instance, let ***A*** be a certain body of a routine (the set of instructions in its **do** clause), ***P*** is precondition, ***Q*** its postcondition and ***INV*** the routine's class invariant. The correctness requirement on ***A*** may be expressed by using the notation introduced earlier in this section as:

$$\{INV \text{ and } P\}\ A\ \{INV \text{ and } Q\}$$

The expression above means that: – "any execution of ***A***, started in any state in which ***INV*** and ***P*** both hold, will terminate in a state in which both ***INV*** and ***Q*** hold". (Meyer, 1997) Here adding the invariant makes both the precondition and the postcondition stronger or equal, i.e., the invariant could either reinforce the conditions or could have no effect on them (redundant conditions). So when implementing the routine's body ***A***, the invariant ***INV*** makes the job easier in addition to the precondition ***P*** due to the assumption that the initial state satisfies ***INV***, further restricting the set of cases that must be handled by the precondition specification. However, in addition to the postcondition ***Q*** which ***A*** must ensure, the routine's body must also ensure that the final state satisfies ***INV***, making the implementation harder. Considering again the earlier *Medicines* class example and its pseudo-specifications shown in the beginning of Section 2.1.1, we demonstrate in Code 4 how we could specify a class invariant. (Eiffel Software)

```
class MEDICINES create
   make
feature
   quantity: INTEGER
   name_length: INTEGER is 20
   total_medicines: INTEGER is 250
...
   addMedicine (medicine: STRING) is
         -- Adds a medicine into the list of medicines.
      requires
         medicine.length <= name_length
      do
            insert(medicine)
      ensures
         quantity = old(quantity) + 1
      end
...
invariant
      quantity <=  total_medicines
end -- class MEDICINES
```

Code 4. Example of a Medicines class implementation with an invariant

In this example, at Code 4, we can see that a total of medicines variable now exists. It's an integer value of 250. In the example we specified that the quantity must always be lesser than or equal to the total of medicines. We specified this as an **invariant**, therefore all routines of the class must preserve it. Before having a specified invariant one could assume that the quantity could be any value upper than 250 on any routine of the class, i.e., it didn't exist a

limit to the quantity of medicines. The invariant represents a general consistency constraint obligatory for all routines of the class. (Eiffel Software) So to preserve this property defined by the invariant, one has to implement the routine's body in a way to not violate what is stated in the invariant clause, in this example, the routine *addMedicine* must also ensure that the variable of *quantity* must not exceed the value defined by *total_medicines*.

So far we used invariants to express global properties of a single class, denominated by *class invariants*, but there is another concept within the invariants known as *system invariants* which describes instance properties that must be preserved by all routines from more than one class. For instances, let ***X*** and ***Y*** be two different classes. An invariant ***INV*** would be a system invariant if instances from both ***X*** and ***Y*** are affected by ***INV***. A system invariant is basically described like a class invariant and it is specified in a class that has references to ***X*** and ***Y*** objects.

Besides the earlier *Medicines* class, let's supposed that we have an *Appointments* class to manage appointments information. In the following example shown in Code 5, ***X*** and ***Y*** are exemplified respectively by the classes *Medicines* and *Appointments*, where *Medicines* is a class that manages *Medicine* objects and *Appointments* is a class that manages *Appointment* objects. The defined **invariant** is a system invariant because it affects instances of these two different classes. The invariant basically states that for every *Medicine* object instances obtained through the *Medicines* instance, their prescription date attribute (i.e., *meds.getDate(i)*) must be higher or equal to the respective *Appointment's* date (i.e., *apps.getDate(k)*), obtained through the *Appointments* instance. This compares the date of the medicine's prescription renewal with the date when the medicine was prescribed for the first time. That is, for all *Medicine* and *Appointment* instances if a *Medicine* instance has an *appointmentID* attribute equal to another *Appointment* instance *ID* attribute, then that *Medicine's* date must have a higher or equal value to the that appointment's date. This invariant restricts the value of a medicine's date making it dependable of the respective appointment's date.

```
class SERVICES create
   make
feature
   meds: MEDICINES
   apps: APPOINTMENTS
...
invariant
  forall( int i; i < meds.getMedicines().length && i >= 0;
        forall( int k; k < apps.getAppointments().length && k >= 0;
              meds.getAppointmentID(i) == apps.getID(k)
              ==>
              meds.getDate(i) >= apps.getDate(k)  ))

end -- class SERVICES
```

Code 5. Example of a Services class referencing Medicines and Appointments classes with a system invariant

Another concept that extends the notion to contracts, at a lower level, within the design by contract is the *inheritance*. The preconditions, postconditions and even invariants can be also inherited.

2.1.1.4. Inheritance

The concept of inheritance allied with the notion of contracts from design by contract brings us to a new level, as contracts can also be inherited by subclasses in terms of object-oriented programming. A routine's precondition and postcondition are inherited by their redefinitions in sub-classes as well as super-class invariants. This is actually the case in JML specifications (see Section 3) which can also be inherited. Although inheritance is one of the pillars of the object oriented paradigm flexibility, many programmers have the difficulty in use it correctly. (Júnior, Figuereido, & Guerrero, 2005) Through the inheritance mechanism one can create new classes from those already existent, and the behaviour from their routines doesn't necessarily have to be maintained by their sub-classes. It is possible to redefine the routines with a partial behaviour or even a complete distinct one. However, from these possibilities and the use of design by contract methods one could redefine a routine that produces an incompatible effect to the described routine's behaviour specification (contract) in the super-class. (Júnior, Figuereido, & Guerrero, 2005) This incompatible redefinition is a problem connected with the bad use of the inheritance, which design by contract helps to avoid in a way that we can redefine those routines as longs as they respect the established original contract defined in the respective inherited routines from the super-classes. (Júnior, Figuereido, & Guerrero, 2005)

For instance, let ***X*** and ***X1*** be two classes where ***X1*** is a sub-class of ***X**,* and ***Y*** any class communicating with an instance of type ***X***. Due to polymorphism, ***Y*** can actually be dealing with an instance of ***X1***. The developer of ***Y*** knows that he must respect the defined contract in ***X***, but he doesn't know of the existence of other classes inheriting ***X***. So, ***Y*** could discover only in runtime that he is communicating with ***X1***, and the contract of a certain inherited routine of ***X1*** could be different from the contract specified in the super-class ***X***. That is, ***Y*** could be calling for a routine under a certain contract, while in reality is communicating with another completely different. In fact there are two things that could make a class deteriorate its super-class contract specification (Júnior, Figuereido, & Guerrero, 2005):

1. A sub-class could make its precondition to be more restrictive than the one from the super-class, causing the risk of any calls previously considered correct by the client class ***Y***'s perspective (in a way that they satisfied the original conditions imposed to the client) to become violating the contract's rules.

2. A sub-class could be making its postconditions to be more permissive, returning a result less satisfactory than the promised to ***Y***.

Under the previous situation the client class **Y** could get "deceived" by a call that makes something unexpected. From this problem, we conclude that every contract specifications must be compatible with the original contract specifications, but nevertheless sub-classes have the right to improve them, i.e., by making its methods' postconditions stronger or making its methods' preconditions weaker. Besides the inheritance rules applied to preconditions and postconditions, also the inheritance mechanism has effect upon the invariants, in a way that these are passed to their inheritors. For instance, an invariant from ***X*** also would be inherited by ***X1***, and this is the case in JML where invariant specifications written in Java interfaces are inherited by the concrete Java classes implementing the interfaces.

The result of the inheritance concept, in which every instance of a class is also an instance of every ascendant class, is also logically valid for the contract specifications defined in the super-classes to be applied to their sub-classes. That is, a set of invariants of a certain class is the sum of all invariants from the ascendant hierarchy of inheritance. (Júnior, Figuereido, & Guerrero, 2005)

Another concept extending the notion of a contract is the treatment of exceptions within a contract between a client and a supplier.

2.3.1.5. Exceptions

As a routine in design by contract is seen like an implementation of a certain specification rather than just a piece of code, and as it is possible for that implementation to fail with respect to the specifications in runtime, then one can extend the notion of a contract to the exception handling. Besides errors in implementations, exceptions in a routine's behaviour can happen due to unpredictable events like hardware malfunctions or another external event. So, in these situations it becomes useful to use exceptional specifications attached to contract specifications to describe exceptional behaviours when some strategy for fulfil a contract doesn't succeed. By this definition and the notion of preconditions and postconditions from a contract, it is possible to establish the following rule: - A routine must not launch an exception when its preconditions is not fulfilled, as it doesn't denote a failure within the routine but it does for the routine's caller. When the routine fulfils its postconditions it must not launch an exception. – This is known as the *principle of exception*. (Júnior, Figuereido, & Guerrero, 2005)

As for the global properties from a class, routines and constructors must preserve and respect the invariants in both normal and abrupt terminations, that is, invariants are included in both normal and exceptional postconditions. (Júnior, Figuereido, & Guerrero, 2005)

2.2. Formal Methods in the Software Development Process

A formal software specification is a specification expressed in a language that has its semantics and syntax mathematically or logically defined. Based on the definition of Sommerville (Sommerville, 2000), the formal methods are a way of employing software correctness in software development processes. The need for a formal specification in a software development process means that we cannot solely rely in natural language to develop a system. The natural language is ambiguous and can lead to inconsistent and incomplete specifications. Formal specifications make possible the capture of software requirements unambiguously as part of a software engineering methodology. By using formal specifications, one might invest more effort in the early phases of software development cycle, especially in the requirement analysis phase. Nonetheless, the use of formal specifications reduces requirements errors as it forces a detailed analysis of them, and also helps to detect and resolve incompleteness and inconsistencies. Hence, the amount of rework due to requirements problems is reduced, and thus also the cost related to the implementation and validation phases. However, according to Sommerville (Sommerville, 2000), in the software engineering, the formal methods are not widely used as software development techniques, although their promise to increase the systems quality by supporting their correctly development according to the client's real needs. Eventually other software engineering techniques have surpassed the need for formal methods for various reasons that extend from the complexity and the incapability of formal methods in dealing with large-scale systems, to frequent changes in requirements and designs in practice. (Liu, Takahashi, Hayashi, & Nakayama, 2009) Sommerville (Sommerville, 2000) suggests that formal specification techniques have not been broadly used in industrial software development environments, because:

I. There is a **lack of methodologies and tools to support the use of formal methods** in software development. Barely minimal guidelines are provided on how to elicit and structure the requirements into formal notation. Lack of guidance makes it hard to developers use formal methods by themselves and from the lack of tools developers

have difficulties of applying formal methods into their development cycles, especially to develop, analyze and process large-scale specifications using formal specification languages. The production of well-defined guiding lines and supporting tools are needed.

II. The use of **formal methods requires the knowledge of discrete mathematics and symbolic logic**. Most of the developers (i.e., software engineers, programmers, and designers) have not been trained in techniques required to develop formal software specifications. Techniques have been tested by Japanese researchers over the last fifteen years in formal methods education programs for undergraduate and graduate students at universities as well as practitioners at companies. (Liu, Takahashi, Hayashi, & Nakayama, 2009)

III. The **formal specifications are an inappropriate tool for communications with the end user** at the later stages of requirements specification. More than the software developers, most end users who provide the requirements and approve their specifications are neither familiar nor comfortable with the formal specification languages. According to Sommerville (Sommerville, 2000), Hall suggests that one can paraphrase in natural language the formal specifications or use animated illustrations, that is, presenting the formal specifications in a form that can be understood by the client.

IV. The use of **formal specifications at initial stages may hold back the creative side of developers**, that is, having a poorly structured problem, the formal representations from it may restrain the developers from exploring alternatives. Formal specifications may not be an ideal tool for exploring and discovering the problem's structure. The problem may have to be studied and understood before being formalized.

V. The use of **formal specifications for development of user interfaces is hard**. With the current techniques is practically impossible for specifying interactive components of user interfaces. Also, some other system components are hard to specify like parallel processing systems, such interrupt driven-systems.

VI. Most of **software development managers are normally conservative** and reluctant in using techniques whose benefits are not yet well-known. The recompense by using formal methods is not immediate and it is hard to quantify. Nevertheless, Sommerville (Sommerville, 2000) concludes that when a conventional software development process (i.e., without using formal methods) is used, validation costs are more than 50% of the whole development costs, and implementation and design costs are the double of the specification cost. With the use of formal methods, the specification, implementation and design costs are almost equal and validation costs are considerably reduced to less than the development costs.

Knowing these difficulties in the wide acceptance of formal methods in software development, one has the challenge to integrate formal methods to the system development effort, especially in large-scale development projects. For this, viable strategies for supporting the integration of formal method techniques into the software development process are paramount important; without existing strategies it may be difficult to integrate formal methods into the real-world development project. Our JML-based strategy (described in Section 5) is a strong attempt to tackle some of the difficulties of making formal methods popular among the developers in the software industry. In the following, we describe how our strategy overcomes some of these difficulties.

To address the **"lack of methodologies and tools to support the use of formal methods in software development"** difficulty (see point I. – Section 2.2), our strategy provides guidelines for iteratively transforming informal functional requirements (given by the stakeholders) into formal specifications. By using this specification transformation process we can formally develop software applications. In our strategy, the specifications go along three stages: the informal, the semi-formal, and the formal ones. The informal specifications are given by the stakeholders, and in a middle stage are then structured into the semi-formal ones (see Section 5.2 for details about the semi-formal specifications creation). The semi-formal specifications are still written in natural language like the informal specifications, but mathematically and logically structured. They serve as an intermediate step for writing formal specifications. The formal specifications are written in JML, and they are produced from the semi-formal specifications. From the JML specifications, the Java programs are developed accordingly. Also, as the JML uses Java syntax, it is a formal specification language easy to be used by any developer with the minimal knowledge about the Java language. There are tools for supporting the validation of Java implementations against their respective JML specifications. The most popular tools in the field are the JML Common Tools (see Section 3.3). Our JML-based strategy provides not only guidance for incorporating formal specifications, but also it benefits from the existence of a variety of supporting tools and the usage of a specification language easy to understand by any Java developer.

To address the difficulty of ***"formal specifications as an inappropriate tool for communicating with the end user"*** (see point III. – Section 2.2), our strategy uses semi-formal specifications as a means of communication between formal specifications and end users. That is, as the semi-formal specifications are closer to formal specifications than the informal ones, yet written in a structured natural language, it becomes easy to communicate with the end user about the system specifications through the semi-formal specifications. Our strategy recommends structuring the semi-formal specifications in a way like the JML specifications, while being written in natural language. For example, the semi-formal specification *if <event/condition> then <restriction/rule>* can be mapped directly to a JML specification *requires <precondition> ensures <postcondition>,* and vice-versa. The following example presents this relation between semi-formal and formal specification stages:

- Semi-formal specification for event *addName*:

  IF *length of name LESS OR EQUAL TO 50* ***THEN*** *stored_name EQUALS name.*

- Formal specification (JML) for event *addName*:

```
...
requires name.length() <= 50;
...
ensures stored_name == name;
...
```

Where **IF ⇔ requires** and **THEN ⇔ ensures**. In our strategy, the semi-formal IF statement is similar to the requires statement from JML, and the THEN is similar to the ensures from JML. As the semi-formal specification, written in natural, can be easily understood by an end user, then we can communicate with them the formal specifications. We conclude that our JML-based strategy provides a first solution for the communication of formal specifications to the end-users.

To address the difficulty of ***"the use of formal specifications at initial stages may hold back the creative side of developers"*** (see point IV. – Section 2.2), our strategy does not make use

of formal specification at the initial stages of a software development process. At the initial stage of the software development our strategy recommends producing domain concepts to understand the problem. After having domain concepts, one can start writing use cases, and then designing informal functional requirements describing the rules. These (unstructured) informal specifications are then ported into semi-formal specifications. These semi-formal specifications are still in natural language and the developers can still discuss them with the stakeholders while giving them enough space for their creative sides. The semi-formal specifications can easily be ported into formal specifications just before starting implementing the system. To provide support to the creative side in later stages of the development, our strategy recommends designing the system using Java interfaces with JML specifications, giving the implementation a higher level of abstraction. During implementation, one can implement the concrete classes (and their methods) in various ways provided that they respect what is specified in the implemented Java interfaces or abstract classes. Furthermore, the JML specifications make use of abstract variables declared as JML abstract types. These JML abstract variables can abstract a complex data structure, allowing programmers to implement those complex data structures as they desire while in concordance with the formal specifications (see Section 3.2 for further details about abstract variables and JML abstract data types).

For addressing the difficulty of **"the use of formal specifications for development of user interfaces is hard"** (see point V. – Section 2.2), our strategy makes use of the B. Meyer's Design by Contract principles. As our strategy complies with the design by contract principles, we can formally describe the behaviour of components with JML and then when implementing a client routine we must respect the contract conditions when programming a call on a specified supplier routine. The use of JML supports a style of programming by contract. By following our strategy we can end up with components that have formal specifications to describe contracts for their methods (JML method functional specifications) or even classes (JML invariants). For instance, these components can be user interface components, and as they use JML specifications, one can implement a system around them by respecting the specifications. That is, all the calls made for the specified components must have to respect their contracts. By employing our JML-based strategy it is possible to develop user interfaces like any other Java program, as it is possible to describe components behaviours.

3. The Java Modelling Language (JML)

JML is a specification language for Java, which as a tool provides support for B. Meyer's design by contract principles (Meyer, 1992). JML was started by Gary Leavens and his team at Iowa State University, but is now an academic community effort with many people involved through the development of tools providing support for the language (The ESC/Java 2 Tool; The Jack Tool; The Krakatoa Tool; van den Berg & Jacobs, 2001). All the concepts discussed in the Design by Contract section (see Section 2.1.1), that is, the notion of contracts along with its preconditions and postconditions; and the concepts of invariants, inheritance and exceptions, also apply to JML.

3.1. The JML Specifications

JML specifications use Java syntax, and are embedded in Java code between special marked comments `/*@ ... */` or after `//@`. A simple JML specification for a Java class consists of pre- and postconditions added to its methods, and class invariants restricting the possible states of class instances. Specifications for method pre- and postconditions are embedded as

comments immediately before method declarations. JML predicates are first-order logic predicates formed of side-effect free Java boolean expressions and several specification-only JML constructs. Because of this side-effect restriction, Java operators like `++` and `--` are not allowed in JML specifications.

JML provides notations for forward and backward logical implications, `==>` and `<==`, for non-equivalence `<=!=>`, and for logical `or` and logical `and`, `||` and `&&`.

The JML notations for the standard universal and existential quantifiers are `(\forall T x; E)` and `(\exists T x; E)`, where T x; declares a variable x of type T, and E is the expression that must hold for every (some) value of type T. The expressions `(\forallT x; P; Q)` and `(\exists T x; P; Q)` are equivalent to `(\forall T x; P ==> Q)` and `(\exists T x; P && Q)`, respectively.

The JML numerical quantifier `(\num_of T x; P; Q)` returns the number of variables x of type T that make both predicates P and Q true; `(\max T x; P; E)` returns the maximum value of the expression E where its variables satisfy the range P; `(\sum T x; P; E)` returns the sum of possible values of E where its variables satisfy the range P.

JML provides specifications for several mathematical types such assets, sequences, functions and relations. As JML is a tool to employ design by contract methods, there is some mechanisms used to support contracts like the specification of method's preconditions and postconditions through the use of respectively the keywords `requires` and `ensures`; the specification of invariants by using the JML keyword `invariant`; the specification of exceptional behaviours to describe how to deal with unexpected behaviours; and also the JML specifications are inherited by sub-classes, i.e., sub-class objects must satisfy super-class invariants, and subclass methods must obey the specifications of all super-class methods that they override. In the following, we briefly review JML specification constructs. A brief description of some JML expressions used in specification can be seen in Section 3.1.1, but the reader is invited to consult (Leavens G. , 2008) for a full introduction to JML.

3.1.1. JML Expressions

In this section we present some of the common JML expressions and a simple example based on the pop() method of a Stack class.

Table 2. Some JML expressions

`requires P`	Specifies a method pre-condition P, which must be true when the method is called. Predicate P is a valid JML predicate.
`ensures Q`	Specifies a normal method post-condition Q. It says that if the method terminates in a normal state, *i.e.* without throwing an exception, then the predicate Q will hold in that state. Predicate Q is a valid JML predicate.
`signals (E e) R`	Specifies an exceptional method post-condition R. It says that if the method throws an exception e of type E, a subtype of `java.lang.Exception`, then the JML predicate R must hold. Predicate R is a valid JML predicate. JML allows the use of the alternative clause `exsures` for signals.
`normal_behavior`	Specifies that if the method precondition holds in the pre-

	state of the method, then it will always terminate in a normal state, and the normal post-condition will hold in this state.
`exceptional_behavior`	Specifies that if the method pre-condition holds in the pre-state of the method, then it will always terminate in an exceptional state, throwing a `java.lang.Exception`, and the corresponding exceptional post-condition will hold in this state.
`assignable L`	Specifies that the method may only modify location L. Any other location not listed in L may therefore not be modified. This must be true for both normal and exceptional post-conditions. Two special assignable specifications exist, `assignable \nothing`, which specifies that the method modifies no location, and `assignable \everything`, which specifies that the method may modify any location. JML allows the use of the alternative clauses modifies and modifiable for assignable.
`\old(e)`	Refers to the value of the expression e in the pre-state of a method. This specification can only be used in normal or exceptional method post-condition specifications.
`\fresh(e)`	Says that e is not null and was not allocated in the pre-state of the method.
`\result`	Represents the value returned by a method. It can only be used in a normal or an exceptional method post-condition.
`invariant I`	Declares a class invariant I. In JML, class invariants must be established by the class constructors, and must hold after any public method is called. Invariants can temporally be broken inside methods, but must be re-established before returning from them.

The following example shows how a JML specification can be used to specify the method pop().

```
public interface Stack {
//@  public model instance JMLObjectSequence stack;

/*@   public normal_behavior
@           requires !stack.isEmpty();
@           assignable size, stack;
@           ensures stack.equals(\old(stack.trailer()));
@    also
@     public exceptional_behavior
@           requires stack.isEmpty();
@           assignable \nothing;
@           signals(java.lang.Exception e) true;
@*/
public void pop( ) throws java.lang.Exception;
}
```

Code 6. Example of how JML can be used to specify a method

In the example shown in Code 6, we can see that method pop() has been given a normal and an exceptional behaviour formal specification. For the normal behaviour, the precondition is defined by the `requires` clause, which states that the stack must not be empty. Then the `assignable` clause specifies that the size and stack instances may suffer a change, that is, only the locations named through the assignable clause, and locations in the data groups associated with these locations, can be assigned to during the execution of the method. A JML assignable clause can be used in a method contract to specify which parts of the system state may change as the result of the method execution. The postcondition in the normal behaviour is defined by the `ensures` clause, which states that the stack will be equal to a portion of the old stack after the execution of pop(). The exceptional behaviour if the stack is empty when attempting to call pop() an exception will be thrown. The `assignable` clause in this case is `\nothing` because nothing is changed within pop() and the signals clause specifies a condition that will be true when an exception of type java.lang.Exception is thrown.

3.2. Abstract Variables

To have a higher level of abstraction in specifications, JML provides support for abstract variables. These are variables that exist at the level of the specification, but not in the implementation. Declarations of abstract variables have the same format as declarations of normal variables, but are preceded by the keyword `model`. As we can't declare concrete variables in interfaces, the abstract variables can be used in interfaces and abstract Java classes to describe abstractly the distinct data types used in the application. The abstract variables can be used to support the writing of correct code for concrete classes that implement the interfaces and the abstract Java classes. In the following Code 7 example we demonstrate a declaration of an abstract variable named *dosage_model* in interface Medicine. The abstract variable *dosage_model* represents the dosage quantity of a medicine.

```
public interface Medicine {
...
//@ public model instance double dosage_model;
...
}
```

Code 7. Example of how JML can declare an abstract variable

Abstract variables can be related to concrete variables (or other abstract variables) by a `represents` clause. A represents clause specifies how the value of an abstract variable can be calculated from the values of the concrete variables (variables at the implementation level). In the following Code 8 example it's demonstrated how we can relate an abstract variable with a concrete expression involving a concrete variable.

```
public class Medicine_Impl implements Medicine {
...
public byte[] dosage;//@ in dosage_model;
/*@ public represents
  @         dosage_model <- dosage[0] + dosage[1]*0.1;
  @*/
...
}
```

Code 8. Example of how JML abstract variables can be represented by concrete values

In the above example, the abstract variable `dosage_model` is related with the expression "`dosage[0] + dosage[1]*0.1`", which maps the values in the byte array dosage into the double value calculated as the sum of the all the values in the array. For specifications purpose we can treat the dosage of a medicine like a double value, but in reality it can be implemented as an array of primitive bytes. In this case, the use of abstract variables gives a level of abstraction that allows us to implement a medicine's dosage information in different ways as long as it respects the specifications.

Abstract variable specifications for interfaces and for abstract classes do not need to be written down again in implementing classes and sub-classes, since JML specifications are inherited by sub-classes and by implementing classes. This ensures behavioural sub-typing. That is, a sub-class object can always be used where a super-class object is expected. Therefore, a sub-class satisfies super-class invariants, and sub-class methods obey the specifications of super-class methods.

For abstracting complex data structures, i.e., modelling complex data structures into specifications, there are model data types provided by the JML, also known as JML abstract data types.

3.2.1. JML Abstract Data Types

The Java Modelling Language (JML) also provides abstract data types from the package `org.jmlspecs.models` to abstract complex data structures. Based on the description of Leavens (Iowa State University, 2002), this package is a collection of types with immutable objects. An object is *immutable* if it has no time-varying state. The types of the immutable objects in this package are all *`pure`*, meaning that none of their specified methods have any user-visible side-effects (although a few inherited from Object do have side effects). Their `pure` methods are designed for use in JML specifications. When using such methods we have to do something with the result returned by the method, as in functional programming. The original object's state is never changed by a pure method. For example, to insert an element *e*, into a set *s*, one might execute *s*.insert(*e*), but this does not change the object *s* in any way, instead, it returns a set that contains all the old elements of *s* as well as *e*. At first we shouldn't worry about the time and space used to make such set, because specifications are not mainly designed to be executed. However, there are justifiable reasons to worry about the efficiency of executing specifications for testing and debugging purposes.

In the following list are described some abstract data types that can be used while declaring abstract variables in JML specifications. The reader is invited to consult (Iowa State University, 2002) for a complete description of JML model data types.

`JMLObjectSequence` – This class defines immutable sequences of objects, including a series of pure methods for sequence manipulation. For example, `insertFront()`, `insertBack()`, `itemAt(int i)`. This type can be used to declare abstract variables to model complex data structures containing objects.

`JMLValueSequence` – This class defines immutable sequences of values, and also including a series of pure methods for value sequence manipulation. This type can be used to declare abstract variables to model complex data structures containing values, such as characters of a String or Integer values of an array.

`JMLEqualsSequence` – This class is similar to `JMLObjectSequence` but has an "`.equals`" method to compare elements.

`JMLType` – There are classes which implements `JMLType` to reflect Java types like `JMLByte` to reflect `Byte`, `JMLChar` to reflect characters, `JMLFloat` to reflect `float` type, etc.

By using these data types in the specifications, one can abstract the way programmers can represent data structures. For example, an abstract variable of the type `JMLObjectSequence` abstracts a complex data structure to hold object instances, which besides simplifying the JML specifications it also gives the freedom, through their representation, of implementing concrete data structures in various ways (object arrays, stacks, queues, etc.) as long as the specifications are respected.

3.3. The JML Common Tools

The JML common tools (Leavens G. T., 2008) is the most basic suite of tools providing support to run-time assertion checking of JML-specified Java programs. The suite includes ***jml***, ***jmlc***, ***jmlunit*** and ***jmlrac***. The jml tool checks the JML specifications for syntax errors. The *jmlc* tool compiles JML-specified Java programs into a Java byte-code that includes instructions for checking JML specifications at run-time. The *jmlunit* tool generates *JUnit* unit tests code from JML specifications and uses JML specifications processed by *jmlc* to determine whether the code being tested is correct or not. Test drivers are run by using the *jmlrac* tool, a modified version of the Java command that refers to appropriate runtime assertion checking libraries.

The JML common tools make it possible the automation of regression testing from the precise and correct JML characterization of a software system. The quality and the coverage of the testing carried out by JML depend on the quality of the JML specifications. The runtime assertion checking with JML is sound, i.e., no false reports are generated. The checking is however incomplete, e.g., users can write informal descriptions in JML specifications. The completeness of the checking performed by JML depends on the quality of the specifications and the test data provided. These JML Common Tools are available at (Leavens G.).

4. Related Work

In this section we start by presenting the various proposed types of strategies for incorporating formal specifications into software development processes. Further, we present the SOFL language that is used for constructing formal specifications and for the software development process, and then we present a framework named ConGu that is aimed at providing support to the checking of formal specifications and Java code, that is, the verification & validation phase.

4.1. Formal Methods Strategies for Software Development Processes

The way people formalize informal software requirements (i.e., the client's requirements to a system to be developed) can be categorized into several strategy types. Some of the proposed strategies suggest going directly from informal specifications (i.e. high level, natural language) to formal specifications (i.e., low level, more mathematical language) making the software development's specification activity being in the formal domain from the beginning. For example, according to Kemmerer (Kemmerer, 1990) through his "Integrated" approach which defines that formal methods is completely integrated into the development cycle, we use critical requirements written in English and stated in precise mathematical terms to

describe the system's behaviour without giving too much implementation details, so later they can be incrementally detailed until the system can be coded according to them. Also Jones (Fraser, Kumar, & Vaishnavi, 1994) uses a similar process by suggesting that proof obligations of VDM decomposition rules can stimulate design steps. Others like Miriyala and Harandi, and Wing (Fraser, Kumar, & Vaishnavi, 1994), have proposed strategies where high-level formal specifications of the system can be derived directly from a precise English statement of critical requirements. A strategy that goes directly from informal specifications to a formal specification without any transitional step is known by using a *direct formalization process*.

However there is another type of formalization strategy used to introduce formal methods into software development processes. Rather than using a direct formalization process, one can define intermediate steps that help to move from the informal the initial natural language to formal specifications. Through this kind of strategy, we resort to one or more semi-formal specifications which provide us with evolutionary steps between the informal natural language specification and the formal specifications. This type of strategy, which starts from informal specifications and moves to formal ones through intermediate specifications, is known as *transitional formalization process*. (Fraser, Kumar, & Vaishnavi, 1994) We can say that the transitional formalization of the specifications can be divided into three degrees: informal, semi-formal and formal. At the informal stage, the specifications are incomplete sets of rules to constraint the system to be developed, usually written in natural language or presented as unstructured pictures that can lead to ambiguous meanings and introduce inconsistencies in the system or its incompleteness. At the semi-formal state, the informal specifications are evolved so as to become closer to the formal ones. Although the semi-formal specifications still use natural language, they are presented with a defined syntax and written in a mathematical form or illustrated in a diagrammatic technique that defines precise rules. Through this technique we are clearing out possible inconsistencies and also detecting possible incomplete specifications. The semi-formal specifications are viewed as helpers to achieve formal specifications from the informal ones. At the formal state, the specifications become closer to code. These formal specifications have a rigorous defined syntax and semantics and can be used to automatically test the code against the specifications (the informal ones evolved into formal specifications) given by the clients. (Fraser, Kumar, & Vaishnavi, 1994) An example of a strategy using a transitional formalization process is the strategy proposed by Kemmerer (Kemmerer, 1990) through the "Parallel" approach. His proposed formalization process approach involves the use of standard development methods (to develop semi-formal requirements) as intermediate steps from which formal specifications are derived.

The strategy proposed in this thesis work is based on a transitional formalization process that integrates formal specifications into the software development process. The strategy is similar to Kemmerer's "Parallel" approach in that we also define an intermediate step that introduces semi-formal specifications before writing JML formal specifications. This transitional process runs in parallel with the development process itself.

4.2. SOFL

The SOFL (Structured Object-based Formal Language) is a language proposed by Liu, Offutt, Ho-Stuart, Sun and Ohba in 1997 (Liu, Offutt, Ho-Stuart, Sun, & Ohba, 1997) to address the problems of lack of formal methods wide acceptance by the industry, namely, the need for integration of formal methods into the software development processes, the requirement of significant abstraction and mathematical skills, and the lack of tools to support the entire formal software development process. Developing a software system using SOFL consists in three separate activities: the requirements specification, the design, and the implementation.

The SOFL provides a specification language, a method, and a systematic process for the development of software systems.

SOFL = Specification Language + Method + Software Process Model

As a specification language, the SOFL integrates Data Flow Diagrams, Petri Nets, and VDM-SL. Data Flow Diagrams provide notation for expressing the overall architecture of a system; Petri Nets are used to provide an operational semantics for the Data Flow Diagrams; and VDM-SL with certain syntactical reason and extension is used for specifying the behaviour of processes occurring in the related formalized Data Flow Diagrams.

As a method, the SOFL consists on a three-step approach, i.e., informal, semi-formal, and formal specifications, for the development of system specifications in a structured way (including requirements, and abstract design specifications) and transformation from a structured abstract design into a more detailed object-oriented design and then implementation of the system. Additionally, the SOFL also offers means for verifying and validating specifications and programs. When using the SOFL methodology, engineers construct the initial condition Data Flow Diagrams and specification modules, and then they use decomposition, evolution, and transformation to construct an object-based design from the structured requirements specifications, at the end they finally transform the design to implementation. (Liu, Offutt, Ho-Stuart, Sun, & Ohba, 1997)

SOFL also provides a software process model that supports a systematic way to develop software systems. The SOFL's software process model includes three main features:

1. The informal and semi-formal specifications are used for capturing and documenting user functional requirements, while the formal specification is used for abstract design.
2. The importance of the development's evolution of informal, semi-formal, and formal abstract design specifications, and refinement for the development of detailed design and implementation.
3. The use of rigorous reviews and tests to verify and validate specifications and program applications.

In SOFL, the process of constructing formal specifications occurs in the three-step, informal, semi-formal and formal specifications, much like in our proposed JML-base strategy. First, the informal specifications are written and then in the semi-formal specification stage these informal specifications must be organized as sets of inter-related modules conforming to the SOFL syntax. The involved data resources are represented as variables and given the appropriate data types. After having variables, pre and postconditions are written in SOFL syntax for specifying processes, or Data Flow Diagrams can be used for the same effect, however the latter option may not be complete for specifying a process. (Liu, Offutt, Ho-Stuart, Sun, & Ohba, 1997)

What distinguishes our strategy from SOFL is that it has a specification syntax more abstract for specifying systems in any OO programming language, and it allows the use of hierarchal diagrams to specify the requirements. In our JML-based strategy we use only textual specifications from the beginning to the end of the specification construction and these textual specifications go through an evolution from the informal requirements to JML formal specifications, where the client is still capable of following them to the semi-formal stage. That is, in our semi-formal stage we still use natural language but in a structured and more mathematical manner which is yet understandable by the client, creating a bridge between the informal functional requirements and JML formal specifications. Also, the JML syntax uses Java

syntax, so it is more adequate to specify Java programs than by SOFL, which is more abstract. Another difference is that SOFL can't be integrated with the implementation code like JML and Java programs. As the JML has the particularity of being a specification language for the Java programming language, it can be integrated with the Java code itself (i.e., the specifications can be written in the same Java code files) and it can be used for testing the code and documenting it at the same time.

People may use either of our strategy or SOFL to develop a Java system. Nonetheless, our strategy is more cost-effective and straight-forward for a Java programmer because JML uses Java syntax.

4.3. ConGu

The ConGu (Contract Guided System Development) is a project that has the purpose of providing a framework for creating property-driven algebraic specifications to fully test Java implementations. The ConGu project adopted a property-driven algebraic approach to specifications rather than the model-based approach to Design-by-Contract like those proposed for JML, Z, Larch and AsmL. (Vasconcelos, Nunes, & Lopes, 2008) The basic idea of an algebraic specification is to specify data types independently of any representation or programming language. An algebraic specification is constituted by a set of sorts, a set of constants and operations symbols, and a set of conditional equations or short equations. Each sort represents a domain of a data structure and each operation symbol represents an operation. More precisely, an operation symbol declaration consists on an operation name, a list of argument sorts and a range sort or a result sort. Operation symbols can be combined to write a specification. Having names for domains of data structures, and declarations for operations, the only thing needed to write a specification is the description of what the operations should do, which is what serves the last constituent part of an algebraic specification, the set of conditional equations (or short equations), which provides the needed descriptions. (Classen, Ehrig, & Wolz, 1993, p. 8)

The main components of ConGu are specifications, modules, and refinements. The specifications used are property-driven algebraic as they define sort and operations on those sorts. The ConGu supports partial specifications whose operations can be interpreted by partial functions with conditional axioms. Each specification defines a single sort that may be defined involving an operation or other sorts like parameters or results of operations. Another component of ConGu is the notion of modules. The notion of module is used for denoting a set of specifications to self-contained them. In order to check the specifications against the Java classes, for violation of axioms or domain restrictions, the specifications logic and Java classes must be bridged, and this is where the ConGu refinement component enters. In ConGu, the refinement mappings have to be defined indicating which sort is implemented by which class, and which operation is implemented by which method. In the refinement mappings activity, the knowledge about concrete representations on the classes isn't required. (Vasconcelos, Nunes, & Lopes, 2008)

The Congu can be used to support a formal development process, but not the entire process. Unlike our proposed JML-based strategy and SOFL, the ConGu as a framework does not support the process of constructing formal specifications from the informal. The ConGu is focused on the validation & verification phase of software development processes. The main aspect of the ConGu approach is to ease the problem of testing implementations against the respective property-driven algebraic specifications to the run-time monitoring of contract annotated classes, supported today by several run-time assertion- checking tools. (Vasconcelos, Nunes, & Lopes, 2008) The ConGu has tool available as a plug-in for the Eclipse

IDE. This tool allows users to test Java classes against a module of specifications to check for runtime axiom violations. The tool reads algebraic specifications and a mapping relating specifications and Java entities, and produces a number of classes that are used to test the original implementation against the given specifications, in a way that is transparent to the user. By using this ConGu technique all specification properties are checked against implementations because monitorable JML contracts are generated to cover them all. (Vasconcelos, Nunes, & Lopes, 2008)

5. The JML-based Strategy for Software Development of Java Programs

This section describes our strategy to incorporate formal specifications into the software development of Java programs. We have developed a strategy for evolving informal functional requirements into formal specifications. This strategy can be employed as part of existing object-oriented software development methodologies. In particular, the strategy suits Bertrand Meyer's design-by-contract principles (Meyer, 1992), which lie on the core of the JML language and JML-based formal methods tools. Hence, software developers must define precise interface specifications for underlying software components, based upon data type's theory and the conceptual metaphor of software contract. The strategy is part of an engineering integrated effort whereby software development is conducted in parallel with a *formal specification* pseudo-phase (see Figure 1) Therefore, JML specifications are evolved from informal requirements and written in parallel with the development of the application itself. In Figure 1, the presented software development process consists in four phases, namely, analysis, design, implementation, and validation-and-verification. In the same spirit of the methodology introduced by Meyer (Meyer, 1997)(see Chapter 11), we do not restrict any phase of the software development cycle to occur before or after any other phase, so the arrows 1 to 5 in Figure 1 convey information on usage rather than on precedence in time.

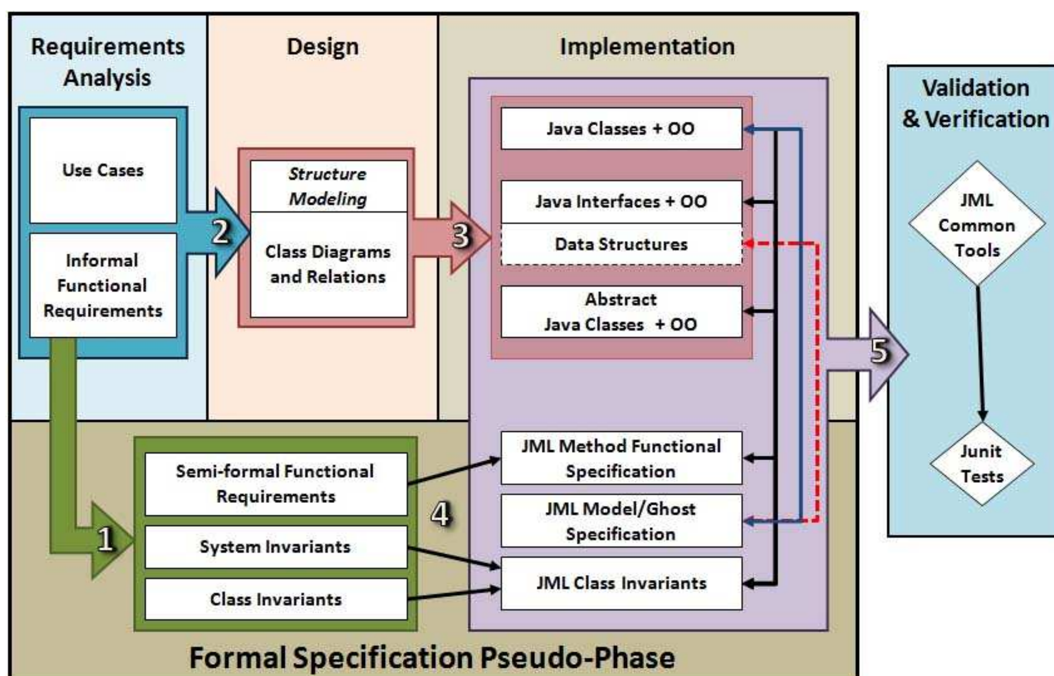

Figure 1. The Software Development Process

During the **analysis phase** - described in Section 5.1 - requirements are gathered, and two documents are produced, namely, the use cases document and the "informal" functional requirements document (i.e., functional requirements written in plain English). As informal functional requirements are expressed in a natural language, ambiguities and inconsistencies can be introduced during the analysis phase. Hence, during the formal specification pseudo-phase, the informal functional requirements document is first evolved into a "semi-formal" requirements document (see Figure 1 – arrow 1), and then ported into (formal) JML specifications (see Figure 1 – arrow 4). The process of passing the informal functional requirements into semi-formal requirements is described in Section 5.2. The semi-formal requirements document is composed of three documents. The semi-formal functional requirements document (later ported into JML method specifications), the class invariant document, and the system invariant document (these two are later ported into JML class invariant specifications). Having formal specifications expressed in JML makes it possible to use JML-based formal methods tools to check for inconsistencies. Evolving the informal functional requirements document into the semi-formal one involves expressing informal requirements into an "if <event/condition> then <restriction/rule>" form (see Section 5.2.1 for more details about semi-formal functional requirements) and class/system invariants are created by expressing the respective informal functional requirements into a restriction/rule written in natural language but more formalized, and normally without making references to conditions, for example, expressions without "if… then…" indication an obligation, rule or restriction like "...mustn't…", "…must…" (For more details on the class and system invariants, see respectively Sections 5.2.2 and 5.2.3). The process of going from the informal functional requirements to semi-formal specifications is described in Section 5.2.

During the **design phase** - described in Section 5.3 - the requirements gathered from the analysis are used to define the structure of the system (see Figure 1 – arrow 2). This structure is later used to write classes, their attributes, their methods, and the relations among them (see Figure 1 – arrow 3). These classes are later formally specified with the JML specifications, yield during the formal specification pseudo-phase.

During the **implementation phase** - described in Section 5.4 - we start by writing Java interfaces and Java abstract classes (from the model structure designed in the design phase, see Figure 1 – arrow 3). From the semi-formal requirements document, JML functional specifications are furnished to the (abstract) methods in these interfaces and classes in Java, and JML class invariants are provided to model global properties of the system. Additionally, JML abstract variable specifications (see Section 3.2 for a description of abstract variables) serve to describe the distinct abstract data types used in the application and how they are manipulated through class inheritance, i.e., abstract variable specifications are used to represent concrete data structures. JML specifications provide support to the writing of correct code for concrete classes implementing the interfaces and the abstract Java classes, but besides that, JML specifications also provide support to a business contract programming style of programming, in accordance with Bertrand Meyer's design by contract principles (see Section 2.1.1 for Design by Contract details).

Finally, during the **validation-and-verification phase** - described in Section 5.5 - the implementation is checked against the specifications (see Figure 1 – arrow 5). This phase can occur iteratively with the implementation phase, with the purpose of developing correctly the application while checking if it is in concordance with its specifications. We employ the JML common tools to do this checking (see Section 3.3). As the methodology is iterative, it is possible to go back to a previous phase and make amendments to JML specifications or code. Notice that inconsistencies can be detected before an implementation for the system is

written. For instance, Java interfaces and Java abstract classes are checked against JML specifications, obtained from the formal specification pseudo-phase, before writing an implementation for those classes and interfaces.

In summary, the proposed strategy is based on JML to incorporate formal specifications into software development processes to support the production of system applications in a way that reflects the client's needs, i.e., correct systems. By employing a specification formalization strategy, informal requirements are evolved from natural language to JML specifications, which provide a high level of formality. The purpose of achieving this high level of formality is to provide support to development of correct systems. JML specifications further serve a complementary purpose, as they also play the role of a precise documentation of the application. For instance, the JML specification of an abstract class precisely describes what the implementation (perhaps written by an external programmer) must be. In software development projects, JML facilitates the communication between developers in a way that it unambiguously describes the expected behaviour of classes, methods, and data structures.

In the following sections we present each development phase with the employment of the strategy to incorporate formal specifications along the process. A running example of the strategy employment for the development of an application is described in Section 6.

5.1. Requirements Analysis

In this Section we describe the first phase of the software development known as the requirements analysis phase. At this phase we get, in an informal way, all the rules and requirements that are expected for the purpose and functionality of the final system (i.e., the product of our development). Also, this phase is the very first step to build the formal specification. All the informal requirements obtained will serve as a base to formulate the JML formal specifications in the formal specification pseudo-phase that follows in parallel the development process.

The first thing to do in the requirements analysis is to extract information from the client and comprehend the concepts of the domain where the system to be developed will work and its purpose. These concepts are things related to the domain where the system will be applied. For example, in case of the HealthCard the concepts could be like “appointments”, “patient”, “doctor”, “health problem”, “diagnostic” and “medicine”. These concepts give us an insight about the environment where the system will work, providing us the basis to communicate with the client and formulate with him the possible use cases in a next stage. One must notice that these concepts eventually will become data entities to be managed by the system; moreover these domain concepts may be later represented as JML abstract variables while formally specifying the system, as the abstract variables normally represent data to be managed in a concrete implementation (see Section 3.2 for information on abstract variables).

The next thing to do, after getting some background about the problem’s domain, is to write the use cases. Like the domain concepts, the use cases are formulated conjointly with the stakeholders (i.e., client, specialized people, etc), through brainstorming or by reading documentation about the domain. These use cases are to be reflected into future methods and functionalities of the system to be developed. The use cases may include some additional textual information like scenarios or activity diagrams. These additional documents give some extra details about the use cases by describing their usage, while clearing some existing ambiguity. For example, a scenario for the use case “Scheduling an appointment” could give some extra information about the data that should be passed to schedule an appointment, like for instance a date, a local, or a doctor’s name. The use cases are formulated as a mean of communication between stakeholders and developers to give an idea of what functionalities

and usage the future system will have within its domain. Afterwards, in the Design phase, these use cases are used to originate a major part of the system's classes and their methods. For example, for the use case "Scheduling an appointment" the method *addAppointment(...)* will emerge in the Design phase. The additional information of the use case can state the entry parameters of the method *addAppoiment*, like for example *addAppointment(date, hour, local, doctor, type)*.

Having formulated domain concepts and use cases are a halfway to start developing a system. As the objective with this proposed strategy is to formally develop a correct application, then rules of operational behaviour must exist to create formal specifications for methods or classes, describing their proprieties and behaviours. These rules basically describe use cases behaviours and limitations. So, the last thing to do in this phase is the description of informal functional requirements, which dictates those needed rules (i.e., to be ported into formal specifications). These requirements are rules (i.e., specifications) written in an informal way that the future system must hold and respect for its purpose, functionality and usability. For example, an informal functional requirement for the HealthCard could be like this: - *"To schedule an appointment, it must be inserted a date, an hour, a local, a doctor or type of appointment"*. Later, these requirements with the combination of the use cases will be used to design the structure model of the system, and also are to be used to formulate semi-formal requirements, system invariants and class invariants, which will become the JML formal specification for supporting the correct development of the system.

It is important to remember that developers are free to return to this phase when on later stages, as this software development phases are iterative. The requirements phase is exemplified in the running example in Section 6.3.1.

5.2. From Informal Functional Requirements to Semi-Formal Specifications

This step occurs during the first stage of the *formal specifications pseudo-phase*, where from the informal functional requirements it can be identified and extracted the semi-formal requirements, and the system and class invariants. These three documents will serve as a base to write down the JML formal specifications of the Java implementation code.

The semi-formal requirements are written in natural language but expressed in a more mathematical and logical form, suitable to be used into JML specifications. In a later stage of the *formal specifications pseudo-phase,* these semi-formal requirements are expressed as JML methods preconditions and postconditions.

At this step, the system and class invariants are identified and written in a semi-formal way. These invariants come from requirements that tend to restrict properties or to impose some general limits of the system. Eventually these kind of informal requirements are to become JML class invariants. The system invariants express global properties of the system classes' instances which must be preserved by all routines, and class invariants express the same thing, but for the respective class only. Although, in JML there isn't a direct way of expressing system invariants, these will be identified as system invariants from the informal functional requirements but later they will be expressed simply as JML class invariants.

Basically, at the end of this step it is required to have three documents. One document with semi-formal requirements which will support method's preconditions and postconditions and two documents with a list of informal requirements classified as class invariants and system invariants.

5.2.1. Semi-Formal Functional Requirements

Some of the informal functional requirements are evolved into a more mathematical form. Yet expressed in natural language, this new form brings requirements closer to JML method specifications. However, the process of evolving informal functional requirements into this new form is not linear, and it requires some expertise and ingenuity. For almost of the informal functional requirements that describe some system's functionality (i.e., method or operation) executed under certain conditions, the general form of a semi-formal functional requirement is "if <event/condition> then <restriction/rule>", in which the guard is an event or a condition that triggers a rule that restricts (changes) the current state of the system. This rule can be regarded as the body of a method in a class, and the condition as the pre-condition under which this rule may be triggered. This new form is closer to JML specification, and the principles advocated by the design-by-contract. For example, considering an informal functional requirement like *"To schedule an appointment, it must be inserted a date, an hour, a local, a doctor or type of appointment"*, one can clearly associate it with a system's operation. In this case the operation is the one obtained from the use case of adding an appointment where certain data must be passed when scheduling an appointment. The semi-formal taken from this informal function requirement for the event of adding an appointment would be something like this:

IF *<date NOT EQUALS null AND hour NOT EQUALS null AND local NOT EQUALS null AND (doctor NOT EQUALS null OR type NOT EQUALS null)>*

THEN *<date EQUALS date_model AND hour EQUALS hour_model AND local EQUALS local_model AND (doctor EQUALS doctor_model OR type EQUALS type_model)>*

The above expression form is a suggestion on how a semi-formal functional requirement could be written. The semi-formal functional requirement expression can be written in any form desired, as long as it is mathematically and logically structured as the above example. It is highly recommended that the semi-formal specifications are written in a form understandable by the clients (i.e., stakeholders) and at the same time structured in a manner that it can be easily mapped into a JML specification. As can be seen, the previous expression is still written in natural language but in a structured form. The expression indicates the conditions under which the operation of adding an appointment must hold. The first statement, the IF statement, indicates the preconditions for adding an appointment that must be respected, and the second statement, the THEN statement, dictates the postconditions that must hold after executing an operation of adding an appointment. One must notice the *"***_model"* fields written in the second statement. It is here that we begin to think about the abstract variables. The fields given in the first statement are entry parameters of an operation, and the model fields given in the second statement are abstract variables that represent concrete data from a certain class. The second statement tells us that the given parameters should be stored, i.e., at the end of the operation, each concrete data represented by the abstract variables must be equal to their respective given entry parameters.

Notice that not all the informal functional requirements can be expressed in semi-formal functional requirements of this form. Some of them can even be expressed as class or system invariants (see Section 2.1.1.3 for a description of class and system invariants) restricting system properties in small and larger scales.

5.2.2. Class Invariants

Some of the informal functional requirements are identified as being class invariants. They are those functional requirements that describe small limitations or boundaries, i.e., limitations of properties that eventually will restrict or describe a certain class. In a first stage of the formal specifications pseudo-phase we turn the identified class invariants into a semi-formalized form of the correspondent informal functional requirement. Later, these class invariants are to be ported into JML class invariants (see Section 5.4.2). For instance, from an informal functional requirement such as *"It must not be possible to overlap schedules in the same date and hour"* we can assume that it imposes a restriction on appointments, so it represents an invariant. In this case the informal functional requirement involves only appointments, so it becomes a class invariant. In this semi-formal phase, the invariants are written in a restrictive form involving natural expressions like "...it must..." or "...it mustn't...", but one can recur to logic forms for writing the semi-formal invariant such by using the expressions of "For all...", "Exists...", etc. As long as it restricts some global property of a class we can write it as a class invariant. So, the class invariant for the previously given informal functional requirement could be written like this:

> **FOR ALL** *objects **a1** AND **a2** of type appointment: IF **a1** NOT EQUAL **a2** THEN (date(**a1**) NOT EQUAL date(**a2**) AND hour(**a1**) NOT EQUAL hour(**a2**))*

Again, the above expression form is only a suggestion on how a semi-formalized class invariant could be written. Because of the ambiguous essence of natural language, the way people identify invariant properties from informal functional requirements is not a deterministic process. Hence, there is no universal rule that fully describes this process. Nonetheless, we give below some hints to identify invariants. Looking at the informal functional requirement example given, we identified as a class invariant because it describes there mustn't ever be appointments with the same date and hour, so this is obviously a limitation of the appointments properties. In the above expression the attributes *hour* and *date* of an appointment object are referred as *hour(***)* and *date(***)*.

5.2.3. System Invariants

Some of the other informal functional requirements are identified as being system invariants. They are those functional requirements that describe restrictions involving more than one distinguishable class, i.e., involving instance properties of more than one class. Also, as we carried out for the class invariants, in a first stage of the *formal specifications pseudo-phase* we turn the identified system invariants into a semi-formalized form of the respective informal functional requirement. Later, these system invariants are to be ported into JML class invariants (see Section 5.4.2). Considering an informal functional requirement like "*The prescription date of a medicine must be bigger than or equal to the date of the appointment in which the medicine was prescribed"*, one can clearly see, by analysing it, that the informal functional requirement is describes a restriction involving more than one class, i.e., appointments and medicines. The informal functional requirement is identified as a system invariant because it suggests that a global access to medicines and appointments in the card must exist, i.e., it involves two distinguishable classes. So, the system invariant for the previously given informal functional requirement could be written like this:

FOR ALL *object **m** of type medicine AND* **FOR ALL** *object **a** of type appointment: appointment(**m**) EQUALS **a** AND date(**m**) MORE OR EQUAL TO date(**a**)*

Once more, the above expression form is only a suggestion on how a semi-formalized system invariant could be written.

5.3. Design

In this section we describe the design phase, which follows the requirement analysis. At this phase, the use cases and the informal requirements from the requirement's phase are used as a base to design the structure model of the future application to be implemented. With the help from the previously defined use cases (including their textual specification) and informal functional requirements, we can have an idea of what modules and their respective functionalities (i.e. parts of the system and their responsibilities) that are needed to design the system application structure. First, a modularization of the requirements is made, i.e., by grouping informal requirements into specific parts of the system. The goal of grouping requirements is to be able to organize the system's structure so it can be more reusable and maintainable, and consequently making the JML specifications simple and reusable. For instances, taking the HealthCard development as an example (see Section **Erro! A origem da referência não foi encontrada.**) its structure is divided into Personal Data, Allergies, Vaccines, Appointments, Diagnostics, Treatments and Medicines. Each one of those modules have their respective responsibilities towards the management and storage of personal patient's information, patient's allergies information, patient's vaccines information, scheduled appointments and the respective diagnostics, treatments and medicines prescribed by a doctor. When implementing the system, those modules are basically the Java packages containing the respective Java interfaces and classes.

We suggest of making class diagrams to model the structure of the system. For each identified module, classes are designed and their methods are added to the model. The classes and their methods are written mainly based on the use cases. At the end, this class diagrams represent the structure of the system to be developed. So, by describing classes, interfaces and their method signatures, one can associate the semi-formal specifications to the structure model. The semi-formal specifications can be used to describe the behaviour of the methods and restrictions within the classes. Later, these semi-formal specifications are evolved into JML specifications and they will describe the behaviour of the implemented methods.

Basically, the relation between the class diagrams (or another model structure), obtained in this phase, with the formalization of the specifications is that here we can begin to associate the semi-formal specifications with the future classes and methods to be developed. The developers can have an idea where to write the JML specifications. For example, which semi-formal specifications will be associated with the method *addAppointment*(***) still written in the structure model. Figure 2 illustrates an example of associating semi-formal specifications with the structure model's methods. In case of a class invariant or system invariant, they are associated with the classes.

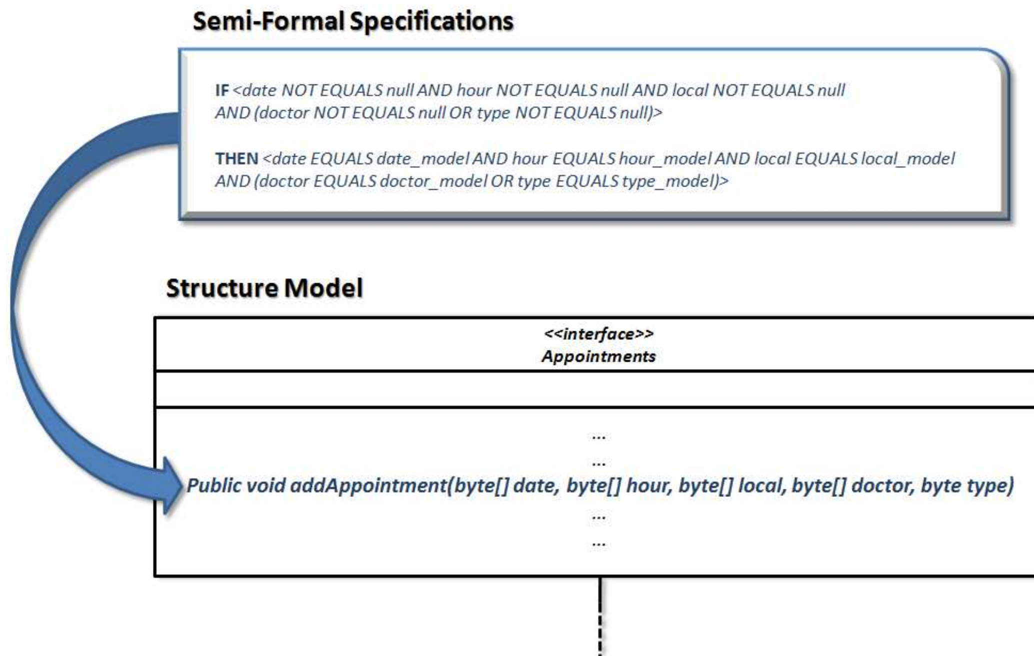

Figure 2. Semi-formal relations with the Structure Model

Important design recommendation: We strongly recommend designing the system by the interfaces. Designing the system by the interfaces means that for every class that probably will have specifications written, it is beneficial if the Java classes are implementing Java interfaces or abstract classes. For the sake of abstraction, the JML specifications are to be written in those Java interfaces, describing the methods that will be implemented in the respective concrete classes. Having a system designed by the interfaces we are making it reusable and maintainable and consequently the JML specifications will also be reusable and maintainable. That is, later we can implement the classes as we pleased as long as we respect the JML specifications written on the Java interfaces.

5.4. Implementation

In this section we describe the implementation phase, which follows the design. At this phase we implement the system structure with the support of the structure models previously defined and we complete this implementation through the support of JML specifications. From the semi-formal requirements and invariants attained in the first step of the *formal specifications pseudo-phase* we get JML specifications to specify how to implement the operations' procedures and their limitations. Through the class diagrams, the Java interfaces and its implementation skeleton classes[2] can be generated. Having those Java interfaces and classes still without implementations inside the methods, and taking a look to the method signatures at the interfaces, we associate with them each semi-formal requirement defined from the informal functional requirements (see example in Figure 2 - Section 5.3), and also each invariant is associated to the respective interface. Those JML specifications are written mostly at the remote Java interfaces and later one can develop the method implementations in the respective concrete classes in many ways as long as the specifications in the interfaces

[2] Skeleton classes are classes without implementation on its methods, only the signatures (i.e., method headers) and variable declaration exist. These classes are future implementation classes, i.e., they contain incomplete methods, without its procedures inside.

are respected. The process of implementing the system begins with the JML specifications writing. This is the final step of the *formal specifications pseudo-phase*, where semi-formal specifications are ported into formal specifications, i.e., the JML specifications. First we should write the JML abstract variables and next the JML class invariants and JML method functional specifications. Having the Java interfaces described with JML specifications we can start implementing the concrete classes according with the written specifications. Besides supporting a correct implementation of the system, the JML specifications also serve as the system documentation integrated with the code itself, a support for testing the code against the specifications and a support for employing a programming respecting the principles of B. Meyer's Design by Contract.

5.4.1. Writing JML Abstract Variables

The JML abstract variables, also known as model variables, are model specifications declared and used only at the JML specifications level. As it is recommended to write the JML specifications in Java interfaces of the classes due to reusability purpose, one cannot declare concrete attributes. In Java interfaces one can declare constant values but not non-static variables, so the use of abstract variables brings the advantage of representing concrete variables at the specification level. Having abstract variables at the specifications instead of concrete variables gives the developers the possibility of modifying those concrete variables without modifying the entire specifications of a class, and the many possibilities of implementing them as they want as long as the JML specifications aren't violated. For example, one could change a concrete variable's name but the specifications would stay correct if the abstract variable still represents that modified concrete variable, or one could even modify how an abstract variable represents concrete properties but still maintaining the old specifications. In a concrete class we can change the concrete variable being represented by using `represents` and still maintain without changes the specifications at the interfaces.

As seen in the requirement analysis phase, the abstract variables first originated from some domain concepts attained at the initial steps. They represent concrete data that will be managed by the classes. Also, when writing the semi-formal specifications, it is possible to identify the relevant abstract variables to be used in the JML specifications. Normally, these abstract variables are identified from data entities written in the semi-formal specifications, for instance, the date of an appointment, or the medicine's designation.

Declaration of Abstract Variables

The first step after making the skeleton Java classes and interfaces should be the abstract variable declaration. These abstract variables will be used in the various JML method functional specifications and class invariants. Abstract variables are declared in JML specifications by using the keyword `model` or `ghost` for ghost variables (which can't be represented and only exist in JML specifications) followed by the keyword `instance`. Abstract variables are declared in a similar way as concrete variables. Abstract variables can be declared as Java standard types, custom types or JML abstract data types. For example, an abstract variable of a Java standard type can be of the type *byte*, *short*, *int*, or any other Java type; an abstract variable of a custom type can be of the type *Appointment, Allergy* (both as example from the HealthCard) or another custom class object; and an abstract variable can be of a JML abstract data type like *JMLValueSequence*, *JMLObjectSequence*, or another type, from the JML's *org.jmlspecs.models package,* that represents a complex data structure (Iowa State University, 2002). For details about JML abstract data types, see Section 3.2.1. An abstract variable declaration is like this:

```
//@ public model instance short xpto_model;
```

This abstract variable xpto_model can represent another abstract variables, concrete variables, values or even expressions.

Linking Abstract Variables with Concrete Variables

The abstract variables can represent other abstract variables or concrete data related to a certain class or classes (except for *ghost* variables). These abstract variables are used to model class attributes or complex data structures in an abstract way, and only exist at the specifications level, being linked with real variables or expressions by using a mechanism of representation through the JML represents clause (see Section 3.2). A representation of abstract variables occurs in the concrete classes like this:

```
private /*@ spec_public @*/ short xpto; //@ in xpto_model;
/*@ public represents
  @   xpto_model <- age;
  @*/
```

Abstract variables are inherited, so they can be used in the concrete classes implementing the Java interfaces with the JML specifications written.

5.4.2. Writing JML Class Invariants

From the class and system invariants semi-formalized in the initial stages of the *formal specifications pseudo-phase,* we get the JML class invariants. Both class invariants and system invariants are to be ported into JML invariants. Apparently there's no difference between the two kinds of invariants when specifying them in JML, because we do it in the same way. However, system invariants are turned into JML invariants that involve instances from more than one distinguishable class and class invariants becomes JML invariants that don't involve instances for more than one class. By "one class" we assume, for example, that a certain Java interface ***X*** and its implementation class ***X1*** are one class, that is, basically we consider them as one class because ***X1*** inherits all the properties do ***X***.

From Semi-Formalized Class Invariants

Considering an *Appointment* class from the HealthCard, from a semi-formalized class invariant attained in the first stage of the *formal specifications pseudo-phase* (see Section 5.2.2) like this: - "**FOR ALL** *objects* ***a*** *of type appointment: date_model NOT EQUALS null AND hour_model NOT EQUALS null AND local_model NOT EQUALS null"* - we can write the following JML class invariant in the *Appointment* Java interface:

```
/*@ public invariant date_model != null
  @                    && hour_model != null
  @                    && local_model!= null;
  @*/
```

Where in any state of an object *Appointment*, its attributes of date, hour and local must never have the value null. This JML invariant is written in the *Appointment* interface and all methods and constructor must respect all it visible state. Each time an *Appointment* object instance is created, it is required to declare and instantiate the concrete variables represented by the abstract variables written in the invariant (i.e., date_model ← date, hour_model ← hout and local_model ← local).

From Semi-Formalized System Invariants

Being a system invariant an invariant that involves two distinguished classes, then JML specification written for a system invariant must be written in a class that makes reference do

those two classes. For example if we have two distinguishable classes, ***X*** and ***Y***, and we have a system invariant making reference to ***X*** and ***Y***, then the invariant formalization must be written in a class ***XY*** that makes reference to ***X*** and ***Y***. The class ***XY*** has a global access to ***X*** and ***Y*** so it makes sense having the invariant written there to restrict properties in which ***X*** and ***Y*** are involved.

Let's consider as example the class ***X*** as *Appointment* class, the class ***Y*** as *Medicine* class and the class ***XY*** as *CardServices* class, all from the HealthCard (see Section **Erro! A origem da referência não foi encontrada.** for a description on the structure of the HealthCard). From a system invariant attained in the first stage of the *formal specifications pseudo-phase* (see Section 5.2.3) like this: - "**FOR ALL** *object **m** of type medicine AND* **FOR ALL** *object **a** of type appointment: appointment(**m**) EQUALS **a** AND date(**m**) MORE OR EQUAL TO date(**a**)"* – Where an appointment has a date and a certain medicine is prescribed in an appointment, then that medicine has a date equal of the respective appointment's date (when it was prescribed) or the medicine has its prescription renovated at later date. The following JML invariant specifies this property that must be preserved:

```
/*@ public invariant
  @    (\forall int i; i < ((Medicine[])medicines.getData()).length
  @                    && i >= 0;
  @           (\forall int k; k < appointments.getData().length
 @                          && k >=0;
  @               ((Medicine[])medicines.getData())[i].getAppointmentID()
  @                       != appointments.getData()[k].getID()
  @               ||
  @               ((Medicine[])medicines.getData())[i].date_model
  @                       >= appointments.getData()[k].date_model
  @           )
  @    );
  @*/
```

5.4.3. Writing JML Method Functional Specifications

The JML method functional specifications are the specifications that describe the method's behaviour. They can describe a method's normal behaviour, their preconditions, postconditions and even its exceptional behaviour. We write these specifications from the semi-formal functional requirements attained in the first step of the *formal specifications pseudo-phase* (see Section 5.2.1). Later, when coding the empty methods, one has to respect these specifications as they describe the conditions under which the methods will correctly function. To start writing the JML method functional specifications we begin by looking at the semi-formal functional requirements. For example, let's consider the example of class Appointment of the HealthCard. From the following semi-formal functional requirement for the method *addAppointment*:

> "**IF** *<date NOT EQUALS null AND hour NOT EQUALS null AND local NOT EQUALS null AND (doctor NOT EQUALS null OR type NOT EQUALS null)>*
>
> **THEN** *<date EQUALS date_model AND hour EQUALS hour_model AND local EQUALS local_model AND (doctor EQUALS doctor_model OR type EQUALS type_model)>"*

We can generate the following JML specification:

```
/*@ public normal_behavior
  @      requires date != null && hour != null && local != null
  @                 && (doctor != null || type != 0);
```

```
  @      assignable appointments_model;
  @      ensures (\forall int i; 0 <= i && i < date.length(); date[i] == date_model[i])
  @                 && (\forall int i; 0 <= i && i < hour.length(); hour[i] == hour_model[i])
  @                 && (\forall int i; 0 <= i && i < local.length(); local[i] == local_model[i])
  @                 &&
  @                 ((\forall int i; 0 <= i && i < local.length(); local[i] == local_model[i])
  @                 ||
  @                 type == type_model);
  @ also
  @ public exceptional_behavior
  @      requires date == null || hour == null || local == null
  @                 || (doctor == null  && type == 0);
  @      signals_only UserException;
  @      signals_redundantly (UserException e)
  @                 appointments_model.equals(\old(appointments_model));
  @*/
public void addAppointment (byte[] date, byte[] hour, byte[] local, byte[] doctor, byte type)
throws RemoteException, UserException;
```

Where we can map the semi-formal functional specifications into formal specifications like: **IF ⇔ requires** and **THEN ⇔ ensures**. In our strategy, the semi-formal IF statement is similar to the requires statement from JML, and the THEN is similar to the ensures from JML. The implementation of *addAppointment* is made in the concrete class implementing *Appointments*. In the specifications, the normal behaviour describes the preconditions and postconditions. These conditions are written by using the JML keywords of `requires` and `ensures`. The first part of the previously presented semi-formal functional requirement is mapped into the `requires` block, and the second part is mapped into the `ensures` block. Under the exceptional behaviour, we state the conditions of an exceptional execution of the method.

5.4.4. Coding the applications

At this step we already have Java interfaces with JML specifications written within them asserting invariants, methods and attributes, and incomplete concrete Java classes (i.e., only with method skeletons). In this step, we begin to code the procedures of the empty methods from the concrete classes which implements the JML specified Java interfaces.

We can implement the concrete classes and their methods in various ways, as long as the specifications are respected. Let's not forget that the JML specifications are written from an evolutive process that comes directly from the informal requirements. Another purpose of formally specifying the Java code with JML, it's the documentation. Besides serving as a mean of correctly implement the code and for supporting its verification and validation against the specifications, it can be used, at the same time as a way of documenting the application Java code. The JML specifications can be used to document the code like *JavaDocs*, however we can't use *JavaDocs* to test the specifications against the code.

5.5. Validation and Verification

While implementing, it is possible to validate and verify the code against the JML specifications. There are tools for supporting validation and verification and the most basic is

the JML Common Tools suite [2]. This suite provides support to the run-time and static assertion checking of JML specifications (see Section 3.3). Checking an application with this suite is an iterative process of checking the implementation with respect to the JML specifications, and then evolving either the specification or the implementation (or both) when a run-time error is produced. Errors can be detected before a concrete implementation for the application is written. For instance, Java interfaces and Java abstract classes are checked against JML specifications, obtained from the formal specification pseudo-phase, before writing full implementation for those interfaces and abstract classes. At this point, programmers can go back to an earlier development phase, e.g., modifying some informal functional requirements; thereafter JML specifications are evolved accordingly.

6. A Running Example

6.1. The HealthCard Application

In the following, we describe the application we used to validate our software development strategy. The application is named HealthCard. It is a smart card application for managing medical appointments. The application has been fully implemented by Ricardo Rodrigues, following the software development strategy introduced by us, as part of his master thesis work (Rodrigues, 2009).

HealthCard stores people's medical information. It is named HealthCard because it runs on a smart card, a pocket-sized plastic card with embedded integrated circuits that process data (see Section 6.2 for further information about smart cards). A typical smart card application includes on-card applets (the applets running on the card), a card reader-side, and off-card applications (e.g., a computer program communicating with the card applets). HealthCard is written in Java Card, a subset of Java used to program card applets. We used the Java Card Method Invocation (JCRMI) model for communication between off-card applications and on-card applets. This model implements a client-server setting with the HealthCard acting as server, and off-card applications as clients, communicating via APDU (Application Protocol Data Unit) messages. Figure 3 shows the structure of the HealthCard smart card.

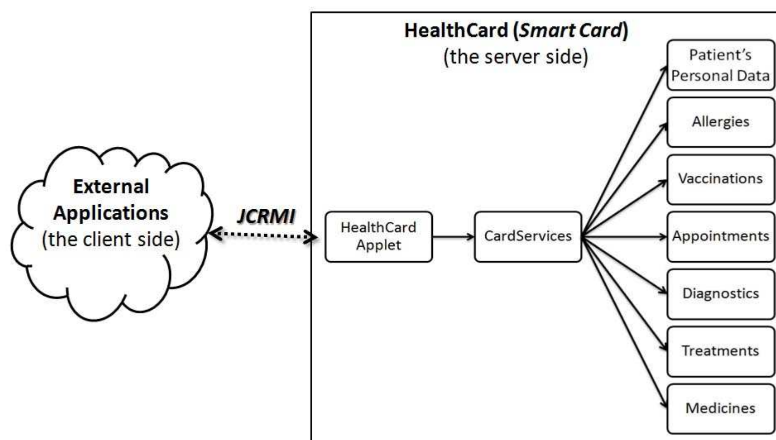

Figure 3. HealthCard application structure

A patient can use his HealthCard to furnish accurate medical information to general practitioners in medical centres with the appropriate system to read it. The HealthCard

manages the patient's personal details, his historical record of allergies, vaccines, diagnostics, treatments and prescribed medicines. The HealthCard is divided in several modules for managing the medical information. Each module has a remote interface, and an implementation class that serves the appropriate services. All the remote interfaces are referenced in a single remote interface named *CardServices* whereby an external client can invoke services. For example, if an external client calls the method *getApp*() in *CardServices*, he gets a reference to the *Appointments* remote interface. This reference can then be used to invoke appropriate methods implementing services.

6.1.1. HealthCard Formal Development

HealthCard addresses the problem of providing accurate and concise medical information to medical centres and general practitioners. The use of formal methods in the HealthCard development process is due to the application's domain nature, that is, due to the fact that the medical domain involves people's healthcare, people's lives and overall, medical information trustiness. Through the use of formal methods we can achieve a correct smart card application, and that means that the application will work as specified, that is, its implementation and execution must respect its specifications and it must function as it is really intended to function (i.e., must be a reliable system). When developing a software application for sensible domains, such as medical, one must develop it correctly (see Section 2.1 for a description about software correctness). Also, besides using JML to specify functionalities properties, we can use JML to address the security and privacy problems related with this kind of medical software application. JML can be used to formally describe security and privacy properties, however supporting a correct security implementation doesn't mean that the system will be secure. Correctness does not necessarily imply security. When addressing these kinds of problems with JML specifications, it is still a challenge if we have to deal with all the low-level details of Java. That is, some program wide security properties such as authentication, confidentiality or integrity are far harder to express in JML. (Warnier, 2006)

This medical software application is to be held in smart cards. Therefore, a patient can carry his medical information on a card and use it when going to any medical centre with the appropriate system to read it. A typical smart card includes in-card applications, i.e., the applets running on the card. For implementing the in-card applications we use the Java Card language (see Section 6.2 for further information on Java Card). This language is a precise subset of the Java language used to program applets for devices such smart cards. In Java Card, smart cards provide two models for the communication between a host application and a Java Card applet. (Ortiz, 2003) The first model is the fundamental message-passing APDU model, which basically relies on the trade of messages in the APDU format between the in-card applets and the off-card applications. The second model is based on the Java Card Remote Method Invocation (JCRMI), in which a Java Card applet is the server and makes accessible functions to external client applications. The smart card technology provides patients with:

1.) A way to digitalize their information.

2.) A mechanism to convey their information to others.

3.) A security mechanism so that their information is not disclosed to non-authorized parts.

Carrying a card with relevant medical information easies the way a patient can tell his health problems to medical professionals. In this way, the card acts as a patient data server. In

our solution, smart cards are used to carry people medical information. It encompasses personal data such as name, age, gender and blood type, as well as medical history about allergies, vaccinations, previous health problems and treatment plans. Figure 4 shows how medical information is organized within a smart card. Notice that the figure conveys in the necessary patient's information contained in the card rather than the structural description of the HealthCard. The information stored can be divided into the patient's personal data, the scheduled appointments and his medical history. Information about the patient's medical history includes allergies, vaccination, health problems and treatment plans associated with health problems. The treatment plans are associated with diagnostics, prescriptions and medical recommendations.

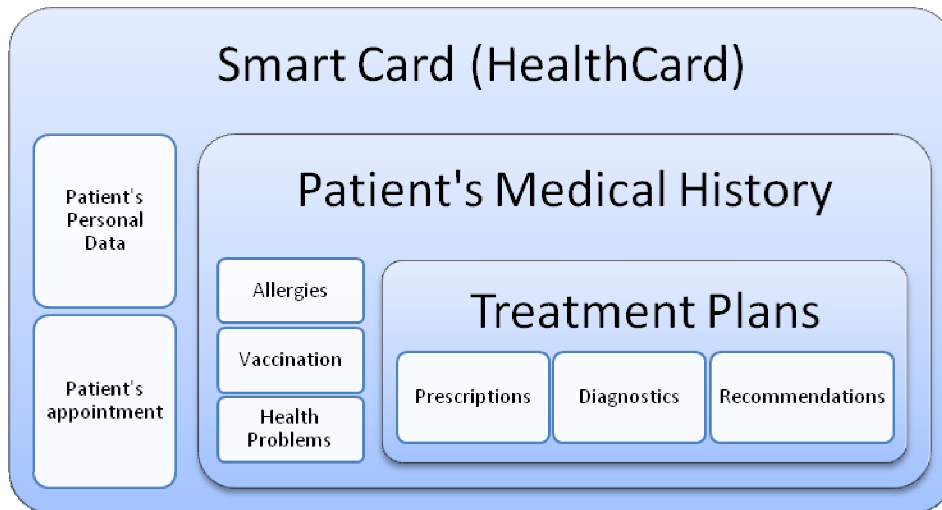

Figure 4. Proposed information held on a smart card for medical appointment management

For managing the data held on the card we need at least an in-card applet that provides functions to manage it. Since we'll use smart cards, we propose the use of *Java Card* for programming those in-card applets (i.e., the health card application). Java Card is a programming language that has in consideration the memory resource limitations of smart cards (Ortiz, 2003) (see Section 6.2). We propose the use of *Java Modelling Language (JML)*[3] for formally specify the health card application's informal requirements. These JML formal specifications are used to support the correct implementation of our application. Also, we propose the use of JML-based tools to check for correctness of the implementation.

6.1.2. HealthCard System Architecture

The architecture of the HealthCard system is illustrated in Figure 5. A patient can use his smart card in any medical centre that has our system implemented.

[3] JML is a formal behavioral interface specification language for Java which includes the essential notations used in Design by Contract as a subset. Leavens, G. T., & Cheon, Y. (2006). *Design by Contract with JML.* Iowa State University; University of Texas at El Paso, Dept. of Computer Science.

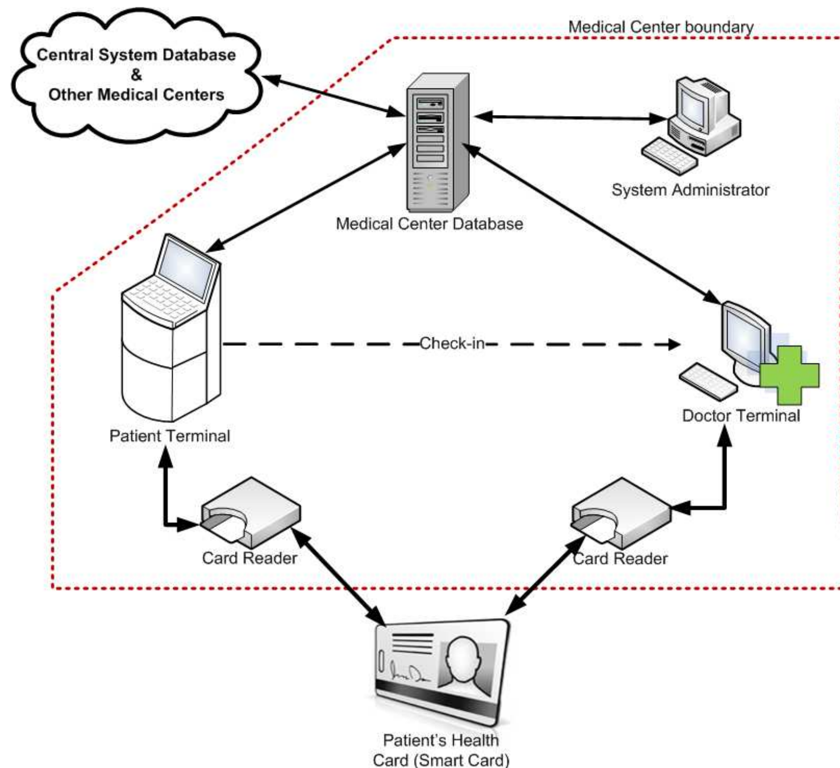

Figure 5. HealthCard System

Besides including the HealthCard application supported in smart cards, the HealthCard system architecture idea consists in more components. However, Ricardo Rodrigues developed the card application and a prototype of an external client only. The system architecture in Figure 5 consists of at least two card terminals. One is the patient terminal, which includes an attached smart card reader. This terminal may be used for appointment scheduling, appointment check-ins, visualization and modification of some in-card personal data, and for requesting medical prescriptions renewals. The second terminal is the doctor's terminal, which also can include a smart card reader. The doctor may insert medical information into the patient's card by using this terminal. Beyond those two terminals our architecture includes a Medical Centre database. This database provides support to the on-the-card patient's information, by storing all known allergies, medicines and vaccines, and other medical standard designations. In this way, the card will only need to keep references to those items rather than the whole designation (i.e. the names of allergies, vaccines, medicines, etc.). Also, that database will provide support to the information about doctor's available schedules and other medical centre information. This medical centre database may be linked to other medical centres and one of them may be the central system database. This central system would update medical information in all medical centres databases. Finally, there's a system administrator that has the responsibility for operating and keeping the medical centre database updated.

System Components:

HealthCard (*smart card*) contains personal and medical information about the card owner (patient) and his scheduled appointments, i.e., contains the HealthCard application that was developed to validate the proposed strategy.

Card reader will serve as terminals for reading/writing the smart cards and linking points to client machines (Patient Terminal and Doctor's computer).

Patient Terminal for appointment scheduling, checking-in and some other basic card operations made by the patient.

Doctor will have a terminal for accessing patient medical data contained on the card.

Medical Centre Database will contain doctor's schedules, medical centre information, patient appointments, and lists of known allergies, health problems and vaccines.

System Administrator will be responsible for maintaining the medical centre database.

Central System Database will update all the medical centres systems.

6.2. Smart cards and Java Card

The HealthCard application involves the technologies of smart cards and Java Card. The HealthCard is implemented in Java Card and it is to be supported in smart cards. A smart card is a plastic card that contains an embedded integrated circuit (IC) and basically resembles a credit card. Most smart cards have both microprocessors and memory, for secure processing and storage. Smart cards are highly secure by design, and tampering with one results in the destruction of the information it contains. (Ortiz, 2003) Usually, a smart card has about 1Kb of RAM and 16Kb of EEPROM, which contains persistent data, including the compiled program code. Smart cards don't contain a battery, and become active only when connected with a card reader. When connected, after performing a reset sequence the card remains passive, waiting to receive a command request from a client (host) application. (Ortiz, 2003) Java Card is a programming language for programming smart cards. Java Card is a subset of the Java programming language specially designed having in mind the memory resource limitations of smart cards. (Ortiz, 2003) ISO 7816 is the international standard for smart cards that use electrical contacts on the card. (Cardlogix Corporation, 2009)

6.2.1. Elements of a Java Card Application

A smart card system is composed by a **card-side** (the applets running on the card), a **card reader-side**, and **back-end** elements (a computer communicating with the card applets). (Ortiz, 2003) In the following Figure 6 we can see an illustration of this composition.

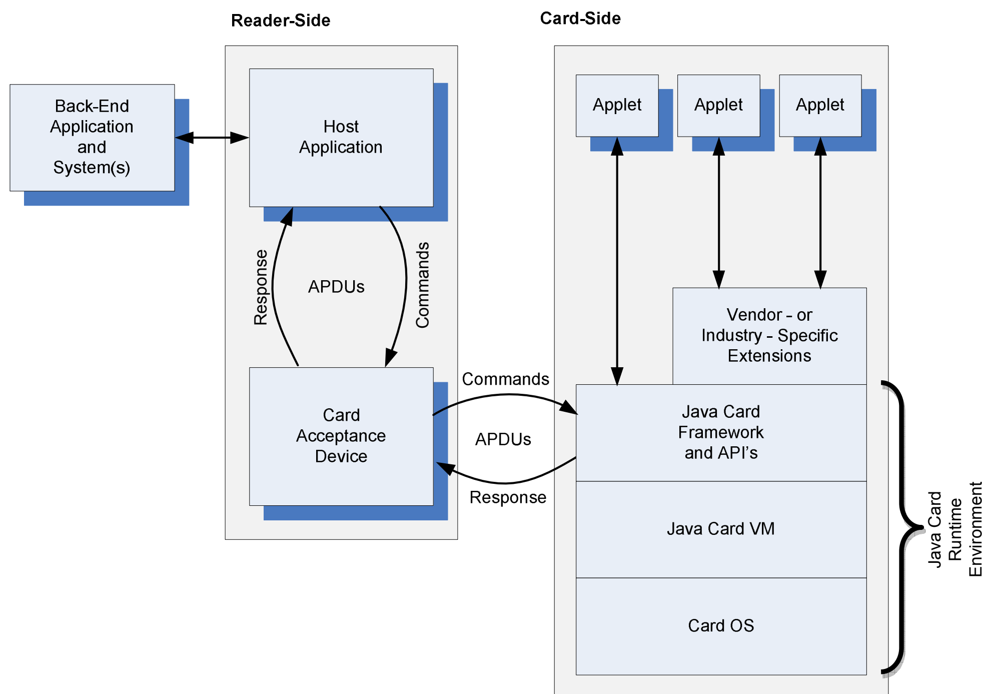

Figure 6. Architecture of a Java Card Application (Ortiz, 2003)

6.2.1.1. *Back-End Application and Systems*

Back-end applications are elements of the system that provide services that support in-card Java applets. For example, a back-end application could provide a connection to security systems that together, with credentials from the card, could result in a better security. In a credit card payment system, the back-end application could provide payment information and access to the credit-card.

6.2.1.2. *Reader-Side Host Application*

Reader-Side terminals can be a PC or an electronic payment terminal, a cell phone, or a security subsystem. In them reside host applications that can handle communication between the user, the Java Card applet, and the provider's back-end application.

6.2.1.3. *Reader-Side Card Acceptance Device*

The *Card Acceptance Device* (CAD) is a card reader. It's the gateway of communication between the host application and the Java Card device, and besides serving as a way of communication, a CAD provides power to the card. A CAD may be attached to a desktop computer using a serial port, or it may be integrated into a terminal such as an electronic payment terminal (ex., at a restaurant or a gas station).

6.2.1.4. *Card-Side Applets and Environment*

In Java Card, an in-card application is an applet. A Java Card can have one or more applets residing the card, along with supporting software. The supporting software consists in the card's operating system and the Java Card Runtime Environment (JCRE). The latter one includes the Java Card VM, the Java Card Framework and API's, and some extension APIs.

All Java Card applets extend the Applet base class and must implement the `install()` and `process()` methods. Later, when installing the applet, JCRE calls `install()`. And every time there is an incoming APDU message for the applet, JCRE calls `process()`.

When loaded, Java Card applets are instantiated, and stay alive when the power is switched off. A card applet acts like as a server and is passive. Once a card is powered up, each applet remains inactive until it's selected. The applet is active only when an APDU has been dispatched to it.

6.2.2. Accessing the Smart Card (Communication in Java Card)

According to ISO 7816-5 standard, each smart card application must have an application identifier (AID). (Cardlogix Corporation, 2009) These AIDs are sequence of bytes between 5 and 16 bytes in length, and in Java Card technology they are used to identify Java Card applets as well as packages of Java Card applets. When inserted a smart card into a card acceptance device, the running external application sends a command to the card containing the AID of the applet to perform the required operation. The AID is crucial for allowing the external applications accessing Java Card applications in smart cards. (Ort, 2001)

For accessing smart cards there are two models for the communication between a host application and a Java Card applet. The first model is the fundamental message-passing APDU model, and the second is based on *Java Card Remote Method Invocation* (JCRMI), a subset of the J2SE RMI distributed-object model.

A logical data packet is exchanged between the CAD (Card Acceptance Device) and the Java Card Framework, which is called APDU (Application Protocol Data Unit). An APDU is sent by the CAD, received and then forwarded to the appropriate applet that processes the APDU command and returns a response APDU. (Ortiz, 2003)

A command APDU has a required header and an optional body, containing:

- CLA (1 byte): This required field identifies an application-specific class of instructions.
- `INS` (1 byte): This required field indicates a specific instruction within the instruction class identified by the `CLA` field.
- P1 and P2 (1 byte each) are required fields used to pass command specific parameters for the qualification of INS, or input data.
- Lc (1 byte): This optional field is the number of bytes in the data field of the command (length command).
- Data field (with length given by Lc): This optional field holds the command data.
- Le (1 byte): This optional field specifies the maximum number of bytes in the data field of the expected response (length expected).

Table 3. A command APDU format (Ortiz, 2003)

Command APDU						
Header (required)				Body (optional)		
CLA	INS	P1	P2	Lc	Data Field	Le

A response APDU has a format much simpler:

- Data field (with a length determined by Le in the command APDU): This optional field contains the data returned by the applet.
- SW1 (1 byte) and SW2 (1 byte) are required status words. They contain the status information as defined in ISO 7816-4. (Cardlogix Corporation, 2009) In case of successful execution, they contain 0x9000.

Table 4. A response APDU format (Ortiz, 2003)

Response APDU		
Body (optional)	Trailer (required)	
Data Field	SW1	SW2

The Java Card implementation of the HealthCard application is based on *JCRMI (Java Card Remote Method Invocation)*. It adds an additional abstraction layer above the message-passing model, avoiding low-level communication through APDU's (Warnier & Oostdijk, Java Card Remote Method Invocation) therefore simplifying the code written and saving memory space in the card. Simplifying the code makes it easier to specify the implementation, which leads to more concise and reliable code.

6.2.3. Java Card Remote Method Invocation (JCRMI)

In the message-passing model for communication between the host application and the Java Card applets we had to program explicitly low-level byte sequences of APDU messages, but with the *Java Card Remote Method Invocation (JCRMI)* framework we don't need to program like that anymore. The JCRMI is similar to Java Remote Method Invocation (JRMI) applied in Java applications. The JCRMI makes it possible to directly call methods from the Java Card smart card. (Oostdijk & Warnier) Basically, JCRMI adds a middleware layer that translates calls to the methods of an applet to ADPU messages. On the card, APDU messages are translated back to methods of the remote object. These processes are called *marshalling* and *unmarshaling*. (Oostdijk & Warnier) These remote objects residing on the card are created on the moment of the applet installation. A client can get a reference to those remote objects. When a client calls a method on the remote object, the method that the client calls on is actually a *stub object* that resides on the client side. This stub translates the method call to an APDU command message and sends it to the card. On the Java Card side this APDU is passed on to a *skeleton object* that translates the message back to a method call. (Oostdijk & Warnier)

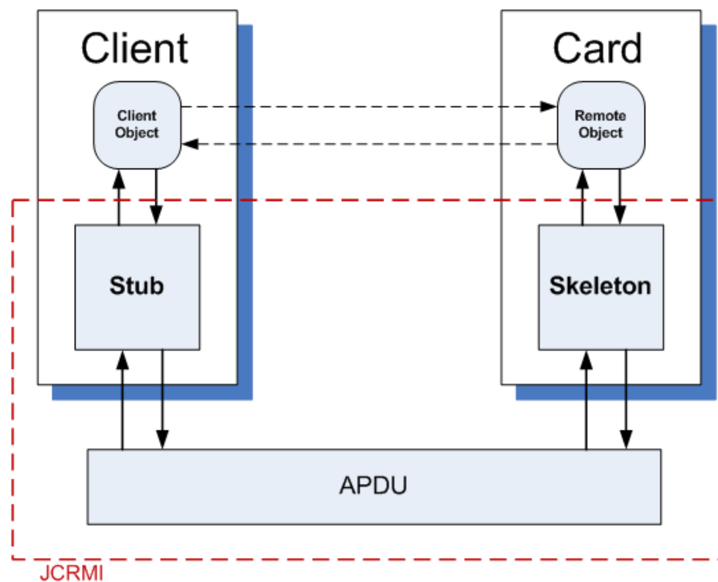

Figure 7. Java Card Remote Method Invocation architecture (Oostdijk & Warnier)

The method call is invoked and the return value is translated to an APDU response message by the *skeleton object*, which then sends it to the client. On the client side the APDU message passes through the stub, which translates it back to a return value.

A JCRMI applet consists of at least one interface and two classes: - a *remote interface*; the *implementation of that interface*, and the *applet class*.

- The *remote interface* extends *java.rmi.Remote* interface and defines what methods can be called with JCRMI. This interface must also be presented on the client side.

- The *implementation of the remote interface* is the implementation itself. It can be used to generate a stub class for the client.

- The *applet class* extends *javacard.framework.Applet* and contains the inherited *install()*, *select()* and *process()* methods. This class act as the entry point for all method calls and directs these to the actual implementations. (Oostdijk & Warnier)

When developing a JCRMI applet we should start implementing the remote interface. From that interface we write its implementation and the client class, the class that will call remote object methods. Next, we compile the code so that we have their class files. In the compilation, the interface will originate a stub, which will provide, to the client, a way to interact with the remote object. The stub and the client class stays at the client side. The applet and remote implementation classes are converted into a cap file and inserted in a smart card. The Figure 1 illustrates this whole process.

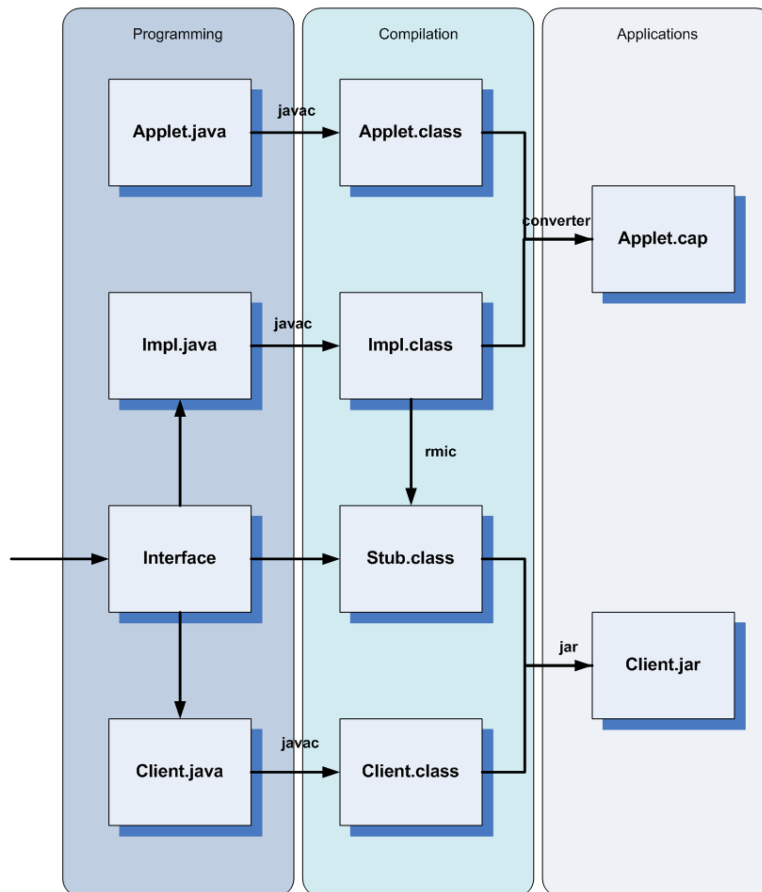

Figure 8. JCRMI applet implementation process (Oostdijk & Warnier)

6.3. JML-based Formal Development of the HealthCard

In the following we describe the employment of our strategy in the development of part of the HealthCard.

6.3.1. Getting the Informal Requirements

During the analysis phase, requirements are described using use cases and functional requirements. The use cases model the purpose and functionality of the application to develop. They are later used to determine what classes, methods and structure will be modelled at the design phase. The informal functional requirements define, in an informal way, the inputs, the behaviour, the outputs, and the restrictions of the system to develop. In the following, we present a small example from the HealthCard system that shows how informal functional requirements are evolved into the three semi-formal requirements documents described in Section 5.2. We present below a use cases example in Figure 9 and some of the informal requirements of the HealthCard application:

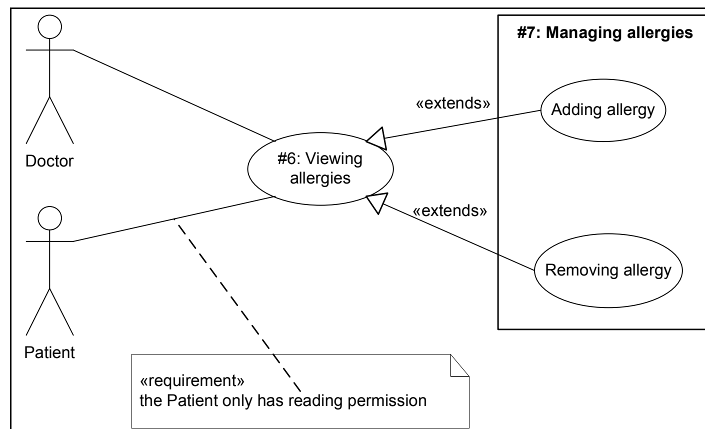

Figure 9. Use Cases of the patient's allergies' information management example

- **IFR1** There must not exist duplicated entries for allergies with the same designation code.
- **IFR2** A fixed number of allergies can be introduced in the card only.
- **IFR3** All allergy designation codes must have a stipulated length.
- **IFR4** The prescription date of a medicine must be bigger than or equal to the date of the appointment in which the medicine was prescribed.

In the following, we show the three semi-formal documents obtained from the informal functional requirements above.

6.3.2. Getting the Semi-Formal Functional Requirements

Some of the informal functional requirements are evolved into a more mathematical form. Yet expressed in natural language, this new form brings requirements closer to JML method specifications. However, the process of evolving informal functional requirements into this new form is not linear, and it requires some expertise and ingenuity. The general form of a semi-formal functional requirement is `if <event/condition> then <restriction/rule>`, in which the guard is an event or a condition that triggers a rule that restricts (changes) the current state of the system. This rule can be regarded as the body of a method in a class, and the condition as the pre-condition under which this rule may be triggered. This new form is closer to JML specification, and the principles advocated by the design-by-contract. Notice that not all the informal functional requirements can be expressed in this form. Some of them can even be expressed as class or system invariants (see Section 6.3.3). As an example of how this semi-formal form is attained, the informal functional requirement **IFR1** is transformed into *if <a new allergy is to be added to the list of referenced allergies, and the allergy designation has already been referenced>, then <the new allergy is not inserted>*. We show below the semi-formal requirements obtained from the first two informal requirements above:

- **SFR1** ***From*** **IFR1**. If a new allergy is to be added to the list of referenced allergies, and the allergy designation has already been referenced, then the new allergy is not inserted.
- **SFR2** ***From*** **IFR2**. If an allergy is to be added to the list of referenced allergies, and the limit of the number of referenced allergies has already been attained, then the state of the card remains unchanged.

6.3.3. Getting the Class and System Invariants

Some of the informal functional requirements are identified as class invariants. They are written from those functional requirements that describe limitations or boundaries in small-scale, i.e., limitations of properties that eventually will restrict or describe a certain object class only. For example, the informal functional requirement **IFR3**: "All allergy designation codes must have a stipulated length", restricts the length of designation code that instances of the class `Allergy` manipulate. To write this class invariant (see **CI1** below), we use a variable *des* to represent the designation code of an allergy. Later on, this variable can be modelled as a JML abstract variable (see Section 6.3.4).

- **CI1** size(*des*) equals to CODE_LENGTH

Because of the ambiguous essence of natural language, the way people identifies invariant properties from informal functional requirements is not a deterministic process. Hence, there is no universal rule that fully describes this process. Nonetheless, we give below some hints to help people identify invariants. **CI1** describes a property on a reference code and its length of an allergy, so that eventually these two will be fields of some class Allergy. The reference code will be an instance variable of this class, initialised in its constructor, and the length will be a static field as it needs to be the same for any instance class. Some other informal functional requirements are identified as system invariants. Unlike class invariants, system invariants describe invariant properties relating objects of distinguished classes. For instance, **IFR4** (see Section 6.3.1) describes a property of objects of classes Appointment, managing information about appointments scheduling, and Medicine, managing information on prescribed medicines in appointments, must satisfy together. **IFR4** becomes the semi-formal system invariant requirement **SI1** below:

- **SI1** For all object m of type medicine, and all object a of type appointment such that if appointment(m) equals to a, then date(m) is bigger than or equal to date(a).

6.3.4. Design and Implementation

During the design phase, the structure of the application is created from the requirements. This structure encompasses class diagrams for interfaces, abstract classes, and concrete classes. In parallel with the design phase, during the formal specification pseudo-phase, semi-formal functional requirements, and class and system invariants are written (Sections 6.3.2 and 6.3.3). Semi-formal specifications are later ported to JML specifications (Section 6.3.5). During the implementation phase, from the structure of the application generated in the design phase, Java abstract classes, Java interfaces and Java classes are written. In a first stage, the implementation only contains code skeletons, so no method in any concrete class is implemented. JML specs are embedded within the code. Hence, the JML Common tools can be used to check the code during early stages of the implementation (i.e., before fully implementing concrete Java classes). Therefore, the Java code can be evolved so as to conform to the JML specifications, or the specifications can be evolved to conform to an expected behaviour. Checking that one conforms to the other is done automatically with the JML Common Tools. JML eliminates programmers' responsibility of keeping track of how properties a program must respect are affected by changes in the code. To have a high level of abstraction in specifications, JML provides support for abstract variables, which exist at the level of the specification, but not in the implementation. Declarations of abstract variables are preceded by the JML keyword *model*, and are related to Java code by a *represents* clause. This clause specifies how the value of an abstract variable is calculated from the values of concrete variables (see Section 6.3.5). Abstract variables are useful in describing properties about

interfaces because these are not allowed to declare (concrete) variables in Java. Within an interface, an abstract variable describes the state of the implementing classes. Abstract variable specifications for interfaces and for abstract classes do not need to be written down again in implementing classes or sub-classes, since JML specifications are inherited. This ensures behavioural sub-typing. That is, a sub-class object can always be used where a super-class object is expected. The reader is invited to see Sections 5.3 and 5.4 for respectively details about the design and implementation phases.

6.3.5. JML Formal Specification Pseudo-Phase

Semi-formal functional requirements **SFR1** and **SFR2** (from Section 6.3.2) relate to method *addAllergy* in interface *Allergies* (see below in Code 9). In Java, interfaces cannot declare attributes, hence, *Allergies* declares an abstract JML variable *as*, modelling stored referenced allergies. The JML *JMLEqualsSequence* type models a sequence of objects that can be compared using the standard method equals (see Section 3.2.1 for a description about JML abstract data types). We declare two additional abstract variables, *size* and *maxsize*, modelling the number of referenced allergies, and the maximum number of referenced allergies. A normal behaviour specification expresses that if all the pre-conditions hold (clauses *requires*) in the pre-state of the method, it will terminate in a state in which all the postconditions (clauses *ensures*) hold. The semi-formal functional requirement **SFR2** is expressed as the JML pre-condition *size < maxsize*, while the **SFR1** appears in two separated normal postconditions that make use of the abstract method *existsAllergy* (not shown here) for checking whether the designation of an allergy has already been stored in as or not. Therefore, if the designation has already been stored, the list of allergies remains unchanged, *as.equals(\old(as))*, otherwise the allergy designation is stored at the end of the list:

- *as.equals(\old(as).insertBack(desigRepr(designation)))*

JML abstract method *desigRepr* (not shown here) maps an array of bytes to a unique value.

```
//@ model instance JMLEqualsSequence as;
//@ model instance short size;
//@ model instance short maxsize;
/*@ public normal_behavior
  @ requires size < maxsize;
  @ requires designation != null && date != null;
  @ requires existsAllergy(designation);
  @ assignable as, size;
  @ ensures as.equals(\old(as));
  @ also
  @ public normal_behavior
  @ requires size < maxsize;
  @ requires designation != null && date != null;
  @ requires !existsAllergy(designation);
  @ assignable \nothing;
  @ ensures as.equals(\old(as).insertBack(
  @ desigRepr(designation)));
  @*/
public abstract void addAllergy ( byte[] designation,byte[] date)
throws RemoteException, UserException;
```

Code 9. Specified addAllergy method from Allergies interface

Abstract specifications are related to actual Java code through the use of a JML *represents* clause. The following Code 10 exemplifies this relation between abstract specifications and concrete variables. In the presented code below, as, declared in *Allergies* shown in Code 9, is related to code in the *Allergies_Impl*, which implements the interface *Allergies*. The abstract variable size is represented as the concrete field *nextFree*, and *maxsize* as the static variable *MAX_ITEMS*. The pure method *allergiesRepr* represents *as* as a *JMLEqualsSequence* produced

by the insertion of all the elements in allergies. In JML, `pure` methods are side-effect free methods.

```
//@ represents size <- nextFree;
//@ represents maxsize <- MAX_ITEMS;
//@ represents as <- allergiesRepr();
/*@ pure model JMLEqualsSequence allergiesRepr() {
  @ JMLEqualsSequence r = new JMLEqualsSequence();
  @ for (short i=0; i < nextFree; i++) {
  @ r = r.insertBack((Object)(allergies[i]));
  @ }
  @ return r;
  @ }
  @*/
```

Code 10. Relating abstract specifications with actual Java code in *Allergies_Impl*

JML Class and System Invariants.

The class invariant **CI1** is expressed as the JML invariant below. This invariant is declared in class *Allergy*.

//@ instance invariant des.size == CODE_LENGTH;

The system invariant **SI1** is expressed as the JML invariant below in Code 11. This invariant suggests that a global access to medicines and appointments in the card must exist. Following the Java Card Remote Method invocation (JCRMI) approach for communication, in which the Java Card applet is the server, the HealthCard application defines an interface *CardServices* that declares all the services available for remote objects. Class *CardServices_Imp*, an implementation of this interface in Java, accesses the information and the state of any remote object in the card. *CardServices_Imp* declares two variables *med* and *app* for keeping track of medicines and medical appointments respectively. Method getData() returns an array of objects of type *Medicine*. Method *getApp()* returns an array of objects of type *Appointment*.

```
/*@ invariant
  @ (\forall int i; i<med.getData().length & i>=0;
  @   (\forall int k; k<app.getApp().length & k>=0;
  @     med.getData()[i].getAppID() == app.getApp()[k].getID()
  @     ==>
  @     med.getData()[i].getDate() >= app.getApp()[k].getDate() ))
  @*/
```

Code 11. A system invariant as JML invariant

7. Conclusion

The use of formal methods reduces the chances for requirements errors as it forces a detailed analysis of those requirements, and also helps to detect and resolve their incompleteness and inconsistencies while developing a software system. The use of formal methods in software development has the purpose of producing correct software programs. According to Sommerville (Sommerville, 2000), when a conventional software development process (i.e., without using formal methods) is used, validation costs are more than 50% of the whole development costs, and implementation and design costs are the double of the specification cost. However, formal methods are not widely used as software development techniques in software industry. Some of the main reasons are: the lack of methodologies and tools to support the use of formal methods; the inefficient use of formal specifications as an

appropriate tool for communicating with the end user; and the inefficient support of developers' creative side while employing formal methods in their development processes. Overall, software development managers feel reluctant to use formal methods techniques because their benefits are not yet well-known. The recompense of using formal methods is not immediate and it is hard to quantify. The reader is invited to see Section 2.2, for a more complete description on the formal methods in software development and the difficulties associated with their wider acceptation.

One of the main goals of thesis work was to integrate formal methods with the system development effort. For this, viable strategies to support the integration of formal method techniques into the software development process are important. In this thesis we propose a JML-based strategy for incorporating formal specifications into software development processes for correctly writing Java programs. This JML-based strategy is in the style of Bertrand Meyer's design-by-contract, and makes use of JML specifications to write the contracts. The written JML specifications are integrated with the Java code itself, but they are written inside special marked comment blocks (see Section 3.1). The JML specifications are declared as `model`, by using that keyword we are declaring that the specification is abstract and they have no influence on the program execution or the Java code writings. This aspect also covers methods defined in the JML specifications declared as `model` and `pure`. That is, developers can write auxiliary specification methods if needed in the same way as a normal Java method, but written in special comment block by declaring them as `model` and `pure`. Again, this kind of methods has no secondary effects on the program execution. Our strategy provides solutions to some of the main difficulties in the wide acceptance of formal methods by the software industry. Our strategy offers basic guidelines for a formal development while supporting the developers' creative side, and by providing the developers a mean of communicating formal specifications with the end-users.

The strategy is part of an engineering integrated effort whereby software development is conducted in parallel with a *formal specification* pseudo-phase (see Figure 1 – Section 5). In this pseudo-phase, our strategy offers a guideline for formal development of Java programs by a stepwise process. In this stepwise process we evolve informal functional software requirements into JML formal specifications and we go through an intermediate stage in which semi-formal requirements are written. The informal functional requirements are suggested by the client (or stakeholders, end-users). Often these requirements are ambiguous, inconsistent and incomplete, due to the use of natural language. We then transform these informal requirements into semi-formal specifications, which are still written in natural language but in a structured way closer to the formal specifications (JML specifications). In the semi-formal stage we produce three kinds of semi-formal specifications: the semi-formal functional requirements, the class invariants and the system invariants. These semi-formal specifications are then ported into JML specifications (i.e., JML method functional specifications and JML invariants). In our strategy, the informal requirements and the semi-formal requirements can be used to support the communication between developers and clients. The formal specifications can be used to support the implementation of the system and communication between developers. Our strategy is defined as a guideline to address the problem of lack of methodologies to incorporate formal methods into software development processes, at least for Java programs. As for the lack of tools problem, there are various tools for supporting JML specification of Java programs. The main suite of tools is the JML Common Tools, which provides tools for compiling, and checking statically and in runtime the JML specifications against Java code. With only these tools we are capable of formally developing a program. Other tools are further available.

In this thesis we suggested a way to write semi-formal specifications in a manner that they can be easily mapped into JML specifications, while still being understood by the client (see Sections 5.4.1 to 5.4.3). Besides serving as an intermediate stage for writing JML specifications, with the semi-formal specifications we can clearly communicate with the client about the formal specifications. As the semi-formal specifications can be easily understood by an end user, then we can communicate the formal specifications to them. By this, our JML-based strategy provides a first solution to the problem of communication of formal specifications to the end-users.

Our strategy addresses the difficulty of providing some liberty to developer's creative side. When employing our strategy, it is recommended to design the system using Java interfaces, derived from the JML specifications, so as to increase the level of abstraction. This provides programmers liberty for their creative side while coding, as long as they respect the specifications. Furthermore, the use of JML abstract variables can provide support to the abstraction of complex data structures. Programmers can implement them later as desired while respecting the formal specifications.

Our strategy adheres to the Design by Contract principles. By following this strategy we can design and implement system components with formal specifications that describe contracts for their methods (JML method functional specifications) or even classes (JML invariants). Further, one can implement a correct system that uses these components and respects the specifications. That is, all the calls programmed for the specified components must respect the contracts defined in JML. Using this programming technique, the specified methods oughtn't to implement validations of their preconditions in their method bodies, because those validations are of the client's responsibility. That is, the method's preconditions must be assured by the clients when they call them. The use of these principles reduces considerably the amount of code from the specified methods implementations, leaving the validation code of preconditions for the client side. This aspect is useful when one has to develop a system with a client-server architectural style in which one of the parts must be light-weight, and also helps to reduce the redundancy of code for instances validation, i.e., a common error while programming a system where validations are made in both sides (*defensive programming*). It is possible to develop user interfaces components and other structures through the employment of our strategy and the use of JML.

We believe that our strategy is indeed simple enough to teach students about the incorporation of formal methods into software development processes. We can teach students on how to bring informal specifications closer to formal ones for developing small Java applications. The teaching should include the subject of basic logics for understanding how an informal functional requirement can be ported into semi-formal specifications, i.e., how we can write semi-formal specifications to be closer to JML specifications. The most important aspects to teach students about our strategy is to present them with ways of producing semi-formal specifications, teaching them the basics of JML and to think on the concept of invariants, abstract variables, and the design by contract principles. A student should basically learn how to pass an informal functional requirement into semi-formal specifications, and then to JML specification.

We also want to emphasise the importance of thinking of invariant properties when developing software. Thinking about invariants prior to writing code is a practice to which programmers do not easily adhere. Having a formal specification of an application and systematically using a tool, i.e., the JML Common Tools, for checking the correctness of the code as it is written forces programmers to think about how the written code affects the consistency and the correctness of the whole program. It is our experience that invariants are

the key notion in formal software development that makes a difference with respect to traditional (non formal methods based) software engineering methodologies (Catano, Barraza, García, Ortega, & Rueda, 2008). In general, programmers feel intimidated by the idea of coming up with an invariant. Often, they design code that can make their programs be in an inconsistent state. We strongly believe JML helps in this sense, from furnishing a friendly Java-like syntax, to making it possible to use first-order logic predicates in JML specifications naturally.

Our strategy can be employed for a formal development of any Java program. In particular, the strategy has been used by Ricardo Rodrigues (Rodrigues, 2009) in the full development of the HealthCard introduced by us in Section **Erro! A origem da referência não foi encontrada.**. In the beginning of this development, some problems related to the definition of our strategy arose. One of the problems was about the generation of formal specifications from informal ones. When initially applying our strategy on the HealthCard, we didn't think of having a semi-formal phase. We faced the challenge of defining a way to write JML specifications from the informal functional requirements. It was hard to write formal specifications directly from informal ones, like it was hard to modify them when the requirements changed. The solution came with the creation of an intermediate step of semi-formal specifications for developing formal specifications from informal ones. Having semi-formal specifications we could start shaping informal functional requirements given by the clients into an organized and structured way, prepared to be mapped into JML specifications. At the same time, we can use semi-formal specifications to communicate the formal specifications to the end-users.

The strategy described in this thesis can be easily employed in the formal development of other Java (or Java Card) applications. As it helped developing a correct HealthCard application for managing information of a patient, it could help other types of smart card applications like the HealthCard as well. For example, our strategy could be employed in the development of student cards for managing information about their academic life, or for smart cards applications for keeping and managing information on a member (of libraries, of sport clubs, etc.), or even smart cards for keeping basic information of elderly people, children and teens for getting service discounts, etc. There are numerous possibilities of employing this strategy for formally develop applications, similar to the HealthCard, for serving the local communities.

As seen in Section 4, there are other strategies for incorporating formal methods into software development processes of Java programs, but our strategy is more cost-effective and straight-forward for a Java programmer because JML uses Java syntax. The strategy in this thesis is described as being used to incorporate JML formal specifications in the development of a Java/Java Card application, but although we chose JML as the formal specification language, these ideas can be adapted to the development of C++ programs, with formal specifications written in the ACSL (ANSI/ISO C Specification Language) (Baudin, Filliâtre, Marché, Monate, Moy, & Prevosto) language instead, and the verification work accomplished with the Frama-C Tool (The Frama-C Tool).

Some future work can be done to enhance the usefulness of our strategy. In this thesis we suggested how one could write semi-formal specifications. We highly propose further studies and the development of a semi-formal language understandable by a common software development client while also being easy to be mapped into JML formal specifications. The semi-formal specification language could be based on our suggestion, but it needs to be standardized. Also, a tool can be developed for automatically convert simple semi-formal expressions into JML specifications. The author of this thesis and Ricardo Rodrigues, have written a prototype tool that converts semi-formal specifications into JML formal specifications for the simplest cases.

Bibliography

(n.d.). From The ESC/Java 2 Tool: http://secure.ucd.ie/products/opensource/ESCJava2/

(n.d.). From The Jack Tool: http://www-sop.inria.fr/everest/soft/Jack/jack.html

(n.d.). From The Krakatoa Tool: http://krakatoa.lri.fr/

(n.d.). From KindSoftware: ESC/Java2: http://secure.ucd.ie/products/opensource/ESCJava2/

(n.d.). From JACK: Java Applet Correctness Kit: http://www-sop.inria.fr/everest/soft/Jack/jack.html

(n.d.). From Krakatoa: a verification tool for Java programs: http://krakatoa.lri.fr/

(2007, June 20). Retrieved September 1, 2009 from EclipseJCDE User Guide: http://eclipse-jcde.sourceforge.net/user-guide.htm

(2008, March 2). Retrieved September 1, 2009 from EclipseJCDE: http://eclipse-jcde.sourceforge.net/

ARC - Care Parent Network. (n.d.). *Information and Medical Forms*. Retrieved December 2008 from CARE Parent Network Resource : http://www.contracostaarc.org/html/careresources.html

Baudin, P., Filliâtre, J. -C., Marché, C., Monate, B., Moy, Y., & Prevosto, V. (n.d.). From ACSL: ANSI/ISO C specification language: http://frama-c.cea.fr/-download/plug-in_development_guide.pdf

Bell, D. (2004, September 15). *UML basics: The class diagram*. From IBM: http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/

Breunesse, C.-B., Cataño, N., Huisman, M., & Jacobs, B. (2008). *Formal Methods for Smart Cards: an experience report.*

Cardlogix Corporation. (2009). *Smart Card Standards*. Retrieved September 2009 from Smart Card Basics: http://www.smartcardbasics.com/standards.html

Catano, N., Barraza, F., García, D., Ortega, P., & Rueda, C. (2008). A case study in JML-assisted software development. In P. Machado, A. Andrade, & A. Duran (Ed.), *Brazilian Symposium on Formal Methods (SBMF)*, (pp. 5-21).

Cataño, N., & Huisman, M. (n.d.). *Electronic Purse Case Study.* From http://www-sop.inria.fr/lemme/verificard/electronic_purse/

Cataño, N., & Huisman, M. *Formal specification and static checking of Gemplus' electronic purse using ESC/Java.* INRIA Sophia-Antipolis, France.

Classen, I., Ehrig, H., & Wolz, D. (1993). *Algebraic Specification Techniques and Tools for Software Development.* World Scientific Publishing Co. Pte. Ltd.

Cornell University. (n.d.). *Making Appointments* . Retrieved December 2008 from Gannett Health Services : http://www.gannett.cornell.edu/accesstocare/appointments.html

Eiffel Software. (n.d.). *DESIGN BY CONTRACT AND ASSERTIONS*. Retrieved January 2009 from Eiffel Software: http://archive.eiffel.com/doc/online/eiffel50/intro/language/invitation-07.html

Fraser, M. D., Kumar, K., & Vaishnavi, V. K. (1994, October). Strategies for Incorporating Formal Specifications in Software Development. *Communications of the ACM , 37*.

Iowa State University. (2002). *Package org.jmlspecs.models*. From JML and MultiJava Documentation: http://opuntia.cs.utep.edu/utjml/jml-javadocs/org/jmlspecs/models/package-summary.html

Júnior, R. D., Figuereido, J. C., & Guerrero, D. D. (2005). Design by Contract com JML. In UNISINOS (Ed.), *XXV Congresso da Sociedade Brasileira de Computação* , (pp. 1455-1499). São Leopoldo.

Kemmerer, R. A. (1990, September). Integrating Formal Methods into Development Process.

Kostrubiak, A. (2009). *Integration of Java Generics Into The jmle.*

Krause, B., & Wahls, T. *jmle: A Tool for Executing JML Specifications via Constraint Programming.* Dickinson College, Department of Mathematics and Computer Science.

Leavens. (n.d.). *Java Modeling Language*. From SourceForge.net: http://sourceforge.net/tracker/?func=detail&atid=510629&aid=2822469&group_id=65346

Leavens, G. (n.d.). *Downloading the JML Common Tools*. From The Java Modeling Language (JML): http://www.eecs.ucf.edu/~leavens/JML/download.shtml

Leavens, G. (2008, May 20). *JML Reference Manual*. From http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman.html

Leavens, G. T. (2008, July 29). *Documentation*. From The Java Modeling Language (JML): http://www.eecs.ucf.edu/~leavens/JML//documentation.shtml

Leavens, G. T., & Cheon, Y. (2006). *Design by Contract with JML.* Iowa State University; University of Texas at El Paso, Dept. of Computer Science.

Liu, S., Offutt, A. J., Ho-Stuart, C., Sun, Y., & Ohba, M. (1997). *SOFL: A Formal Engineering Methodology for Industrial Applications.*

Liu, S., Takahashi, K., Hayashi, T., & Nakayama, T. (2009, June). Teaching Formal Methods in the Context of Software Engineering. *inroads — SIGCSE Bulletin , 41*, pp. 17-23.

McDermott, B., Elliott, J., Fabbri, L., Panseri, P., & Primerano, F. (1998, January 9). *Disadvantages of Smart Cards*. (Massachusetts Institute of Technology) Retrieved January 22, 2009 from Smart Cards: http://web.mit.edu/ecom/Spring1997/gr12/4DISADV.HTM

Medical Assistant.net. (n.d.). *Medical Assistant Net - What is a SOAP Note?* Retrieved December 2008 from Medical Assistant: http://www.medicalassistant.net/soap_note.htm

Meyer, B. (1992). Applying "Design by Contract". *Computer* , pp. 40-51.

Meyer, B. (1997). Design by Contract: Building Reliable Software. In *Object-Oriented Software Construction* (pp. 331-410). Prentice Hall.

NetBeans. (2009). *NetBeans IDE*. From NetBeans.org: http://www.netbeans.org/

Oostdijk, M., & Warnier, M. *On the combination of Java Card Remote Method Invocation and JML.* Univ. Nijmegen, Dept. Com. Sci.

Ort, E. (2001, January). *Writing a Java Card Applet*. From Sun Developer Network (SDN): http://java.sun.com/javacard/reference/techart/intro/

Ortiz, C. E. (2003, May 29). *Sun Developer Network*. Retrieved January 2009, from An Introduction to Java Card Technology - Part 1: http://java.sun.com/javacard/reference/techart/javacard1/

Priestley, M. *The Logic of Correctness in Software Engineering.* University of Westminster, Cavendish School of Computer Science, London.

Riehle, D. (2000). *Method Types in Java.* SKYVA International.

Rodrigues, R. (2009). *JML-Based Formal Development of a Java Card Application for Managing Medical Appointments.* University of Madeira.

Sommerville, I. (2000). Formal Specification. In *Software Engineering* (pp. 159-169).

Swiss Federal Institute of Technology Zurich. (n.d.). *JML2 Eclipse Plug-In*. From Department of Computer Science - Chair of Programming Methodology: http://www.pm.inf.ethz.ch/research/universes/tools/eclipse

The Frama-C Tool. (n.d.). From http://frama-c.cea.fr

The Krakatoa Tool for Java Program Verification. (2009, 1 30). Retrieved 10 22, 2009 from Krakatoa: http://krakatoa.lri.fr/krakatoa0.html

Tucker, A., & Noonan, R. (2001). Program Correctness. In *Programming Languages: Principles and Paradigms.* McGraw-Hill Science/Engineering/Math.

van den Berg, J., & Jacobs, B. (2001). The LOOP compiler for Java and JML. In *In Proceedings of TACAS* (pp. 299-312). Springer.

Vasconcelos, V. T., Nunes, I., & Lopes, A. (2008). Monitoring Java Code Using ConGu. *WADT 2008.* Italy.

Warnier, M. (2006). *Language Based Security for Java and JML.*

Warnier, M., & Oostdijk, M. (n.d.). Java Card Remote Method Invocation. University of Nijmegen.

List of Figures

Figure 1. The Software Development Process 22
Figure 2. Semi-formal relations with the Structure Model 29
Figure 3. HealthCard application structure 34
Figure 4. Proposed information held on a smart card for medical appointment management 36
Figure 5. HealthCard System 37
Figure 6. Architecture of a Java Card Application [26] 39
Figure 7. Java Card Remote Method Invocation architecture [30] 42
Figure 8. JCRMI applet implementation process [30] 43
Figure 9. Use Cases of the patient's allergies' information management example 44

List of Code

Code 1. Example of a Medicines class specified with pseudo-specification 4
Code 2. Pre-condition example for a factorial computation [8][3] 6
Code 3. A redundant test [3] 6
Code 4. Example of a Medicines class implementation with an invariant 7
Code 5. Example of a Services class referencing Medicines and Appointments classes with a system invariant 8
Code 6. Example of how JML can be used to specify a method 15
Code 7. Example of how JML can declare an abstract variable 16
Code 8. Example of how JML abstract variables can be represented by concrete values 16
Code 9. Specified addAllergy method from Allergies interface 46
Code 10. Relating abstract specifications with actual Java code in *Allergies_Impl* 47
Code 11. A system invariant as JML invariant 47

List of Tables

Table 1. A design by contract example [8] 5
Table 2. Some JML expressions 14
Table 3. A command APDU format [26] 40
Table 4. A response APDU format [26] 41