JML-Based Formal Development of a Java Card Application for Managing Medical Appointments

Authors:

Ricardo Miguel Soares Rodrigues, 2064903

Advisor:

Néstor Cataño

Acknowledgments

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

I would like also to thank João Pestana, my colleague, for his collaboration on this thesis and being by my side while we worked together. 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.

Index

Figures

Code

Tables

1. Introduction

Although formal methods can dramatically increase the quality of software systems, they have not widely been adopted in software industry. Many software companies have the perception that formal methods are not cost-effective cause they are plenty of mathematical symbols that are difficult for non-experts to assimilate. The Java Modelling Language (short for JML) Section 3.3 is an academic initiative towards the development of a common formal specification language for Java programs, and the implementation of tools to check program correctness. This master thesis work shows how JML based formal methods can be used to formally develop a privacy sensitive Java application. This is a smart card application for managing medical appointments. The application is named HealthCard. We follow the software development strategy introduced by João Pestana, presented in Section 3.4. Our work influenced the development of this strategy by providing hands-on insight on challenges related to development of a privacy sensitive application in Java. Pestana's strategy is based on a three-step evolution strategy of software specifications, from informal ones, through semi-formal ones, to JML formal specifications. We further prove that this strategy can be automated by implementing a tool that generates JML formal specifications from a well-defined subset of informal software specifications. Hence, our work proves that JML-based formal methods techniques are cost-effective, and that they can be made popular in software industry. Although formal methods are not popular in many software development companies, we endeavour to integrate formal methods to general software practices. We hope our work can contribute to a better acceptance of mathematical based formalisms and tools used by software engineers.

The structure of this document is as follows. In Section 2, we describe the preliminaries of this thesis work. We make an introduction to the application for managing medical applications we have implemented. We also describe the technologies used in the development of the application. This section further illustrates the Java Card Remote Method Invocation communication model used in the medical application for the client and server applications. Section 3 introduces software correctness, including the design by contract and the concept of contract in JML. Section 4 presents the design structure of the application. Section 5 shows the implementation of the HealthCard. Section 6 describes how the HealthCard is verified and validated using JML formal methods tools. Section 7 includes some metrics of the HealthCard implementation and specification. Section 8 presents a short example of how a client-side of a smart card application can be implemented while respecting formal specifications. Section 9 describes a prototype tools to generate JML formal specifications from informal specifications automatically. Section 10 describes some challenges and main ideas came acrorss during the development of the HealthCard. The full formal specification and implementation of the HealthCard smart card application presented in this document can be reached at https://sourceforge.net/projects/healthcard/.

2. Preliminaries

In the following, we present the HealthCard smart card application we have formally implemented using JML technology. We start presenting some problems related to the management of medical appointments, and define some important concepts for the domain application. Then, we propose a solution to the development of a Java Card Health Card Application that deals with privacy sensitive medical people information. We introduce the Smart Card technology that will provide support to the development of our application, and introduce the programming language Java Card, which will be used to develop the card side application.

2.1. Managing Medical Appointments

In this section we present and discuss a common problem about medical appointments (Section 2.1.1). Next, we describe our solution to the development the HealthCard, in which we also make a short introduction to the proposed solution technologies, namely, smart cards.

2.1.1. General Description of the Problem

Finding a medical history is not always an easy task when a doctor is checking a patient, especially during the first checking. In most cases, medical histories are kept in a particular medical centre as hard-copy files. Hence, if a patient has to go to a different medical centre other than to the one he usually goes, it might be the case that he could not know how to tell the doctor about his medical problem, and if he could, doctors always need a dossier with patients' information (that is, they need a medical history). Treating a patient without having his/her medical information can be time-consuming. The following case describes this problem in real life:

"John is British and usually travels between Portugal and the USA. One day, while in Portugal, he got sick and went to the doctor. The Portuguese doctor started making a diagnosis. He asked John if he already got his Tetanus immune shot, but John didn't know. After listening John and running some medical exams, the doctor identified the possible problem and then prescribed some medications to John.

One week later, John went to New York and his health got worse, so he decided to go to the nearest medical centre to see a doctor. The American doctor asked him if he already had gone to see a doctor about his problem before, and what medication he has been taking. John didn't know all the names of the medication prescribed by the Portuguese doctor, and worse, he couldn't explain exactly what problem he was suffering!"

2.1.2. Problem analysis

The main issue on the previous scenario is the lack of communication between patients and the medical staff, especially between different medical centres. Lack of communication means a possible inefficient patient treatment. Knowing relevant patient information can save time and result in a faster medical diagnosis of the patient situation.

Typically, a patient would not know how to express to another doctor all the information about his health problems. He would not know what medication he's taking, or his medical history, and drug allergies or other types of allergies he suffers, and could consequently be mistreated. In any case, even if the doctor explained the medical problem to the patient well and even if the patient could roughly explain his medical problem to others; a new doctor would anyway need a ”record" with the patient's information. This information is usually kept as "dossiers" in medical centres.

2.1.3. Proposed solution

We propose here a solution to the problem described above. Our solution builds on the software development strategy introduced by João Pestana (see Section 3.4), formal methods, JML (see Section 3.3), and smart cards (see Section 2.2). The reason why we use of this strategy in the development process of the HealthCard application is due to the application's domain nature in which people's medical information is involved. This information is privacy sensitive. Through the use of formal methods in this strategy we develop a correct smart card application. Formal methods ensure that the application behaves as described in its specification. When developing a software application for privacy sensitive domains, like medical ones, the programmer must implement mechanisms that ensure that information is not disclosed to non-authorized parts. In our work, such mechanisms are implemented through the use of JML formal specifications, and the use of formal methods tools to check program correctness (see Section 3 for a description about software correctness). Security properties such as authentication, confidentiality or integrity are difficult to express with JML, and are out of the scope of this master thesis work. [1]

The HealthCard application runs on 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 smart card is a pocket-sized card with embedded integrated circuits that can hold and process data. A typical smart card includes in-card applications (the applets running on the card), a card reader-side, and back-end elements (a computer communicating with the card applets). For implementing the in-card applications we use the Java Card language (see Section 2.2). 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.[2] 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 1 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.

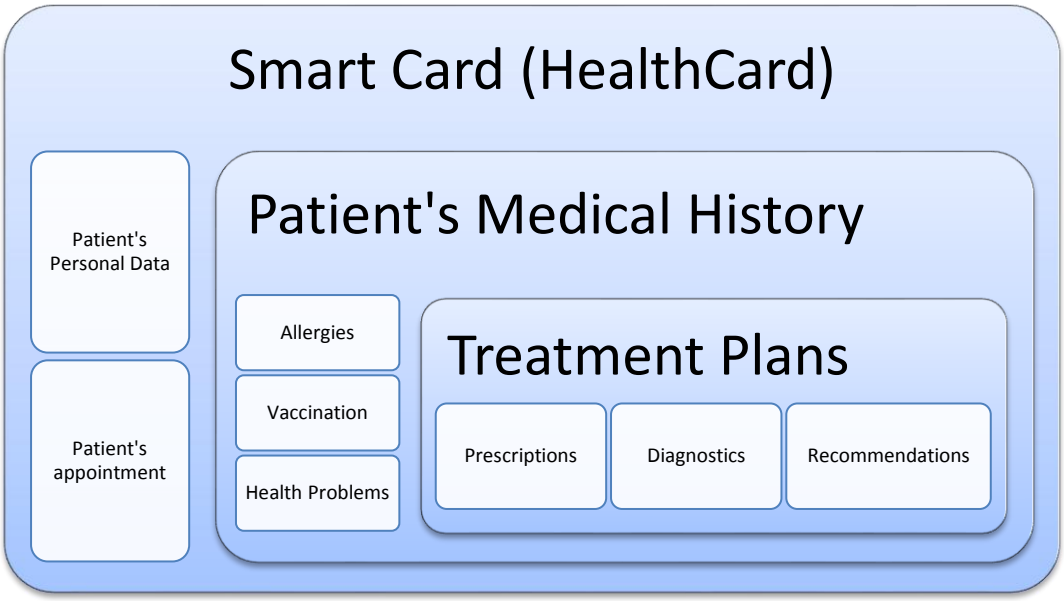

Figure 1. Proposed information held on a smart card for medical appointment management

For managing the data stored in 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 [2] (see Section 2.2). We can follow the strategy that supports the software development process of the Health Card application as at same time uses *Java Modelling Language (JML)*[1] for formally specify the Health Card application based on the informal requirements. These JML formal specifications used in this strategy will lead us to a correct implementation of our application. Also, we propose the use of JML-based tools to check for correctness of the implementation.

2.1.3.1. Proposed System Architecture

The architecture of our proposed application is illustrated in Figure 2. A patient can use his smart card in any medical centre that has our system implemented.

[1] JML is a formal behavioral interface specification language for Java which includes the essential notations used in Design by Contract as a subset. [33]

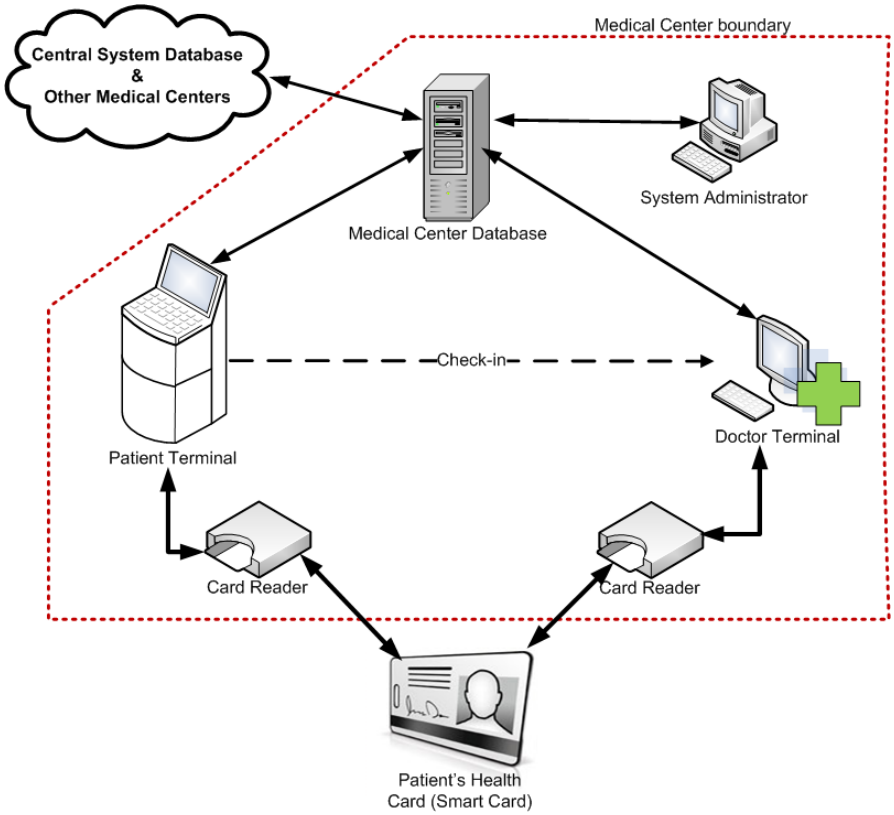

Figure 2. Proposed system using a smart card

This system architecture 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. This is used by medical staff, possibly a doctor providing a way to access the patient's personal and medical information. 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.

Proposed System Components

This section presents the list of components of the proposed system architecture.

Health Card (*smart card*) contains personal and medical information about the card owner (patient) and his scheduled appointments.

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.

2.1.3.2. Proposed System Scenario

At the end of the development of our system, the following scenario should be:

"John has an appointment with his doctor today at 5:00 PM. John will be using his HealthCard smart card at the Yorkshire medical centre for checking-in prior to his appointment. The Yorkshire medical centre has a smart card compatible card reader set for patients to check-in using their HealthCards. Therefore, doctors can access patient's medical information relevant to appointments, e.g., general description of the patient's medical problem, past treatment information, historical of medicines used during past treatments, previous record of allergies related to a particular medical substance, age and gender of the patient, etc. HealthCards are dynamic: last week, when John was away on vacations in the United States, he fell back into illness after his 7 hours trip from Heathrow airport in London to the John Kennedy airport in New York. He visited an American doctor who read John's medical information from his HealthCard. The American doctor prescribed an alternative complementary treatment for John's health problem. The details of this complementary treatment were stored into John's HealthCard smart card. John's doctor will be able to consult the details of the American doctor's prescription when John will be back into Yorkshire."

2.2. Smart cards and Java Card

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. [2] 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. [2] 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. [2] ISO 7816 is the international standard for smart cards that use electrical contacts on the card. [3]

2.2.1. 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). [2]

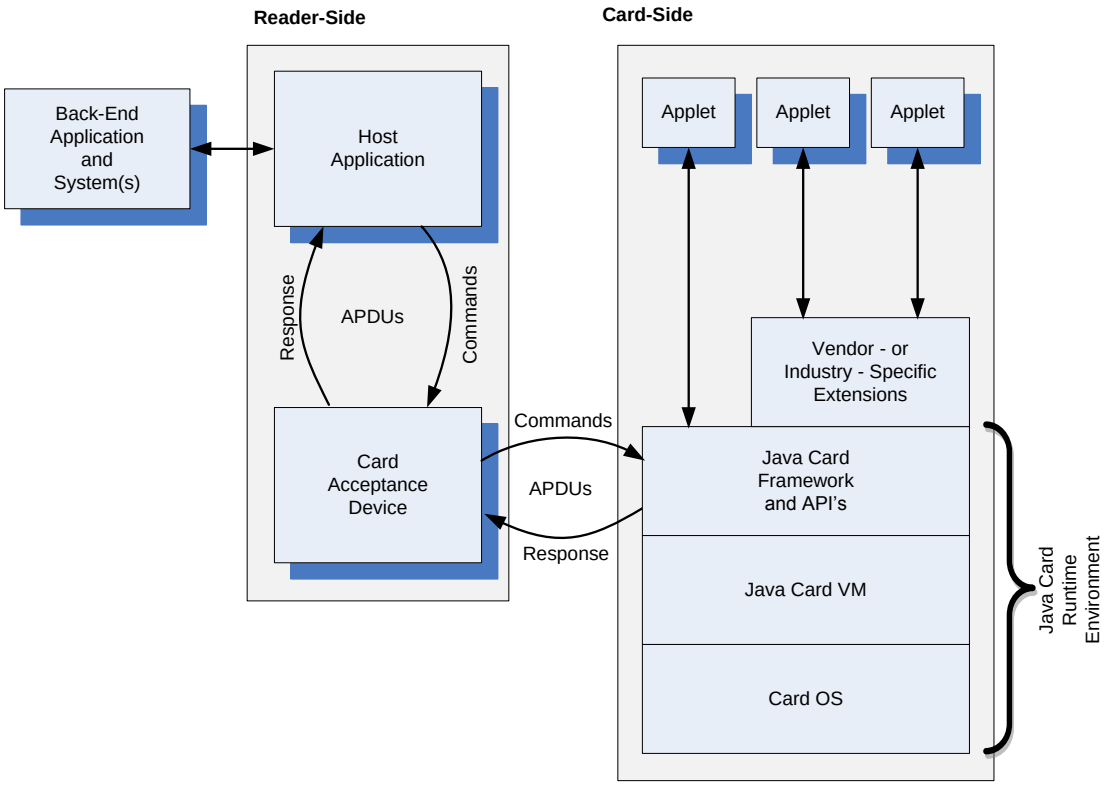

Figure 3. Architecture of a Java Card Application [2]

2.2.1.1. 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.

2.2.1.2. 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.

2.2.1.3. *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).

2.2.1.4. 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.

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

According to ISO 7816-5 standard, each smart card application must have an application identifier (AID). [3] 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. [4]

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 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. [2]

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 1. A command APDU format [2]

Command APDU						
Header					Body	
(required)					(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. [3] In case of successful execution, they contain 0x9000.

Table 2. A response APDU format [2]

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

Our Java Card implementation of the application for managing medical appointments 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 [5] 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.

2.2.3. 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 makes it possible to directly call methods from the Java Card smart card. [6] 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*. [6] 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. [6]

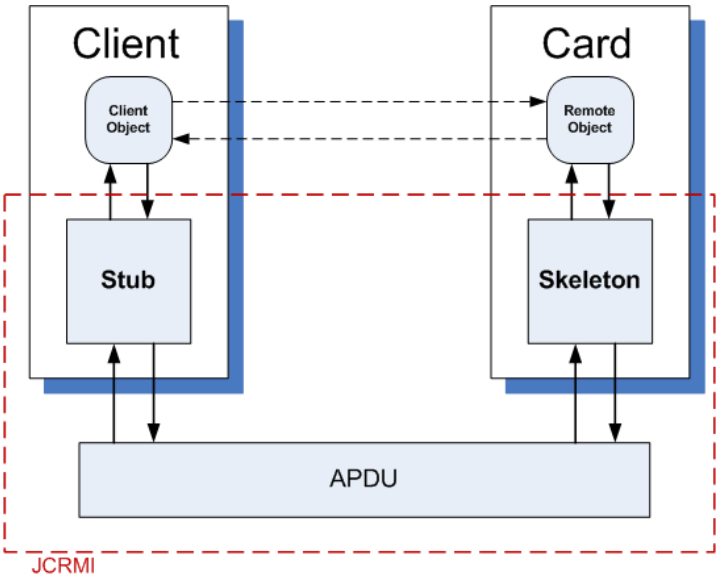

Figure 4. Java Card Remote Method Invocation architecture [6]

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. [6]

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 5 illustrates this whole process.

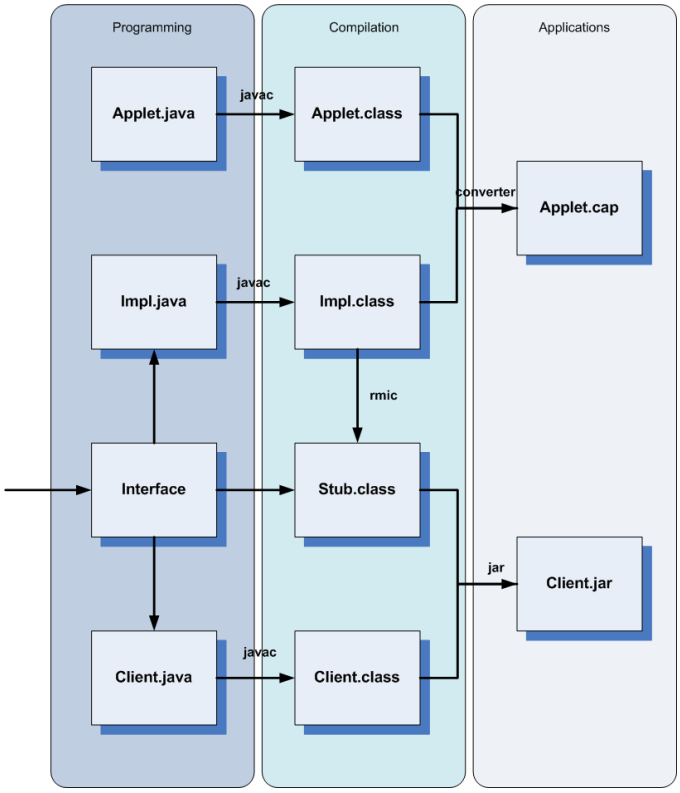

Figure 5. JCRMI applet implementation process [6]

3. Formal Software Development

During this section, we introduce the use of formal methods in software development, describing the software correctness and its fundamental essence. We continue, giving a brief introduction to the Java Modelling Language (JML) and for last we consider the strategy of João Pestana, as followed strategy of this thesis work.

3.1. 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. 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 prone to specifications inconsistencies and their incompleteness. Formal specifications make possible the capture of software requirements unambiguously as part of a software engineering methodology. By using formal specifications, one involves investing more effort in the early phases of software development cycle, especially in requirement analysis. This 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, as also the cost related to the implementation and validation phases. However, according to Sommerville [7], 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. [8] Sommerville [7] 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. [8]

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 [7], 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 [7] 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 use in software development processes, one has the challenge of integrating the formal methods with the

system development effort, especially in large scale development projects. For this, viable strategies for supporting the integration of formal method technique into the software development process are important, and without existing strategies it may be difficult to integrate formal methods into the real-world development project.

3.1.1. Formalization Strategies for Software Development Processes

The way people can use formal methods to formalize their informal specifications (i.e., the client's requirements to a system to be developed) into formal ones 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 [9] 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 to much implementation details, so later they can be incrementally detailed until the system can be coded according to them. Also Jones [10] uses a similar process through his suggestion that proof obligations of VDM decomposition rules can stimulate design steps. Others like Miriyala and Harandi, and Wing [10], have process propositions where high-level formal specifications of the system can be derived directly from a precise English statement of critical requirements. A strategy which 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 which 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 recur to one or more semi-formal specifications providing us 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 by using a *transitional formalization process*. [10] We can say that the transitional formalization of the specifications can be divided into three degrees: - informal, semi-formal and formal. At the informal state, 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 more close 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. By 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 more closely to the 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 from the clients. [10] An example of a strategy using a transitional formalization process is the strategy proposed by Kemmerer [9] 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.

3.2. 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. [11] 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 [12]. 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 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*." [12]

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, 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. [12] 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. [12]

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. [12] Further, in Section 3.3, we present 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).

3.2.1. 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 Section 3), 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. [12] 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. [12]

Before discussing further the design by contract we'll show below an example of a Medicines class operation specified with pseudo-specification on how assertions are used in practice [13]. Here, preconditions and postconditions are represented respectively by **require** and **ensure** keywords. [12] 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.
     requires
        medicine.length <= name_length
     do
              insert(medicine)
```

```
        ensures
            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. [12]

3.2.1.1. Obligations and Benefits

The precondition is related to the client in a way that it defines the conditions under which is legitimate for a client 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. [12] The following example taken from [14] shows how *design by contract* plays out for the factorial computation in respect for client/suppliers' benefits and obligations.

Table 3. A design by contract example [14]

	Obligations	Benefits
Client	(*Satisfy precondition* :) Pass	(*From postcondition* :) Receive computed
Supplier	(*Satisfy postcondition* :) Compute	(*From precondition* :) Can assume that

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. [12] 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 test 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. [12] 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, from Table 3, 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.

3.2.1.2. 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. [12] 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

    do … end
```

Code 2. Pre-condition example for a factorial computation [14][12]

We may write the "*do … end*" algorithm for computing the factorial without concerning whether is negative or not. This concern is taken care by the precondition which becomes the clients' responsibility. [12] If the "*do*" clause was on the form as seen in Code 3:

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

Code 3. A redundant test [12]

Then the test "" 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[2] and may imply a performance penalty. [12] We have applied the principle of non-redundancy in the development of the HealthCard. By applying the principle of non-redundancy, we are light weighting the card side applications. 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. While developing the HealthCard system, employing this principle is advantageous due to the limited memory of smart cards.

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.

3.2.1.3. 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 consider an invariant as being an extension for both preconditions and postconditions of every class's routines. [12] 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 and P*} *A* {*INV and Q*}

This 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". [12] 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 implementation of the Medicines class example and its pseudo-specifications shown in this section, we demonstrate in Code 4 how we could specify a class invariant. [13]

```
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
```

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

```
        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. [13] 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 at a class that has references to ***X*** and ***Y*** objects.

In the following example shown in Code 5, ***X*** and ***Y*** are exemplified respectively by the classes ***Medicines*** and ***Appointments***. The defined **invariant** is a system invariant because it affects instances of the two different classes which basically states that for every medicine object instances obtained through the Medicines instance, their prescription date attribute must be higher or equal to the respective appointment's date, obtained through the Appointments instance, when the medicine was prescribed for the first time. That is, for all medicines and appointments instances if a medicine instance has an appointment ID 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 that value of a medicine'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 and used to the preconditions, postconditions and even invariants is the *inheritance*.

3.2.1.4. 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 (see Section 3.3). Although inheritance is one of the pillars of the object oriented paradigm flexibility, many programmers have the difficulty in use it correctly. [15] 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 routine's behaviour specification described (contract) in the super-class. [15] This incompatible redefinition is a problem attained 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. [15]

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 [15]:

3.3.1. 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.

3.3.1. 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.

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. [15]

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

3.2.1.5. 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 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*. [15]

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. [15]

3.3. 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 [16]. 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 [17; 18; 19; 20]. All the concepts discussed in the Design by Contract section (see Section 3.2.1), that is, the notion of contracts along with its preconditions and postconditions; and the concepts of invariants, inheritance and exceptions, also apply to JML.

3.3.1. 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 `(\forall T 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 3.3.1. 1, but the reader is invited to consult [21] for a full introduction to JML.

3.3.1. 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 4. 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.

3.3.2. 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.

3.3.2.1. 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 [22], 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 [22] 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.

3.3.3. The JML Common Tools

The JML common tools [23] is a 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 [24].

3.4. The JML - Based Software Development Strategy

This strategy is introduced by João Pestana [25]. The strategy integrates formal specifications written in JML to the software development of Java applications. The strategy evolves informal functional requirements into formal specifications, which can be employed as part of existing object-oriented software development methodologies[12]. Hence, software developers can define precise interface specifications for underlying software components,

based on data types and the conceptual metaphor of the design-by-contract [16]. In Figure 6, the strategy is outlined as a formal specification pseudo phase (bottom of Figure 6). This strategy is similar to the Kemmerer's "Parallel" approach in that he also defines an intermediate step that introduces semi-formal specifications before writing the JML formal specifications.

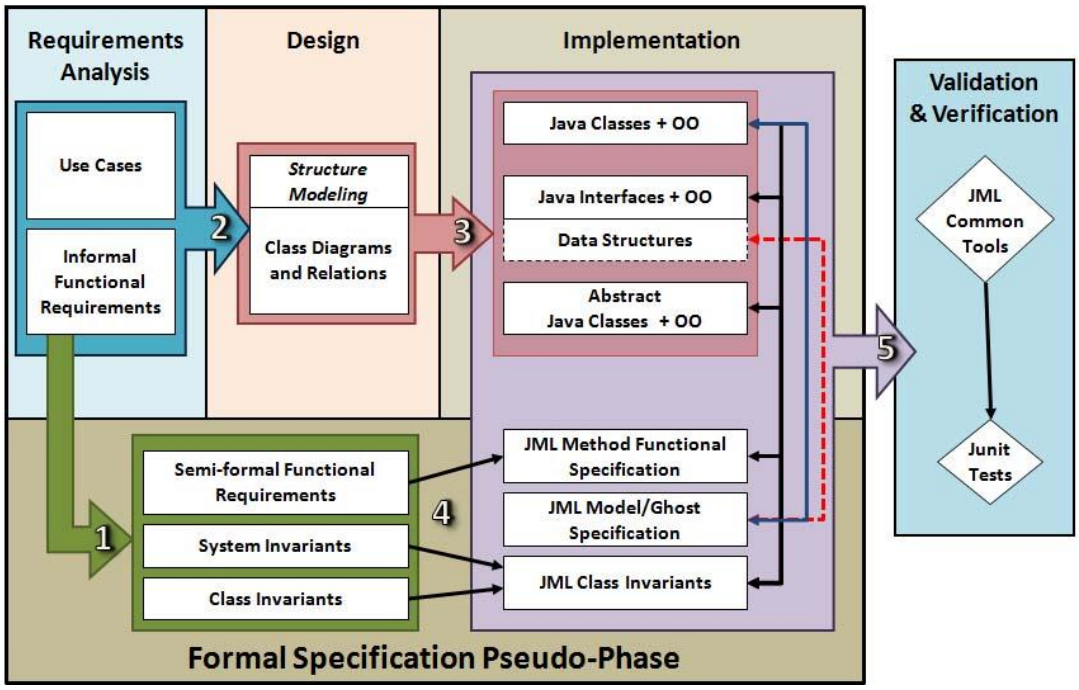

Figure 6. The Software Development Process

The formal specification pseudo-phase runs in parallel with the analysis, design, and implementation phases in an integrated manner. There is no restriction in any phase to occur before or after any other phase, so that arrows 1 to 5 in Figure 6 convey information on usage rather than on precedence in time. Software development phases are iterative so that they can be revisited at later phases to obtain a correct implementation of the system. During the analysis phase, requirements are gathered, and two documents are produced, namely, the use cases document and the "informal" functional requirements document. As informal functional requirements are expressed in a natural language, inconsistencies can be introduced during the analysis phase.

Some aspect new in the strategy is the formal specification pseudo-phase, where the informal functional requirements document is first evolved into"semi-formal" requirements document (see Arrow 1), and then ported into (formal) JML specs (see Arrow 4). Having formal specifications expressed in JML makes it possible to use JML based formal methods tools to check for flaws. The semiformal requirements document is composed of three documents.

The semi-formal functional requirements document (ported into JML method specifications), the class invariant document, and the system invariant document (these two are ported into JML class invariant specifications) (see Arrow 4). Evolving the informal functional requirements document into the semi-formal one involves expressing informal requirements into an ***if*** <event/condition> ***then*** <restriction/rule> form, this is an idea proposed by the author of this strategy.

During the design phase, the requirements gathered from the analysis are used to define the structure of the system (see Arrow 2), which is later used to write classes, their attributes, their methods, and the relations among them (see Arrow 3). These classes are asserted with the JML specifications written during the formal specification pseudo-phase.

During the implementation phase, we start by writing Java interfaces and Java abstract classes. From the semi-formal requirements document, JML functional specifications are written within the (abstract) methods in these Java interfaces and classes, and JML class invariants are written, modeling global properties of the system. Finally, JML abstract variables are defined to describe the distinct abstract data types used in the application, and how they are manipulated through class inheritance (see Section 3.2.1.4). JML specs provide support to the writing of correct code for concrete classes that implement the interfaces and the abstract Java classes. JML specifications also provide support to a business contract programming style of programming, in accordance with Bertrand Meyer's design-by-contract principles.

During the validation-and-verification phase, the implementation is checked against the specifications (Arrow 5), using the JML Common Tool[24]. Furthermore, it is possible to go back to a previous phase and make amendments to the JML specifications or the implementation itself. 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 these classes and interfaces, or JML specs can be validated in isolation [26].

This strategy will be used during our implementation in this thesis work where it be described all steps, referred in this section and how helpful will be the formal specification in the software development process.

4. The Approach

To develop the HealthCard, we first obtain the information about the domain and concepts related to medical appointment. Then, we use these concepts to obtain the requirements and uses cases, so that we can apply João pestana's software development strategy (see Section 3.4 for a description of this strategy). As part this strategy, we use JML to provide support to the writing of formal specifications. Therefore, we can validate the correct implementation of the Health Card application introduced in Section 2.1.

4.1. Domain Concepts

Before getting the Use Cases and Functional Requirements, the first step of our development was the extraction of the most relevant concepts from the problem at hand, and the comprehension of the medical appointments domain. The concepts extracted are shown in Figure 7. These concepts are things related to the domain where the future system will be applied, and their purpose is to help to understand the domain and to allow for a better comprehension when talking with the stakeholders, formulating use cases and functional requirements. Eventually, later these domain concepts are shown as class attributes at the design phase, and they will be modelled in JML specifications as abstract variables representing class attributes.

The extraction of domain concepts was carried out by talking with specialized people (doctors, medical students, nurses, patients) and doing literature review about medical appointments [27; 28].

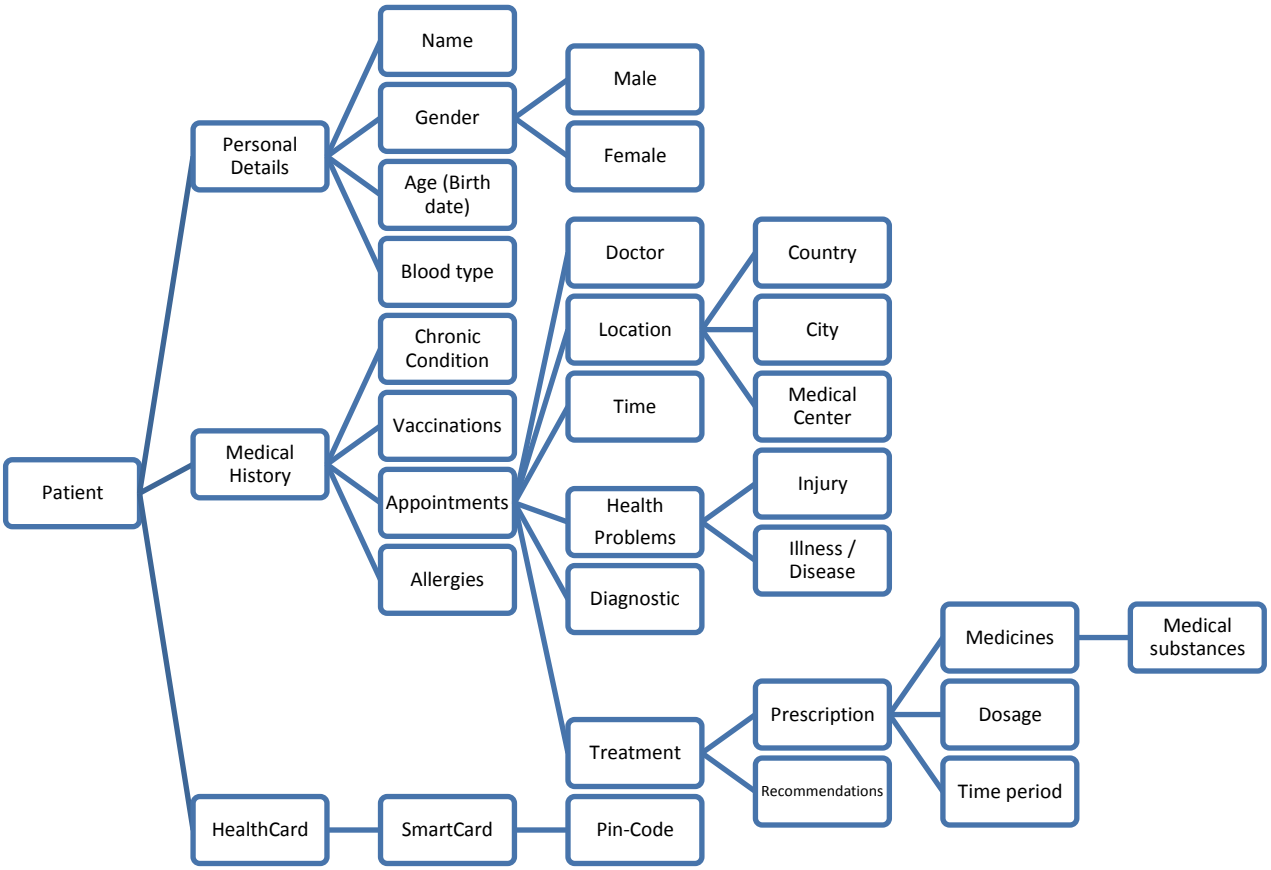

Figure 7. Summary of the medical appointments concepts hierarchy

Next, with the main domain concepts obtained, a hierarchy between concepts was made so we could have a good perspective of the domain, as seen in the above Figure 7. Of course, our concept model suffered an evolution while our comprehension of the domain was growing.

One of the specialized people (a doctor) that we've talked suggested that we follow SOAP notes [29] as a metaphor to implement our Health Card system. SOAP notes are written to improve communication among the medical staff when caring for a patient. These are used to display the assessment, problems and plans in an organized format. The letters S-O-A-P stand for Subjective, Objective, Assessment and Plan. Where: - Subjective consists of patient's symptoms descriptions; Objectives are related to measurable symptoms such as blood pressure, temperature, pulse or diagnostic results; Assessment is the diagnosis of the patient's condition made by a doctor; and Plan refers to medical prescriptions which may be treatments, other diagnostic tests or medical recommendations. SOAP notes facilitate better medical care when used in the patient's record for review and quality control. [29] The following two examples describe SOAP notes:

SOAP Note Example 1:

Patient Name: *Robert Dreg*

DOB: *09/17/1967*

Record No. *D-679dk978*

Date: *12/4/2007*

S—*Pain in left hip x 3 months; worse when walking or doing exercise. NKDA.*

O—*Wt. 195 lb, Ht. 5'5'', normal ROM both hips, no swelling or redness.*

A—*Possible osteoarthritis; R/O rheumatoid arthritis*

P—*blood work—sed rate, rheumatoid factor, x ray L hip PA and lateral; ibuprofen 600 mg t.i.d po; recheck 2 months.*

SOAP Note Example 2:

Patient Name: Lisa Brown

DOB: 2/3/1960

Record No. B-583uw809

Date: 10/19/2001

S—Pt. here for weekly BP check, no complaints. NKDA, NKA.

O—BP 142/88; Atenolol 50 mg daily

A—hypertension controlled

P—Continue Atenolol; RTO 6 months

Abbreviations keys:
WT = weight
HT = height
BP = blood pressure
Pt = patient
RTO = Return to office
ROM = range of motion
R/O = rule out
PA= posterior/anterior
NKDA = No known drug allergies
NKA = No known allergies

Where **S** stands for Subjective, **O** stands for Objective, **A** stands for Assessment and **P** stands for Plan. One must notice that normally in a real SOAP note the descriptions are abbreviated. Some of the domain concepts came from this notion of SOAP notes. Our system to be developed would be based on SOAP notes which are used in many medical centres and offices, but in our case we would use codes instead of abbreviations to save space in the Health Card, and the client systems would have references to those codes.

Having the domain concepts, we proceed with the formulation of Use Cases. These will model the functionalities of the future system.

4.2. Use Cases

In this section we describe how we get the Use Cases and we present them with their respective diagrams and the textual scenarios for each one of them. In the beginning, after having a general idea of the Health appointments domain, we indentified twenty relevant use cases for our system. Those are shown in the list below. Through these, we made detailed ones that helped us to design the structure model of our system in combination with the functional requirements. Also, at this phase the Use Cases served us as a base to make the informal functional requirements.

These are the Health Card system use cases obtained at a first stage:

1. Inserting personal code
2. Modifying personal code
3. Inserting personal data
4. Modifying personal data
5. Viewing personal data
6. Searching a doctor

7. Making an appointment
8. Modifying an appointment
9. Cancelling an appointment
10. Viewing medical history
11. Inserting a medical history entry
12. Modifying a medical history entry
13. Removing a medical history entry
14. Checking-in appointment
15. Viewing appointments
16. Setting up database information
17. Renewing medical prescription
18. Viewing medical prescriptions
19. Accepting/Denying medical prescriptions
20. Viewing patients request

They were written in a general from the domain concepts and from talking with the stakeholders. Their description is:

1. **Inserting personal code**: When inserted a health card into a card reader, the pin-code is asked. The card owner should then insert the pin-code digits.

2. **Modifying personal code**: The card owner can modify the pin-code of his card. But first, it will be asked his pin-code before he can proceed to change it.

3. **Inserting personal data**: The required patient's personal data (e.g. name, gender, birth date, blood type, ID number or passport ID, his birthplace and nationality) are included when the card is created by the Card Issuer. Personal optional data (e.g. contact, an address, social security number, etc) are inserted by either, the card issuer or the owner.

4. **Modifying personal data**: The card owner can change his personal optional data (e.g. contact, an address, social security number, etc).

5. **Viewing personal data**: From this use case it is possible to view the card owner's personal data, if the pin-code was inserted correctly.

6. **Searching a doctor**: When the card owner inserts his health card into a patient terminal, he can search for a doctor. The main objective here is to check time availability of the searched doctor and then schedule an appointment.

7. **Making an appointment**: It should be possible for a card owner to schedule an appointment with his Health card. A doctor can also schedule an appointment for his patient, if necessary, being the information about the appointment schedule in the owner's card. When making an appointment, date, place (insert automatic by the system), and doctor (or type of appointment) are stored in the card.

8. **Modifying an appointment**: The card owner can change an appointment with his card. He can choose, from the schedule appointments, which one he wants to change. The system will show an alternative of the doctor's schedule availability, and the card owner can choose to change the appointment date, or doctor/type of appointment.

9. **Cancelling an appointment**: The card owner can cancel an appointment with his card by inserting it into a patient terminal.

10. **Viewing medical history**: Medical staff can see medical history from the patient. Visualization of medical history includes the visualization of health problems, prescriptions / past and recent treatment plans, allergies, vaccinations and medicine prescription.

11. **Inserting a medical history entry**: Only medical staff can insert into the patient card an entry about the patient medical history. This includes inserting a new allergy to the list of the patient's allergies, information about some health problem and plan of treatments, or adding information in the vaccination list. When some entry is added to the medical history, it is the responsibility of the system for the association of the doctor who made the entry and the date of that entry.

12. **Modifying a medical history entry**: Changes in an entry of the patient's medical history only can be made by medical stuff. Data changed has associated the date of modify and the doctor's name that changed it.

13. **Removing a medical history entry**: Removing an entry from the patient's medical history only can be made by medical stuff. Removed data will be temporally removed, being possible to go back during the medical appointment. The doctor will be responsible for data removed.

14. **Checking-in appointment**: When a patient arrives to an appointment, he can use the card to do the check-in. So the medical staff can know that the patient already arrived.

15. **Viewing appointments**: The card owner can visualize his appointments when he inserts his card into a patient terminal. It will be given the possibility to change or cancel a medical appointment.

16. **Setting up database information**: The system administrator can insert, modify or erase data from the central database of the system. That central database will have doctor's availability schedules and information about the doctors, information about the medical centre, list of allergies and vaccines.

17. **Renewing medical prescription**: Patients can renew a medical prescription for a medicine without having to schedule an appointment. They could choose the prescription that they want to renew.

18. **Viewing medical prescriptions**: The patient and the medical staff can visualize all medical prescription as list of medicines prescribed by the doctors. It is possible to patient renew a prescription that he is seeing when inserted a card into a patient terminal.

19. **Accepting/Denying medical prescriptions**: The doctor can access the requests of prescriptions to renew, where they can accept or reject. Later, when the patient inserts the card into a client terminal will know the status of the order.

20. **Viewing patient's requests**: The doctor can visualize a list medical prescription requests to renew.

These use cases suggests that user roles must be implemented so that authorized users can perform authorized operations. Each user role eventually characterizes which operations can be done by a user performing the respective role.

4.2.1. System Actors

After having identified the main use cases, we identified the possible system users (actors in the system). These are the system actors that we identified:

- Doctor / Medical staff

- Patient / Card owner
- Card Issuer
- Administrator

System Actors descriptions

Doctor / Medical staff

From the system's point of view, this is the user who consults the patient's medical and personal information and who is responsible for adding, modifying or removing data from the medical history.

Patient / Card owner

In this context, the patient is the card owner, and the entire card's data refers directly or indirectly to him. The patient will interact directly with the system for scheduling appointments, or for consulting his scheduled appointments, allergies, vaccinations or medical prescriptions.

Card Issuer

The card issuer is the person responsible for creating new cards. He's responsible for inserting into a card the patient's data.

Administrator

The system administrator is the responsible for the updating of the central database in the medical centre. He updates doctor's schedules, lists of known allergies and vaccines, and other general medical centres information.

4.2.2. Use Case models

From the identified use cases and system actors we wrote Use Case models. They relate use cases with the possible system users. These Use Case models give an insight of the system's functionalities and usage. Later, they will support the design of the system's structure, i.e., through the Use Case models we can obtain the classes and their methods that probably are needed. The following Figure 8 ilustrates a main use cases diagram of the Health Card system containning the initial use cases.

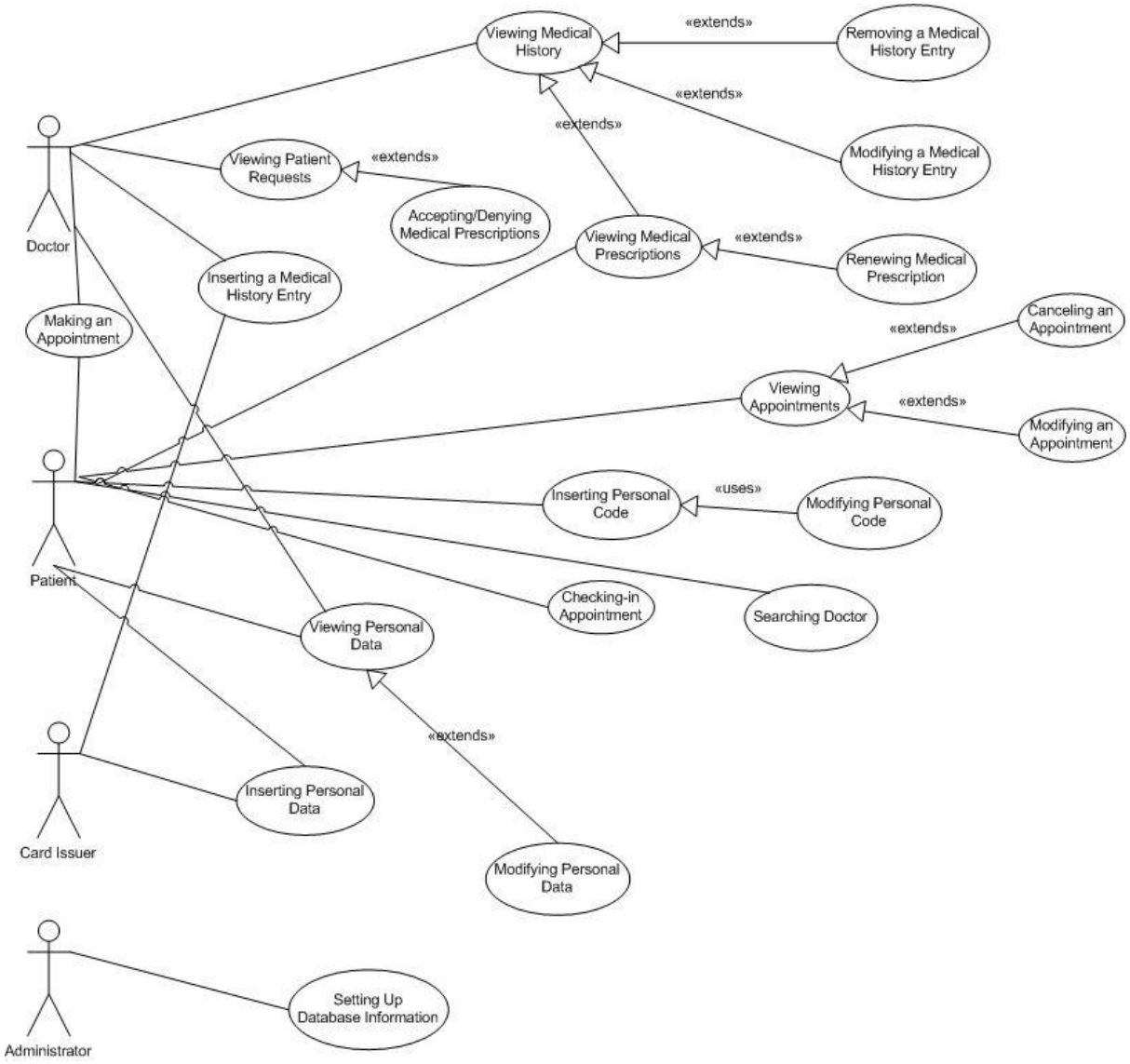

Figure 8. Main Use Cases diagram

From the main use cases diagram, we divided the use cases into four sub-diagrams: - the medical history, the personal data, the appointments scheduling and the administration. In this way, we have detailed and organized use cases, where it is visible which ones are from the card side and from the external side of the Health Card system. These can be consulted in the *Annex* of this document at ***1. Annex: Use Cases Diagrams***. As an example, Figure 9 presents the Use Cases sub-diagram for *Appointments*.

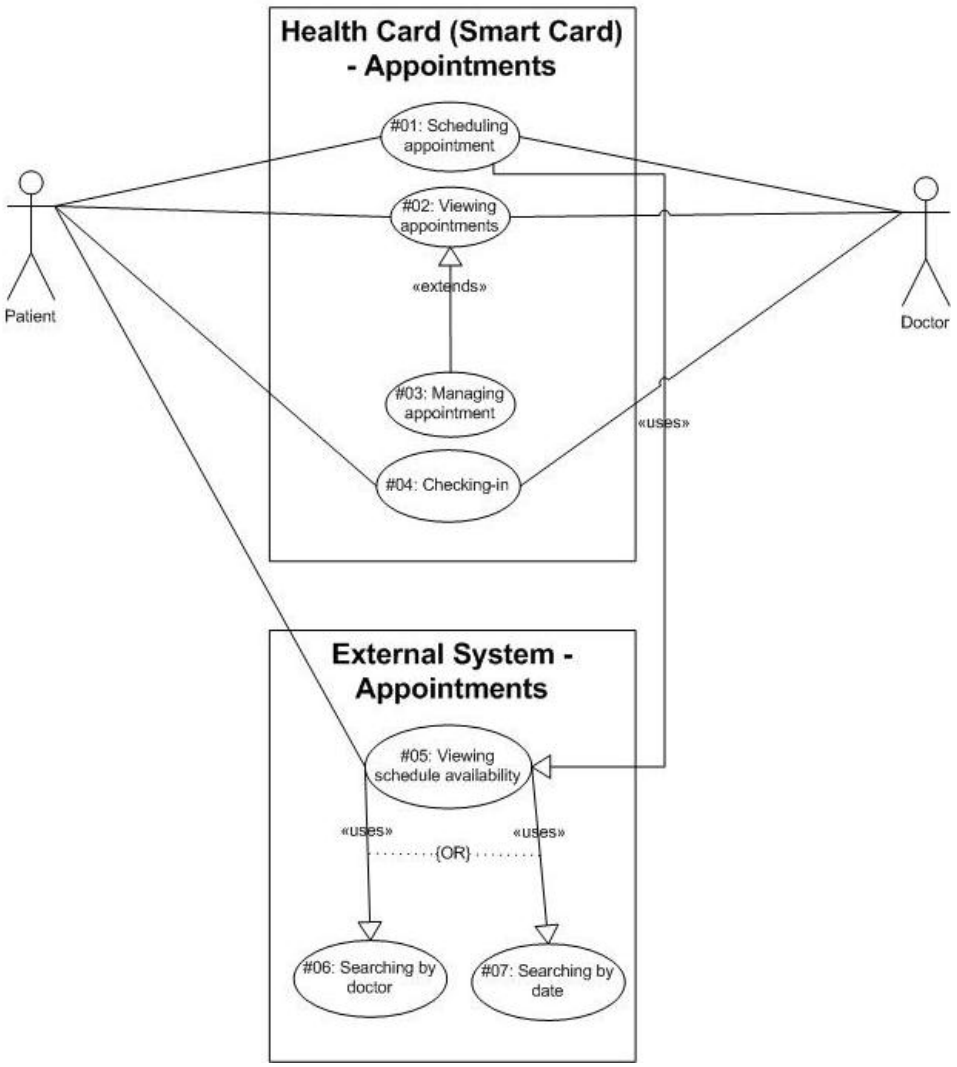

Figure 9. Use Cases sub-diagram of Appointments

Also, for each of the use case presented in the Use Cases diagrams (i.e., the use cases sub-diagrams seen in the annex) we did textual specifications, or scenarios. The purpose of writing textual specifications of the use cases is to clarify in natural language the interactions between users and the system. These Use Case Textual Specifications can be seen in the *Annex* at ***2. Annex: Use Case Textual Specifications***. As an example we present in the following Table 5, a textual specification for the Use Case *"#03: Managing Appointments"* from *Appointments*.

Table 5. Textual specification of "Managing Appointments" from Appointments' Use Cases

Name: *Managing Appointments*	ID: *C03*
Main Scenario **This use case extends the <u>Viewing Appointments</u> use case. In this use case the user chooses to manage an appointment after selecting it. Then, the system offers ways of modifying and cancelling the selected appointment. The user chooses one of the options.**	
Alternative Scenario 1 (the user chooses to modify) **The user chooses to modify the scheduled appointment after selecting the schedule entry. The system then offers a way of modifying the date and the doctor or type of appointment by showing to the user the available schedules. The user then chooses the pretended date, doctor and appointment type and confirms. The**	

system stores the changes into the card.
Alternative Scenario 2 (the user chooses to cancel) **The user chooses to cancel a scheduled appointment after selecting the schedule entry. The system then asks for the confirmation. The user confirms it and then the system removes the scheduled appointment.**

Like the given examples, for each part of the system, we made the Use Cases diagrams and its textual specifications.

Medical History

This sub-diagram illustrates use cases that are related with the patient's allergies, chronic conditions, health problems, diagnostics, treatments, medical prescriptions. – *see **Annex: Use Cases Diagrams – Figure i*** - It includes the following use cases:

- A01: Viewing Medical History
- A02: Viewing Chronic Condition
- A03: Managing Chronic Condition
- A04: Viewing Vaccinations
- A05: Managing Vaccinations
- A06: Viewing Allergies
- A07: Managing Allergies
- A08: Viewing Family History
- A09: Managing Family History
- A10: Viewing Health Problems
- A11: Managing Health Problems
- A12: Viewing Current and Past Medication
- A13: Viewing Diagnosis Information
- A14: Managing Diagnosis Information
- A15: Viewing Treatment Plan
- A16: Viewing Medical Recommendations
- A17: Managing Medical Recommendation
- A18: Viewing Medical Prescription
- A19: Requesting Prescriptions Renewal
- A20: Managing Medicines

The textual specification of the medical history use cases can be seen in the *Annex* at ***2. Annex: Use Cases Textual Specification*** under the *A: Medical History Use Cases* section.

Personal Data

This sub-diagram illustrates use cases that are related with the patient's personal information, like managing required or optional personal details. – *see **Annex: Use Cases Diagrams – Figure ii*** - It includes the following use cases:

- B01: Viewing Personal Data
- B04: Managing Optional Details
- B05: Managing Required Details

The textual specification of the personal data use cases can be seen in the *Annex* at ***2. Annex: Use Cases Textual Specification*** under the *B: Personal Data Use Cases* section.

Appointments

This sub-diagram illustrates use cases that are related with appointments scheduling and management. – *see **Annex: Use Cases Diagrams – Figure iii*** - It includes the following use cases:

- C01: Scheduling Appointment
- C02: Viewing Appointments

- C03: Managing Appointments
- C04: Checking-in
- C05: Viewing Schedule Availability
- C06: Searching by Doctor
- C07: Searching by Date

The textual specification of the appointments use cases can be seen in the *Annex* at ***2. Annex: Use Cases Textual Specification*** under the *C: Appointments Use Cases* section.

Administration

This sub-diagram illustrates use cases that are related with the external system administration, like updating the system's central database (lists of allergies, vaccines, doctor's registries, schedules, etc). As our thesis work only focuses on the card side of the whole system, these use cases weren't considered as a priority for us. – *see* ***Annex: Use Cases Diagrams – Figure iv*** - It includes the following use cases:

- D01: Updating Allergies
- D02: Updating Vaccines
- D03: Setting medical centre information
- D04: Managing doctor's schedules

The textual specifications of the administration use cases haven't been written, as these use cases aren't a priority to the purpose of this work.

Having the Use Cases and its Textual Specifications, next we write the informal functional requirements. These basically are rules and specifications that the system to develop must hold.

4.3. Informal Functional Requirements

From the discussions with stakeholders and from reading documents about medical appointments, the functional requirements were described in an informal way. These informal functional requirements are listed at ***3. Annex: Informal Functional Requirements***. As an example, and continuing using the *Appointments* part of our work, we describe some of its informal functional requirements (FR):

Scheduling

FR107. A scheduled appointment must contain data of a place (local), a date/time and a doctor and an appointment type.

FR108. When adding a new scheduled appointment information it is required to specify its date, hour, local and also the doctor and the type of appointment.

FR109. If the limit of possible appointment insertions is achieved, the system must not allow insertions of appointments into the card.

FR121. The system must not allow modifying the data (i.e. date and time, local, doctor, type of appointment) of an appointment after checking in to that appointment.

FR123. It must not be possible to overlap schedules in the same date and hour.

Checking-in

FR125. When a check-in of a scheduled appointment is not made in a period of 1 day, that scheduled appointment must be erased from the card.

Effective Appointments

FR127. To an effective appointment there must be related diagnostic inserted by the appointment doctor.

FR128. An appointment has an effective status when it has medical information associated.

FR129. An appointment cannot be deleted from the card if its status is effective.

Some of the informal functional requirements will turn into semi-formal requirements, others will be used to write class and system invariants.

4.4. From Informal Functional Requirements to Formal Specifications

This step occurs during the first stage of the *formal specifications pseudo-phase*, where from the informal functional requirements – see ***3. Annex: 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 form, suitable to be used into JML specifications. These semi-formal requirements are to be 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.

4.4.1. Semi-Formal 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 3.2.1.3 for a description of system invariants).

As an example of how this semi-formal form is attained, the informal functional requirement FR108 is transformed into if < adding a new scheduled appointment > then < it must be inserted a date, an hour, a place and doctor or a type of appointment >. We show the complete list of semi-formal requirements obtained from the informal ones in the annex – see ***4. Annex: Semi-Formal Requirements***. As an example, the following Table 6 describes this

process between informal functional requirements and semi-formal requirements for the *Appointments*.

Table 6. Semi-Formal Requirements from Appointments informal functional requirements

ID	From Requirement	Semi-Formal Requirement
SFR108	FR108	If adding a new scheduled appointment, then it must be inserted a date, an hour, a place and doctor and a type of appointment.
SFR109	FR109	If adding a new scheduled appointment and the limit has been achieved, then the system must do no changes.
SFR121	FR121	If an appointment is already checked-in, then the appointment header cannot be modified (date and time, local, doctor, type of appointment).
SFR124	FR124	If an appointment is checked-in, then that appointment must turn into a checked-in state.
SFR129	FR129	If removing an appointment and its status is effective, then the system must do no changes.

For each semi-formal requirement we gave an identification (i.e., ID) like we did for the informal functional requirements. This was done for the sake of requirements traceability, i.e., to easily trace them in later stages of development.

4.4.2. 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. All this class invariants are identified with a reference code in the format of CIxxx. Later, these class invariants are to be ported into JML invariants (see Section 5.2). We show the complete list of the Health Card's class invariants obtained from the informal functional requirements in the annex – see ***5. Annex: Class Invariants***. Continuing with the Appointments example, we describe some of the informal functional requirements identified as class invariants:

FR108. All scheduled appointment must contain data of a place (local), a date/time and a doctor or an appointment type.

- **CI108.** For all object ***a*** of type appointment, such that local(***a***) not equal to null, date(***a***) not equal to null, hour(***a***) not equal to null and (doctor (***a***) not equal to null and type(***a***) not equal to null).

FR124. It must not be possible to overlap schedules in the same date and hour.

- **CI124.** For all objects ***a1*** and ***a2*** of type appointment, if ***a1*** not equal to ***a2*** then (date(***a1***) not equal to date(***a2***) and hour(***a1***) not equal to hour(***a2***)).

FR126. When a check-in of a scheduled appointment is not made in a period of 1 day, that scheduled appointment must be erased from the card.

- **CI126.** For all objects ***a*** of type appointment, if status(**a**) less than CHECK_IN then timeDifference(schedule_time(***a***), actual_time) must be less than 24 hours.

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 **FR108** as a class invariant because it describes that an appointment must have the attributes of a date, hour, place, doctor and appointment type, so eventually the Java class of an appointment must declare those attributes and the constructor must initialize them. As for the **FR124**, there mustn't ever be appointments with the same date and hour, so this is obviously a limitation of the appointments properties. The last informal requirement from the example, **FR126**, is considered as a class invariants because it restricts the non existence of scheduled appointment with 1 day after the date and hour of their schedule. After 1 day they must be checked-in or deleted.

4.4.3. 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 5.2). Considering the following informal functional requirement from Medicines:

- **FR93.** *The prescription date of a medicine must be bigger than or equal to the date of the appointment in which the medicine was prescribed.*

We can see that the above 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. All this system invariants are identified with a reference code in the format of SIxxx.

- **SI93.** *For all object **m** of type medicine and for all object **a** of type appointment such that appointment(**m**) equals to **a** and date(**m**) is bigger than or equal to date(**a**).*

In the Annex of this document we present the complete list of the Health Card's system invariants obtained from the informal functional requirements in the annex – see ***6. Annex: System Invariants***.

4.5. 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 (and its 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 about 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, besides having its implementations reusable, the JML specifications also can be easily reused. In our case, our structure was 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.

4.5.1. Structure Modelling

Having in mind that our application is based on the Java Card Remote Method Invocation (JCRMI) model (see Section 2.2.3), our structure also had to have architecture in conformance. Since our work was centred on the use of the Java Card language, we gave more importance to the design of the server side of the system, i.e., the card side. The design that we describe in this section is all about the card side, the side of the system that will contain the JML specifications that will provide support to, among other things[3], the development of the client applications, external to the card.

Each previously mentioned module in the beginning of this Design section, as java packages, contains a remote java interface in which are described constant values and signatures of the methods that will be called remotely by external clients. Through these remote java interfaces, the client applications can invoke remote methods, i.e., functions served by the card applications. Also, it's at the remote java interfaces where most of the JML specifications are written to define the behaviour of the remote methods (i.e., preconditions and postconditions) as well as its invariants. These JML specifications, at the remote java interfaces (i.e., at the server side, the card side), support the coding of the respective implementation classes (i.e., concrete classes) including its method procedures as well as to inform the external applications developers (i.e., the client side, the external side) of the conditions of each remote method that can be called.

Applying the JCRMI model (see Section 2.2.3), one produces remote java interfaces that are a convenient point where one can write JML specifications for the methods that can be called by external clients. These JML specified remote java interfaces are shared with the client side and while developing the client side applications, the developers can program the client side supported by the specifications from those shared java interfaces. When writing a procedure that makes a call to a certain JML specified remote method, the developers are aware of the preconditions and postconditions of those remote methods as well as the its class invariants. The developer is aware that while writing the client applications and making a remote method call, he must respect the method precondition, and by doing so, he may safely assume the method postcondition. This process of writing code follows the design by contract methodology, in which method's calls at the client side are programmed according to the specifications of these methods at the server side, i.e., a contract between the client and server sides is enforced. Having described the structural implications brought by the relations between the server side and the client side when using a JCRMI model and JML specifications, we will next demonstrate how we can go from use cases and informal functional requirements obtained at the analysis requirement, to a model of the structure of the system to be implemented.

As said before, in the beginning of the Design section, the informal functional requirements and use cases obtained at the requirement analysis were organized into modules. Based on

[3] JML specifications support the implementation of the respective specified methods, its documentation and communication between developers and consequentially the support of the development of client procedures that contains calls to the specified methods.

the use cases, the possible operations (i.e., in the form of methods signatures) were formulated at each different class for each module, and method's parameters were written from the informal functional requirements descriptions. Starting, as an example, from the requirements related to the *Appointments* of our *Health Card* system, we'll describe the process from the analysis requirements to a fully structure model at the end of the Design phase.

Based on the following use cases from *Appointments* (see Figure 10), one can extract certain functionalities and classes.

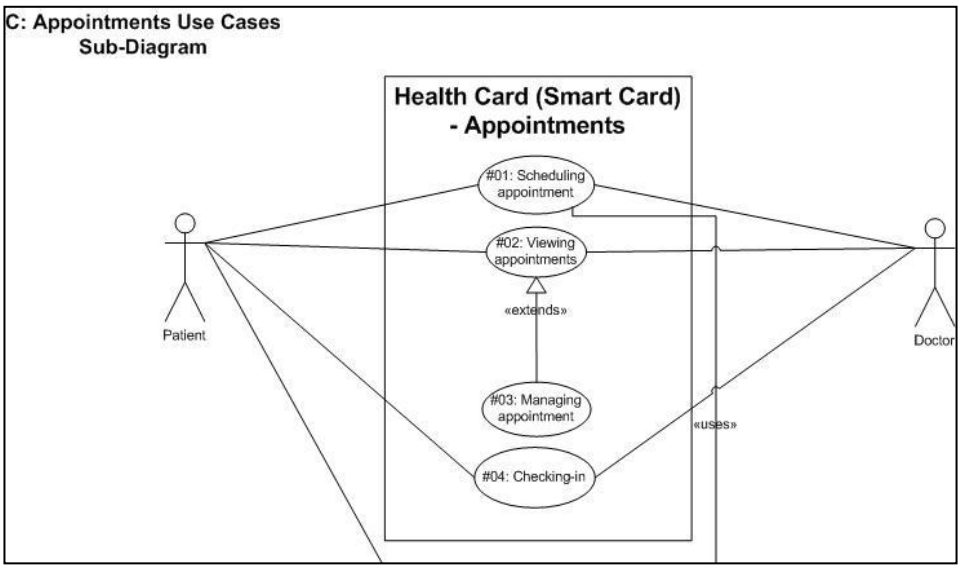

Figure 10. C: Appointments Use Cases Sub-Diagram

Our objective at this phase is to have a structure model of the system and for accomplish that we can use *class diagrams*[4] from the *UML*. The first thing to do is to identify the possible classes and write them from a particular set of related use cases, in this example, the Appointments class is identified. Next, we can start formulating methods (i.e., operations) for that class. Of course, we'll also need to define parameters for certain methods to receive (i.e., parameters for the mutation methods[5]). The methods' parameters can be extracted from the informal functional requirements descriptions. Having a look back at Figure 10, from the use case *Scheduling appointment (C01)*, one can assume that the class *Appointments* needs the method *addAppointment*, and being a mutation method, to add an appointment we need to pass certain values, so we need the informal functional requirements to get details on how and what is needed to perform that operation. Looking at the following informal functional requirement:

- *FR108. A scheduled appointment must contain data of a place (local), a date/time and a doctor and an appointment type.*

We can see that for the creation of a new appointment we need to pass values of a local, a date, an hour, a doctor and an appointment type. So the signature for the method of adding an appointment is like:

[4] Class diagrams are structure diagrams of the UML 2 that show the static structure of a system being modeled. They focus on the elements of a system, showing the relationships between them and even their internal structure. [34]

[5] Mutation methods are methods that do something to an object, and typically receive parameters and do not return anything. For example: `void setDoctor(byte[] doctor)` [35]

- *addAppointment(byte[] date, byte[] hour, byte[] local, byte[] doctor, byte type)*

And for the management of those attributes we define query[6] and mutation methods for each one of them (i.e., *getAppointmentDate, setAppointmentDate, getAppointmentHour, setAppointmentHour*, etc.). For the other use cases from *Appointments*, the process of creating methods for the class *Appointments* is similar:

- *Viewing appointments (C02)* – This use case implies that we should use query methods for it. So the following query methods can be used to support the information viewing about appointments:

    - *getAppointmentDate(short position)*
    - *getAppointmentHour(short position)*
    - *getAppointmentLocal(short position)*
    - *getAppointmentDoctor(short position)*

- *Managing appointment (C03)* – This use case includes modifying information about an appointment and removing it. That implies the definition of mutation methods. The following mutation methods can be used to support the modification or removal of appointments:

    - *setAppointmentDate(short position, byte[] date)*
    - *setAppointmentHour(short position, byte[] hour)*
    - *setAppointmentLocal(short position, byte[] local)*
    - *setAppointmentDoctor(short position, byte[] doctor)*
    - *removeAppointment(short position)*

Note that the parameter position is the value of the *Appointment* object's position in the data structure containing *Appointment* objects.

For the last use case, *Checking-in (C04),* and from the informal requirement about checking-in an appointment: – F*R125. When a check-in of an appointment is made, that scheduled appointment must be turned into a checked-in appointment* – we've created a concept of appointment status. This status is represented as an attribute of an *Appointment* to inform about the status of it, for example, status *scheduled appointment* or *checked-in* or *done*. The query and mutation methods for that attribute status we also written (i.e., respectively *getAppointmentStatus* and *setAppointmentStatus*).

The following Figure 11 illustrates the remote java interface for the management of Appointments, obtained from the previous process.

[6] Query methods are methods through which we can request information, but it doesn't change the state of an object. For example: *byte[] getDoctor()* [35]

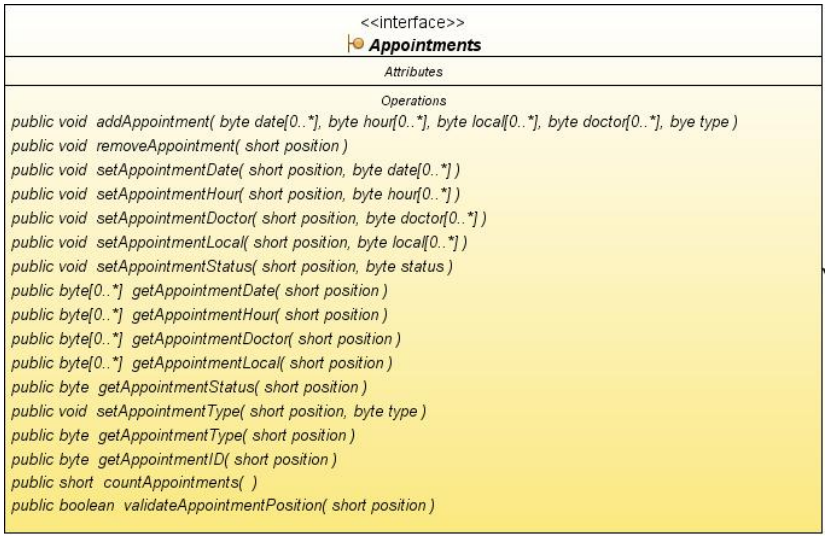

Figure 11. Model Structure of Appointments remote java interface

This *Appointments* interface is the provider of the card remote related with appointments scheduling management, i.e., this interface is shared between the client and server applications where the client can access them remotely. Also, it is here that JML specifications for the appointments scheduling management will be written at the Implementation phase. One must note that from Figure 11, that we also created a *getAppointmentID* method. This method is necessary because each appointment managed by the *Appointments* must be unique, i.e., each one has a different internal identification having the purpose of facilitating their relations with other modules' objects (i.e., diagnostics, treatments, medicines). Each *Appointment* object instance managed by the *Appointments* class is also represented by a class itself. The classes representing single appointment objects are the interface *Appointment* and its implementation class *Appointment_Impl* as seen in Figure 12. Each Appointment object holds information about a real appointment and has methods to manage its attributes.

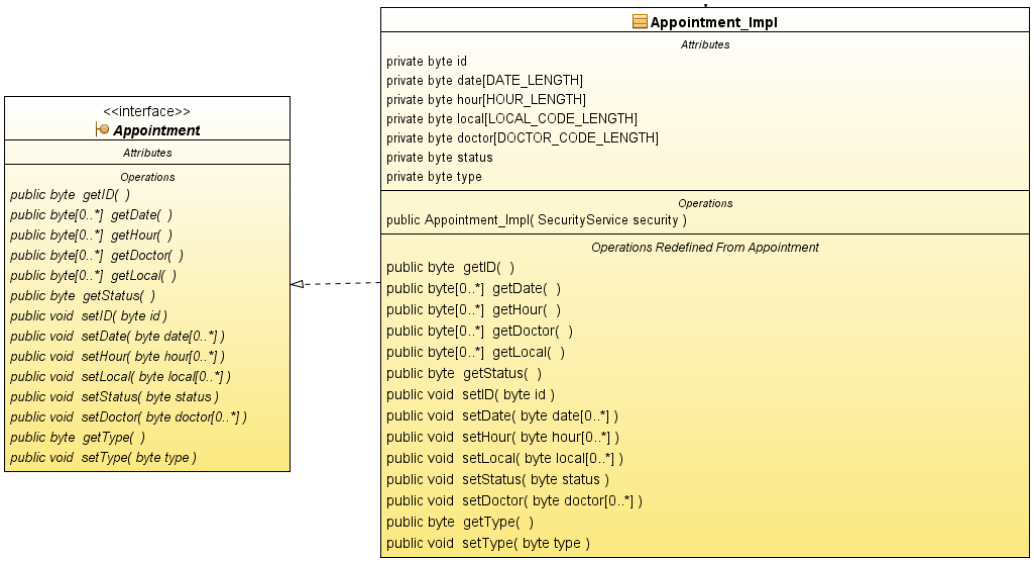

Figure 12. Model Structure of Appointment and its implementation class

The *Appointment* interface isn't remote like the *Appointments* interface, so an *Appointment* object isn't accessed directly by the client applications but it is managed through the *Appointments* interface and its implementation class *Appointments_Impl*, i.e., through a data structure in *Appointments* containing *Appointment* objects instances. The Figure 13 shows the

dependency between *Appointments* and *Appointment*, where *Appointments_Impl* is the implementation class of the interface *Appointments* and *Appointment_Impl* is the implementation class of the interface *Appointment*.

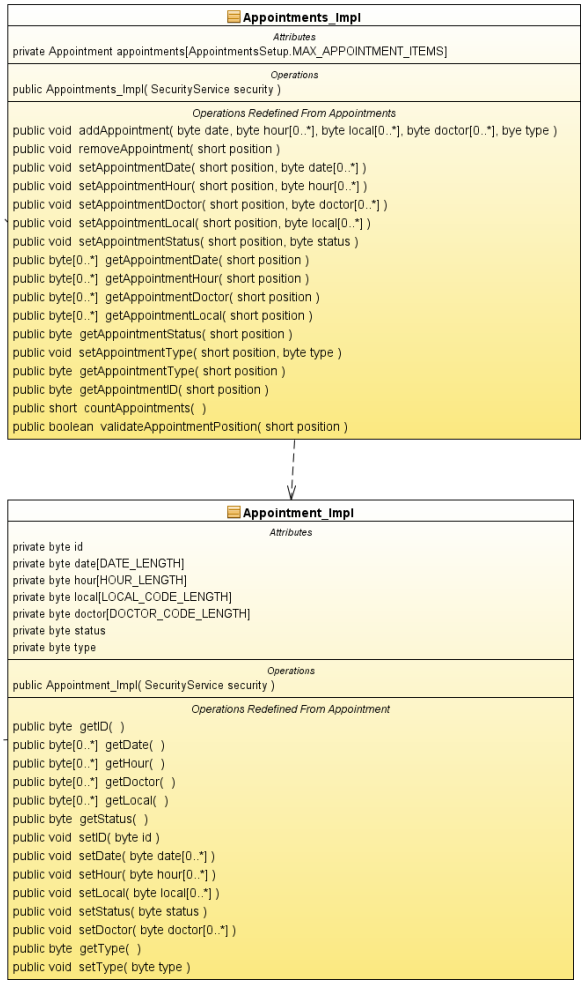

Figure 13. Model Structure of the Appointments and Appointment implementation classes

The rest of the modules were design like the *Appointments* exemplified in this section, and another module was created to manage the rest of the card application parts, this one was named *CardServices*. Basically *CardServices* is dependent of the *Personal, Allergies, Vaccines, Appointments, Diagnostics, Treatments* and *Medicines* modules and have methods to get the their instances, so the client applications can make calls to the remote interfaces of each one of them. Figure 14 illustrates the *CardServices* module and its dependencies. One must notice that the *CardServices* module acts like a broker, where the client applications can access other modules through it. This decision of creating a broker module for the card side came when we implemented our first prototype. We had some difficulty in getting the other module interfaces while calling them on the applet of the *Health Card.*

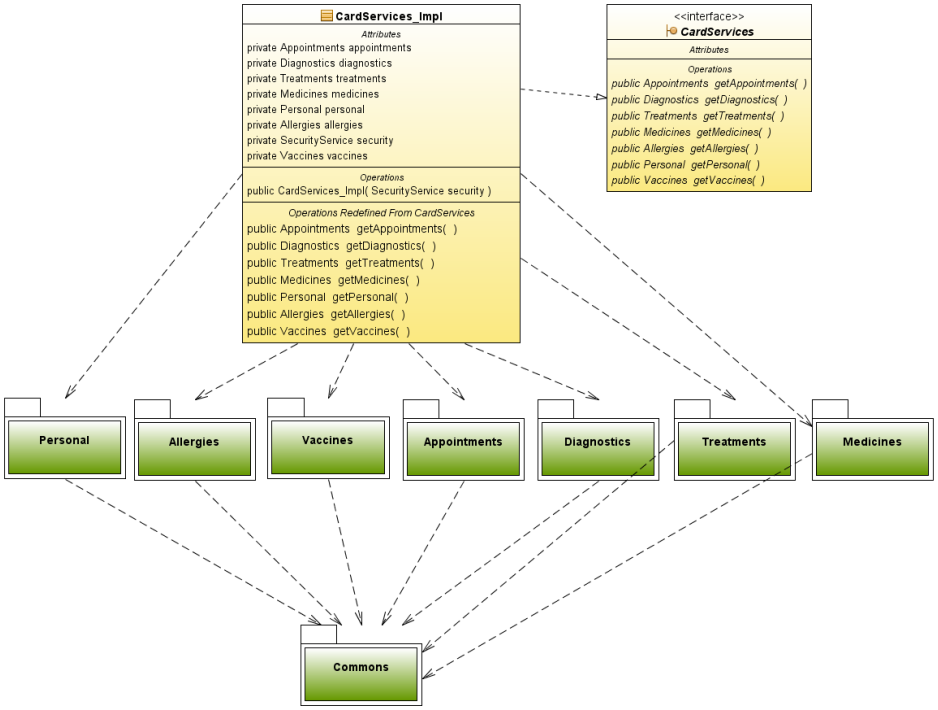

Figure 14. Structure Model of the Health Card

Concluding this section, we've described the process at the Design phase that goes from the initial requirements to a model structure of the system, by giving an example of the Appointments module. For the complete model structure of the card application the Annex document should be consulted – see ***7. Annex: Class Diagrams***.

5. 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 supported by 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[7] can be generated using an automatic code generation tool or manually. In our work we have used *NetBeans IDE* [30] to design the system structure model and to automatically generate the java code skeletons from it. This automatic code generation is not complete and sometimes is blind and produces mistakes, so one has to correct the code manually. For example, all array types written in the structure model were wrongly generated to code, so we had to correct those data types by hand. Having generated those java documents, which are still without implementations inside the methods, and taking a look to the method signatures at the interfaces, we begin to associate with them each semi-formal requirement defined from the informal functional requirements (see section 4.4), and also each invariant is associated to the respective interface. Those JML specifications are written mostly at the remote java interfaces (due to the JCRMI employment) and later one can develop the method implementations in the respective concrete classes in many ways as long

[7] 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.

as the specifications in the interfaces are respected. First we describe the process of writing abstract variables (i.e., model fields) to represent the concrete class attributes at the specifications level, next we describe the formalization process done at the formal specifications pseudo-phase which from the semi-formal requirements we get JML specifications of method's preconditions and postconditions and also the specification of class invariants at the interfaces, and finally we describe the advantages of implementing a system with the support of JML specifications and problems encountered while implementing it as well as the solutions found.

5.1. Writing the Abstract Variables

When writing the JML specifications, one is most likely to refer to implementation attributes (i.e., concrete class properties), but as we have seen in this document most of the specifications are written in the Java interfaces (or abstract classes). However, in Java interfaces one can declare constant values but not non-static variables, so the use of abstract variables (or abstract variables) brings an advantage in this case because one can use them to represent 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. 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. For example, in a concrete class we could change the concrete variable being represented by using represents and still maintaining without changes the specifications at the interfaces:

Before modifying id attribute's name

```
private /*@ spec_public @*/ byte id; //@ in id_model;
/*@ public represents
  @     id_model <- id;
  @*/
```

After modifying id attribute's name to appointmentID

```
private /*@ spec_public @*/ byte appointmentID; //@ in id_model;
/*@ public represents
  @     id_model <- appointmentID;
  @*/
```

By this, the specifications would stay exactly the same if the concrete variable is modified.

We start declaring abstract variables in JML by using the keyword `model`, or `ghost` for ghost variables (which can't be represented and only exist in JML specifications). 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* 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[22]. For details about JML abstract data types, see Section 3.3.2.

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 connected with real variables or expressions by using a mechanism of representation through the JML *represents* clause (see Section 3.3.1. 2). Continuing from the structure model example given in the Design phase (see Section 4.5), we identified the possible abstract variables by analysing the design structure, specially, the attributes for each class. Having analyzed the structure model of the *Appointments* module, these were the attributes that would contain information about an *Appointment* object: - *appointment identification*, *date, hour, local, doctor, appointment type* and *appointment status* – and they were modelled in the specifications respectively like: - *id_model, date_model, hour_model, local_model, doctor_model, type_model* and *status_model*. Also, looking to the structure model of the *Appointments* module, more precisely at *Appointments* interface and its implementation class *Appointments_Impl*, we noticed the necessity of having an abstract variable representing a structure that would contain *Appointment* object instances. We named it *appointments_model* and for the sake of data abstraction, we declared it as a JML abstract data type *JMLObjectSequence* instead of *Appointment* array. The *JMLObjectSequence* is one of the object collections type defined for JML, where it treats its elements as object references (i.e., addresses) rather than the values of those objects. For example, when inserting an object in one *JMLObjectSequence*, that object isn't cloned but its reference is stored. The choice to abstract the structure holding *Appointment* object instances was to abstract the way how a developer could program a data structure to store the appointments, and by a data structure for holding objects can be implemented in many ways (see Section 3.3.2). For the same reason we used the JML abstract data type *JMLValueSequence* for declaring some other abstract variables, which represents a complex data structure containing values like primitive byte values (the mainly use in the HealthCard specifications). The following piece of code (see Code 9) describes the declaration of abstract variables in the *Common*[8] and *Appointment* interfaces.

```
public interface Common {
        …
        //@ public model instance JMLValueSequence date_model;
        …
        //@ public model instance JMLValueSequence hour_model;
        …
        //@ public model instance JMLValueSequence local_model;
        …
        //@ public model instance JMLValueSequence doctor_model;
        …
        …
}

public interface Appointment extends Common{
        …
        //@ public model instance byte id_model;
        //@ public model instance byte status_model;
        //@ public model instance byte type_model;
        …
}
```

Code 9. Common and Appointment interfaces abstract variables

In Code 9, we have abstract variables declared in *Common* interface which is extended by *Appointment* interface. Hence, all JML specifications from *Common* are inherited by *Appointment*, including these abstract variables. As abstract variables are inherited from *Common*, they can be referred in *Appointment* specifications. In our work, we chose to declare abstract variables in a super interface because abstract variables can be reusable (i.e., they can

[8] Common is an interface, from our work, with concrete constant values, JML model variables and JML specifications used by multiple classes that extend this one. This includes specifications about the correct format of a date, an hour, a doctor code and a medical center's local code.

represent multiple concrete variables) and for the fact that we needed those abstract variables in specifications of multiple classes, as well as invariants related with those abstract variables which also were specified in *Commons* (invariants are also inherited, see Section 3.2.1.4). For example, at the *Common* interface we have the abstract variable *date_model* and it can be reused for specifying a birthday date, an allergy identification date or an appointment date. The following piece of code (see Code 10), describes the connection between concrete variables and abstract variables made at the implementation classes, i.e., the representation of concrete variables by abstract variables at the classes implementing the specified interfaces.

```
public class Appointment_Impl implements Appointment {

        ...

        private /*@ spec_public @*/ byte id; //@ in id_model;
        /*@ public represents
          @     id_model <- id;
          @*/
        private /*@ spec_public @*/ byte[] date; //@ in date_model;
        /*@ public represents
          @     date_model <- toJMLValueSequence(date);
          @*/

        private /*@ spec_public @*/ byte[] hour; //@ in hour_model;
        /*@ public represents
          @     hour_model <- toJMLValueSequence(hour);
          @*/

        private /*@ spec_public @*/ byte[] local; //@ in local_model;
        /*@ public represents
          @     local_model <- toJMLValueSequence(local);
          @*/

        private /*@ spec_public @*/ byte[] doctor; //@ in doctor_model;
        /*@ public represents
          @     doctor_model <- toJMLValueSequence(doctor);
          @*/

        private /*@ spec_public @*/ byte type; //@ in type_model;
        /*@ public represents
          @     type_model <- type;
          @*/


        ...

}
```

Code 10. Concrete attribute fields of an Appointment and its model representations

In Code 10, each abstract variable represents a concrete variable or an expression. This representation is made through the JML specification using the clause *represents*. Looking at the JML specifications we can see an expression *toJMLValueSequence* being used at the *represents* clauses. This expression is a pure helper method, at the JML specifications level, made to convert an array of bytes into a *JMLValueSequence* (a pure method doesn't have side effects on an implementation runtime). This is how we can make a representation of a concrete array of bytes by a *JMLValueSequence* abstract variable. The following Code 11 describes the specification helper method *toJMLValueSequence*.

```
public interface Common {

        ...
        /*@ public static model pure JMLValueSequence toJMLValueSequence(byte[] b){
          @             JMLValueSequence _m = new JMLValueSequence();
          @             for(int i = 0; i < b.length; i++)
```

```
          @                    _m = _m.insertBack(new JMLByte(b[i]));
          @            return _m;
          @ }
          @*/
        ...
}
```

Code 11. Specification helper method toJMLValueSequence

As seen in the previous code, every byte of the array is casted into a *JMLByte*, an abstract JML type reflecting the Java type *Byte*, and inserted into a temporary object _m of the abstract type *JMLValueSequence*. This method is written as a JML model method in the super interface Common, and it's inherited by the *Appointment* interface which is implemented by Appointment*_Impl*.

After modelling the attributes, i.e., defining abstract variables from the class variables, we begin to look at the method signatures in the interfaces and at the requirements. The next step in an implementation with the support of JML specifications is to write down the method's preconditions and postconditions specifications, and also to write the class invariants which must be respected by the entire class, including its methods and constructors. These specifications will use the abstract variables that were obtained at this first step of the implementation phase.

5.2. Writing JML Invariants

Continuing from the *Appointments* example given in the *Design* phase and abstract variables written descriptions (see respectively Sections 4.5 and 5.1) and going back to the defined semi-formal class and system invariants (see Sections 4.4.2 and 4.4.3), we describe in this section the process made in the final stage of the *formal specifications pseudo-phase*, in which from the class and system invariants semi-formalized we get the JML specifications. 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 an interface *Appointment* and its implementation class *Appointment_Impl* are one class, that is, basically we consider them as one class because Appointments is just an interface for the concrete class *Appointment_Impl*.

From the class invariant C106 – **"***For all object **a** of type appointment, such that local(**a**) not equal to null, date(**a**) not equal to null, hour(**a**) not equal to null and (doctor (**a**) not equal to null and type(**a**) not equal to null).*" – attained in the first stage of the *formal specifications pseudo-phase* presented in Section 4.4.2, we get the following JML invariant in Code 12:

```
/*@ public invariant
  @     local_model != null
  @   &&
  @     date_model != null
  @     &&
  @   hour_model!= null
  @     &&
  @   doctor_model != null
  @     &&
  @   type_model != 0;
  @
  @*/
```

Code 12. Class Invariant example as a JML invariant

Where *local_model, date_model, hour_model* and *doctor_model* are *JMLValueSequences* modelling concrete data structures containing values, and *type_model* is a byte. 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 previous abstract variables.

From the system invariant attained in the first stage of the formal specifications pseudo-phase presented in Section 4.4.3 – "*SI100. For all object m of type medicine, and all object a of type appointment such that appointment(m) equals to a, then date(m) is bigger than or equal to date(a).*" – From it 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 in Code 13 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
  @             )
  @     );
  @*/
```

Code 13. System Invariant example as a JML invariant

The presented system invariant is formally specified in JML at the *CardServices* interface, because as this invariant suggests that a global access to medicines and appointments in the card must exist, and following the Java Card Remote Method invocation (JCRMI) approach for communication, in which the Java Card applet is the server, the interface *CardServices* defined in the HealthCard 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 *medicines* and *appointments* for keeping track of medicines and medical appointments respectively. In Code 13, method getData() returns an array of objects of type *Medicine* and method getApp() returns an array of objects of type *Appointment*. As seen in Code 13, we check if each medicine's date, represented by the abstract variable *date_model*, is the same respective appointment's date or a later date. This is done by using the JML *\forall* clause, where we go through every instance of Medicine and check if the correspondent Appointment instance (i.e., through a unique appointment identification attribute) has the same date or a later one. This comparison is possible only because, in this case, the abstract variable *date_model* was declared as a float type and it represents an expression that calculates a float value from a concrete sequence of byte values of an attribute date. This representation in `Medicine_Impl` and at `Appointment_Impl` classes is described in the following Code 14.

```
private/*@ spec_public @*/  byte[] date;//@ in date_model;
       /*@ public represents
         @    date_model <- date[2]*100+date[3]+date[1]*0.01+date[0]*0.001;
         @*/
```

Code 14. Representation of date_model in Medicine_Impl and Appointment_Impl

While specifying JML invariants we came across with a problem attained with the use of */exists* quantifier expression and JML abstract data types. For instances, when testing the following expression the result was always true.

```
…
requires (\exists int i; 0 <= i && i < MAX_APPOINTMENTS;
        ((Appointment)appointments_model.itemAt(i)).getId() == id);
…
```

The expression states that it is required to exist one *Appointment* object instance with an attribute *id* equal to the parameter *id* passed in a certain method. But, even if it wasn't true, when testing this expression, the result was always true. Besides this example, other similar expressions had the same problem. We finally concluded from several experiments, that the problem was in the use of quantifiers and JML abstract data types. In our example, *appointments_model* is of the abstract data type *JMLObjectSequence*. We have tried to report this problem in the Java Modeling Language forum at SourceForge.net [31], but we haven't got any word on that. For solving this problem, we made methods that only exist in the specifications level, through the use of the JML keyword model, to replace JML expressions that didn't function as supposed. The following Code 15 describes a JML helper method, *existsID*, to replace the previous expression.

```
/*@ public pure model boolean existsID(byte id){
  @    for(int i = 0; i < appointments_model.int_length(); i++){
  @          if(appointments_model.itemAt(i) != null){
  @             if(((Appointment)diagnostics_model.itemAt(i)).getId() == id)
  @                      return true;
  @          }
  @    }
  @          return false;
  @ }
  @*/
```

Code 15. JML helper method to support JML specifications

The method described in Code 15 is pure, i.e., without side effects, and exists only in the JML specifications. The method returns a boolean indicating the existence of an Appointment object with an *id* equal to the given *id* parameter. These helper methods are written as comments like JML specifications, and can be used to support invariant, methods' preconditions and postconditions specifications.

5.3. Writing JML Method Specifications

Continuing from the *Appointments* example given in the *Design* phase and abstract variables written descriptions (see respectively Sections 4.5 and 5.1) and going back to the defined semi-formal requirements (see Section 4.4.1), we describe in this section the process made in the final stage of the formal specifications pseudo-phase, in which from the semi-

formal requirements we get the JML specifications. Table 6 in Section 4.4.1 describes the semi-formal requirements that will support the JML specifications written for the methods in class Appointment, i.e., they will support the writing of method's preconditions and postconditions specifications. From those semi-formal requirements of *Appointments,* presented in that table, we get JML preconditions and postconditions specifications, but before we present the process of transformation between them, one must know that not all JML preconditions and postconditions come from the semi-formal requirements. Some JML specifications can be more oriented to the code itself rather to the requirements, for example, a specification for a data structure that describes how its elements are organized or a specification for a postcondition which guarantees that data was correctly modified. The reason why this kind of specifications isn't obtained from the requirements is that normally the requirements are written at a higher level, more closely to the real problem rather than a lower level like the Java coding.

In the following tables (Table 7 to Table 11), JML preconditions and postconditions are described. These were obtained from each of the semi-formal requirements presented in Table 6 from Section 4.4.1. One must note that each normal behaviour precondition has an inverse one at the exceptional behaviour of a method. This is for the specification of method behaviour when the normal preconditions in a contract between a caller and a server are not met (see Section 3.2.1.5 for a description about exceptions on a contract in design by contract).

Table 7. JML specifications from Semi-Formal Requirement SFR108

FROM: **SFR108**
Normal Behaviour

`requires date != null`
`&& hour != null`
`&& local != null`
`&& doctor != null;`

Exceptional Behaviour

`requires date == null`
`|| hour == null`
`|| local == null`
`|| doctor == null;`

The JML preconditions in Table 7 are written from the formal requirement SFR108: – "*If adding a new scheduled appointment, then is necessary to insert date and hour of the appointment and also the local and doctor and type of appointment.*" – Where it must be inserted these parameters when adding a new appointment, and if they must be inserted then they must be not null. Roughly speaking, in the normal behaviour, the preconditions written using the *requires* clause obligates to enter not null values when calling the method *addAppointment*. At the exceptional behaviour we have the reverse of the normal behaviour preconditions, i.e., the preconditions of the exceptional behaviour are for those cases that do not fit into the preconditions of the normal behaviour.

Table 8. JML specifications from Semi-Formal Requirement SFR110

FROM: **SFR110**
Normal Behaviour
`requires appointments_model.count(null) > 0;`
Exceptional Behaviour
`requires appointments_model.count(null) == 0;`

In Table 8, is presented an abstract variable from a collection of JML types. In the precondition's expression there's a reference to an abstract variable named *appointments_model*. This is an abstract variable of the JML abstract data type *JMLObjectSequence* which represents a data structure containing *Appointment* objects. In the previous table, the precondition written in JML was based on the semi-formal requirement SFR110: – *"If adding a new scheduled appointment and the limit has been achieved, then the system must do no changes."* – So in JML we made an expression that uses the *count* method from *JMLObjectSequence*, which is a pure method that hasn't any side effects, to count the number of null objects in *appointments_model*. By this, when there are zero occurrences of a null object, it means that the structure of appointments is full, and then the preconditions for an exceptional behaviour are met, i.e., there aren't conditions met to add a new appointment because it's full. To add an appointment, there must be at least one position in *appointments_model* with a null object, so somewhere in the *addAppointment* implementation there must be a mechanism that adds an object to a position where its object reference is null.

Table 9. JML specifications from Semi-Formal Requirement SFR122

FROM: **SFR122**
Normal Behaviour
`requires ((Appointment) appointments_model.itemAt(position)).status_model == AppointmentsSetup.STATUS_SCHEDULE;`
Exceptional Behaviour
`requires ((Appointment) appointments_model.itemAt(position)).status_model != AppointmentsSetup.STATUS_SCHEDULE;`

The previous Table 9 has a precondition written in JML that is based on the semi-formal requirement SFR122: – *"If an appointment is already checked-in, then the appointment header cannot be modified (date and time, local, doctor, type of appointment). "* – So, for this JML specification we wrote what it's required for the appointment status to be scheduled, a status before the check-in. So for the methods where this precondition will be necessary, it is required that the appointment status is equal to the *STATUS_SCHEDULE* constant value. In our work we used this JML precondition in the specifications of *setAppointmentDate*, *setAppointmentHour*, *setAppointmentLocal*, *setAppointmentDoctor* and *setAppointmentType*. All these methods are mutation methods, and after checking-in an appointment its header information cannot be modified, i.e., once the appointment is checked-in, the schedule date and hour, local, doctor and appointment type cannot be modified. For this precondition expression, we pass, as a method's parameter, the *position* of the *Appointment* object to be modified and through the *JMLObjectSequence's* method *itemAt*, we get the reference of that object and we gain access to its specifications, which in this case is the abstract variable *status_model*.

Table 10. JML specifications from Semi-Formal Requirement SFR125

FROM: **SFR125**
Normal Behaviour
`ensures ((Appointment) appointments_model.itemAt(position)).status_model == status;`

In Table 10, a postcondition in JML specification is described. That postcondition was written from the semi-formal requirement SFR125: – "If an appointment is checked-in, then that appointment must turn into a checked-in state." – So, when checking-in the system must ensure that the status of the appointment is modified. This is a postcondition because the semi-formal requirement states a system responsibility towards an event. We have used this postcondition for specifying the method *setAppointmentStatus*.

Table 11. JML specifications from Semi-Formal Requirement SFR130

FROM: **SFR130**
Normal Behaviour
requires ((Appointment) appointments_model.itemAt(position)).status_model == AppointmentsSetup.STATUS_SCHEDULE || ((Appointment) appointments_model.itemAt(position)).status_model == AppointmentsSetup.STATUS_CHECK_IN;
Exceptional Behaviour
requires ((Appointment) appointments_model.itemAt(position)).status_model == AppointmentsSetup.STATUS_DONE;

Finally, in Table 11 we have the JML specifications written from the last semi-formal requirement of Appointments: – *"If removing an appointment and its status is effective, then the system must do no changes."* – Where the abstract variable *status_model* must have a value equal to the constant *STATUS_SCHEDULE* or *STATUS_CHECK_IN* when removing an appointment. An appointment may be deleted if it has a *status* equal to schedule or checked-in. This preconditions is used for specify the *removeAppointment* method.

After writing JML preconditions and postconditions from the semi-formal requirements, we have almost every method specified. However, not all the preconditions and postconditions come from the semi-formal requirements. Some other preconditions and postconditions are written to support the specification of implementation code, i.e., specifications more related to the coding itself. It is good to reinforce the specifications that we already have from the requirements with other specifications, especially to specify things that the requirements weren't sufficient to support it, like aspects of the programming such as data structure management or specification of input values like the maximum value of a byte type.

Every single JML precondition or postcondition previously written for a certain method is brought together to describe in more detail its behaviour. As an example, from the previous obtained JML specifications described in the tables of this section (and others not showed in this examples), one can specify the *addAppointment* method as it follows:

```
/*@ public normal_behavior
  @     requires appointments_model.count(null) > 0;
  @     requires date != null
  @           && hour != null
  @           && local != null
  @           && doctor != null;
  @     requires date.length == DATE_LENGTH
A @           && hour.length == HOUR_LENGTH
  @           && local.length == LOCAL_CODE_LENGTH
  @           && doctor.length == DOCTOR_CODE_LENGTH
  @           && 0x00 < type && type <= AppointmentsSetup.MAX_TYPE_CODES;
  @     requires_redundantly (\forall int i; 0 <= i && i < DOCTOR_CODE_LENGTH;
  @                           ((byte)0x41 <= doctor[i] &&  doctor[i] <= (byte)0x5A )
  @                            || ((byte)0x61 <= doctor[i] &&  doctor[i] <= (byte)0x7A )
  @                           || ((byte)0x30 <= doctor[i] &&  doctor[i] <= (byte)0x39 ));
B @     assignable appointments_model;
```

```
  @     ensures appointments_model.count(null) == \old(appointments_model.count(null)-1);
  @     ensures_redundantly getLast(appointments_model) instanceof Appointment;
  @     ensures getLast(appointments_model).date_model.equals(toJMLValueSequence(date);
  @     ensures getLast(appointments_model).hour_model.equals(toJMLValueSequence(hour);
  @     ensures getLast(appointments_model).local_model.equals(toJMLValueSequence(local);
  @     ensures getLast(appointments_model).doctor_model.equals(toJMLValueSequence(doct);
  @     ensures getLast(appointments_model).type_model == type;
  @ also
  @ public exceptional_behavior
  @     requires appointments_model.count(null) == 0
  @          || date == null || date.length != DATE_LENGTH
  @          || hour == null || hour.length != HOUR_LENGTH
  @          || local == null || local.length != LOCAL_CODE_LENGTH
  @         || doctor == null || doctor.length != DOCTOR_CODE_LENGTH
  @          || !(0x00 < type && type <= AppointmentsSetup.MAX_TYPE_CODES)
  @          || (\forall int i; 0 <= i && i < DOCTOR_CODE_LENGTH;
  @                         ((byte)0x41 <= doctor[i] &&  doctor[i] <= (byte)0x5A )
  @                         || ((byte)0x61 <= doctor[i] &&  doctor[i] <= (byte)0x7A )
  @                         || ((byte)0x30 <= doctor[i] &&  doctor[i] <= (byte)0x39 ));
  @     assignable \nothing;
  @     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;
```

C D E F

Code 16. JML specifications for *addAppointment* method

As can be seen in Code 16, at point **A** we have the *addAppointment* normal behaviour precondition. At point **D** we have the reverse of all normal behaviour preconditions from points **A**. If one of the normal behaviour preconditions is not met, then the exceptional behaviour preconditions must be met and in that case it is specified that the system must do no changes. The normal behaviour postconditions of *addAppointment* are specified at point **C**, and exceptional postconditions are specified at point **E**. At the normal behaviour and exceptional behaviour, the *assignable* clause at points **B** and **E** are indicating the method's side effects, i.e., what data is to be modified. Only the fields listed on the *assignable* clause can be modified. If we had *assignable* *nothing*, the method shouldn't modify anything and if we had *assignable* *everything*, the method could modify any variable.

Having described how we defined JML abstract variables, JML invariants and how we written the JML specifications for specifying the method's preconditions and postconditions, next we describe how the JML specifications supported the concrete classes code writing while developing the HealthCard application.

5.4. Writing the Code

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. Taking again the *addAppointment* example with JML specifications from the *Appointments* in Code 16 (see Section 5.3), we can start to implement it in the concrete class *Appointments_Impl* by respecting its specifications and class invariants. The following Code 17 describes the implementation of *addAppointment* (in *Appointments_Impl*) according to its specifications written in Appointments interface.

```
public void addAppointment(byte[] date, byte[] hour, byte[] local, byte[] doctor, byte
type) throws RemoteException, UserException {
```

```
  boolean notInUse = true;
  byte id = 0;
  for(byte code = 0x00; code <= 0x7F; code++){
     for (short j = (short)0; j < (short)AppointmentsSetup.MAX_APPOINTMENT_ITEMS; j++) {
        if(appointments[j] != null){
              if(appointments[j].getID() == code){
                    notInUse = false;
                    break;
              }
         }
      }
      if(notInUse){
       id = code;
       break;
      }
     notInUse = true;
   }

   for (short i = (short)0; i < (short)AppointmentsSetup.MAX_APPOINTMENT_ITEMS; i++) {
     if (appointments[i] == null) {
       Appointment a = new Appointment_Impl(id, date, hour, local, doctor, type);
       appointments[i] = a;
       break;
     }
   }
}
```

Code 17. Implementation of method addAppointment from Appointments_Impl

In Code 17, in the first *for* cycle we are automatically trying to generate an identification for the new appointment to be added as a unique byte value. In that cycle, it is verified incrementally if a byte is already an identification value for another appointment instance, until the mechanism finds a value that isn't equal to any other appointment identification value. This mechanism respects a class invariant, from *Appointments*, which specifies that all appointments must have a unique identification attribute (not shown in this document). In the second *for* cycle we are looking in the appointments array for the first position with a null object. When we find a position with a null object, a new *Appointment* instance is created with the given arguments and with the generated identification (id) and then inserted into the appointments array. This last mechanism respects the postcondition normal behaviour specifications, presented in Code 16 (see Section 5.3), which states that an *Appointment* instance must be inserted and its attributes must be equal to the given arguments by the client when calling *addAppointment*. Also, in the *for* cycle, when looking for the first position with a null object we are meeting with a specified invariant (not shown in this document) which states that there must not exist any position with a null object between two positions with Appointment objects in the Appointment array.

One must notice that the precondition specifications for the method *addAppointment* in Code 16 from the previous Section 5.3, aren't validated in the method's implementation. This is because of the design by contract principles, i.e., the client calling for the method is the responsible for the method's preconditions validation. By this programming technique, it helps to avoid redundancy in the code (see Section 3.2.1.2). Redundant code may cause problems to obtain a reliable system and its performance may be penalized, i.e. problems in the ability of a system or component to perform its required functions under stated conditions for a specified period of time. Also by reducing the preconditions' validations, one is reducing the code from the server side. In our case this brings benefits, as smart cards have limited memory.

6. Validation and Verification

In this section we describe the validation and verification phase, which follows the implementation. In the software development strategy used, the validation and verification phase doesn't necessarily occur after the implementation phase, it can occur somewhat iteratively with the implementation phase. In this phase, the application java code is checked against its JML specifications statically and in runtime. If any of these cases is true, we are free to go back to other phase.

6.1. Static Assertion Checking

Even before implementing the concrete classes, it is possible to check the specifications in the Java interfaces for errors or inconsistencies. 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. The static assertion checking is made through the JML Common Tool called jml, which is a JML Type Checker (see Section 3.3.3). We have used Eclipse IDE for implementing the HealthCard application, and we have installed the JML2 Eclipse plug-in which included tools based on the JML Common Tools. One of the plug-in tools is for runtime assertion compilation and another for static assertion checks of JML specifications (semantically and syntactically). The tool used for static assertion checking of the Eclipse plug-in is based on the JML Common Tool jml, and can be used in real time while writing the specifications. The JML2 Eclipse plug-in is available in [32].

6.2. Runtime Assertion Checking

While making a runtime assertion checking, one is simulating the execution of an application's code and checking the execution against the code specifications, i.e., checking the Java code against its JML specifications in runtime. To make a runtime assertion checking, first we need to compile JML-specified Java classes into a Java byte-code that includes instructions for checking JML specifications at run-time. This can be made through the use of the JML Common Tool named *jmlc* (see Section 3.3.3). The files obtained after compiling the JML-specified Java programs are like the common compiled files of a Java program (*.class), but these can be used in runtime assertion checking with JML specifications. The following step is to generate the JUnit unit tests of the JML-specified Java classes to be tested in runtime. This can be done through the use of the *jmlunit* tool from the JML Common Tools. The *jmlunit* tool generates *JUnit* unit tests code from JML specifications. When using the *jmlunit* tool, two kinds of class files are generated, the test class and test data class. The test class generated file has methods for testing each class (to be tested) method, while through the test data class we provide data sets to be used as inputs on the methods to be tested. These two generated files are named respectively as *'*_JML_Test.java'* and *'*_JML_TestData.java'*, for instance, if generating test cases for the *Appointments* we would generate through the *jmlunit* the files *'Appointments_JML_Test.java'* and *'Appointments_JML_TestData.java'*. These two files are compiled normally as java classes, and they are executed as test drivers by using the *jmlrac* tool, a modified version of the java command that refers to appropriate run-time assertion checking libraries.

After executing the test drivers, the test results are listed. For each tested method (including the constructor) are shown a number of tests as the number of combinations of the provided given data through the test data class, i.e., methods are tested with combinations of different data as their parameters. When detecting some violation in invariants, or method's preconditions or postconditions, in the respective test result is described where and what

caused the violation. Below we describe a practical example from a verification and validation done to the method *addAppointment* from *Appointments* class.

After writing the specifications and the implementation of *addAppointment* (see Code 16 and Code 17 in Section 5.3), we have to check if the implementation is correct according with its specifications. We start the runtime assertion checking by generating the test classes. When writing the test data sets in the test data class (in this example, *Appointments_Impl_JML_TestData.java*) we should try to write down data that may violate some method's normal behaviour specifications during the runtime tests. The objective of supplying possible invalid data is to observe how the methods and their specifications reacts during the tests, that is, to check if the specifications are complete and to observe if the method tests are returning unexpected results, which in the latter case indicates some problem with the implementation or even in the specifications. During a running test, if the method to be tested receives arguments, then the inserted test data is arranged into all possible combinations to be passed as arguments. The test data supplied through a test data class have data types, thus, for testing a method, for each argument type described on the method's header, a value from the test data sets of the same type is passed through. In our work, most of the supplied data sets are of the byte types, and they are written as hexadecimal values. In Code 18 the *addAppointment* method header, from *Appointments* class, is presented. We describe the tests made to this method below as an example.

```
…
addAppointment(byte[] date, byte[] hour, byte[] local, byte[] doctor, byte type)
…
```

Code 18. Method addAppointment header

In Code 18, the method to be tested accepts byte sequences for the parameters *date*, *hour*, *local*, *doctor* and one byte for the parameter *type*. In the generated test data class 'Appointments_Impl_JML_TestData.java', we supply the data sets in methods for each different data type. Among the supplied test data, some are of the Object type, and are supplied for testing the creation of class *Appointments_Impl* objects through the method *make*: - c*ase 0: return new Appointments_Impl()* (see below Code 19, line 3). For each test data type there are methods where we supply data. For example, there is a method for supplying byte sequences, and another for supplying bytes values. In Code 19, among the supplied byte sequence data, we supplied some valid values, like {0x01,0x01,0x14,0x09} hexadecimal values for a *date* attribute (01/01/2009)*;* {0x09,0x00} for an *hour* attribute; {0x41,0x45,0x62,0x7A,0x69,0x35,0x67,0x39} for a *local* attribute and {0x69,0x69,0x69,0x69,0x69} for a *doctor* attribute. One must know that the null value is given as argument by default during the tests. Also in Code 19 we supplied a byte value 0x02, but by default during the tests the values 0xFF, 0x00 and 0x01 are given as arguments.

```
   …
1  protected Object make(int n) {
2                     switch (n) {
3                     case 0: return new Appointments_Impl();
4                     default:
5                         break;
6                      }
   …
7  protected java.lang.Object[] addData() {
8                     return new byte[][] {
9                             {0x09,0x00},
10                            {0x01,0x01,0x14,0x09},
```

```
11                              {0x69,0x69,0x69,0x69,0x69},
12                                {0x41,0x45,0x62,0x7A,0x69,0x35,0x67,0x39}
13                      };
   …
14 protected byte[] addData() {
15                      return new byte[] {
16                                  (byte)0x02
17                      };
18                  }
   …
```

Code 19. Test data sets for Appointments_Impl

According to *addAppointment*'s preconditions, the arguments *date, hour, local* and *doctor* must have a precise length defined by the constant attributes *DATE_LENGHT, HOUR_LENGHT, LOCAL_LENGHT* and *DOCTOR_LENGHT* (see Section 5.3 - Code 16) Also, not shown in *addAppointment* specification examples in this document, there is a specification constraining a *date* attribute, with the purpose of specifying a correct date format. This *date* attribute specification is written in the *Common* Java interface which is extended by *Appointments*, therefore Appointments inherit the specifications in Common. Also similar constraints exist for constraining the value format of an hour, local and doctor attributes. Continuing our *addAppointment* runtime checking example, one must notice that the valid length for each argument received by *addAppointment* is: - 4 bytes for a *date*, 2 bytes for an *hour*, 8 bytes for a *local* and 5 bytes for a *doctor*. The argument *type* must have only one byte. This limitation is one of the preconditions for the method *addAppointment*. In Figure 15, some test results for *addAppointment* are listed. The results are presented as blue ✕ and green ✔ symbols, the blue symbol indicates a test failure and the green symbol indicates a successful test. We can observe in Figure 15 that there are two successful tests. The successful tests indicate that there were no violation on the preconditions, postconditions and invariants. The arguments passed through *addAppointment* in those two successful tests were valid according to the methods preconditions.

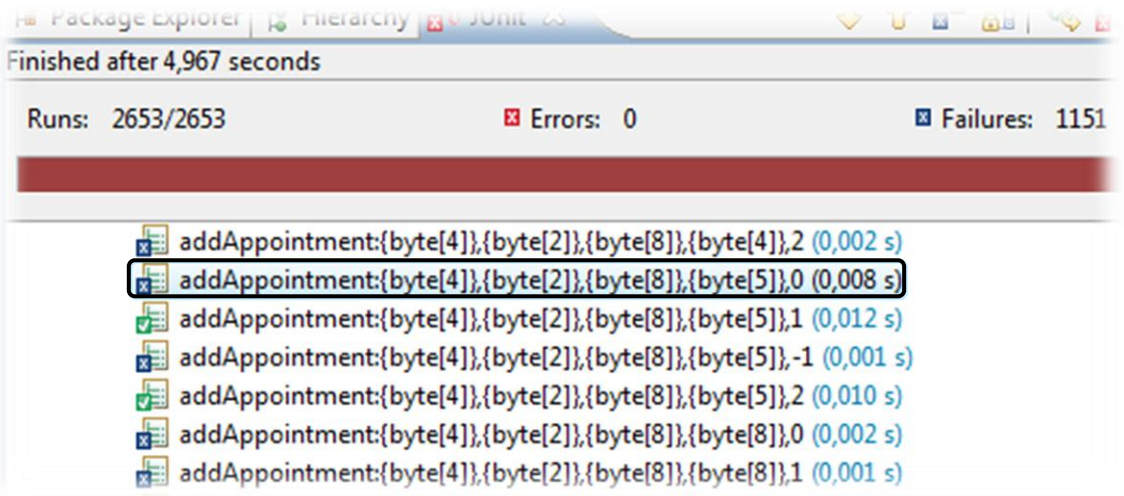

Figure 15. Some runtime assertion checking test results of addAppointment

One must notice the difficulty in checking visually which supplied byte sequences were passed as arguments, as it only shows to the tester, the type and number of elements (for instance, byte[4]). When having a method with multiple byte sequence arguments we noticed

that the tests are made through the combination of all byte sequences given in the same order as written in the data test class.

In the previous Figure 15, it's selected one of the test results. This selected test resulted in a failure, because there was a precondition violation. In the following Figure 16, the failure reason of the selected test result is shown in detail. The tester can consult these details to check where and what violation occurred while testing the method. In our example, there is a precondition violation.

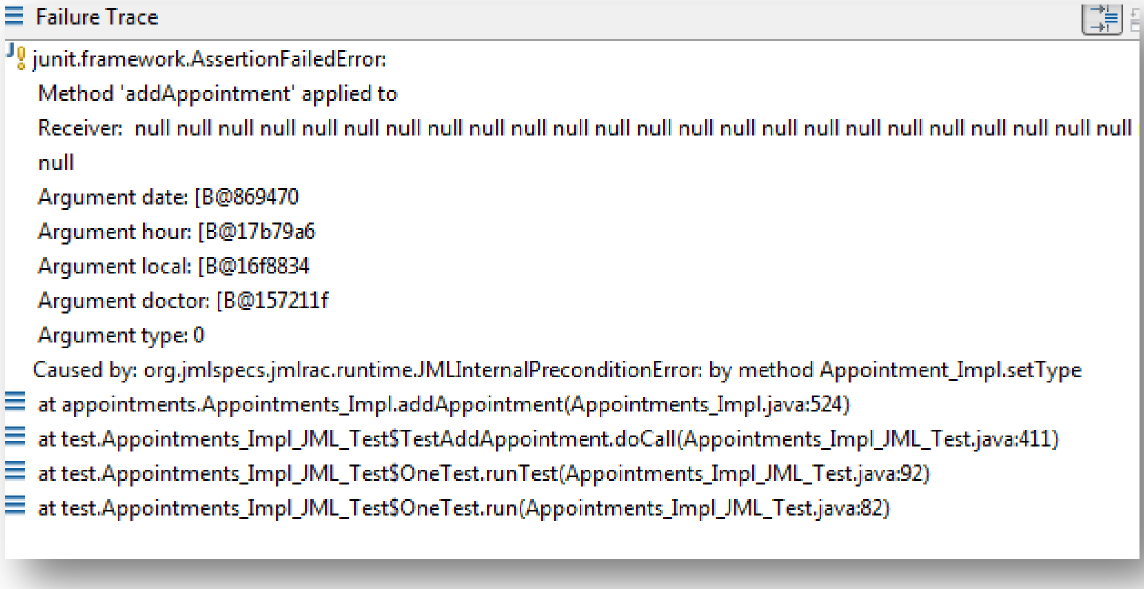

Figure 16. addAppointment test failure details

We can see that the violation was caused by a precondition error in a method named *setType* within the *addAppointment* method (i.e. JMLInternalPreconditionError). This precondition violation happened because there are a JML normal behaviour specification for the method *addAppointment* which specifies that the argument *type* must be a byte value higher than (byte) 0x00, and an exceptional behaviour specification that specifies the inverse (see Code 20). By this failure we have tested the precondition specifications of *addAppointment*, with the assurance that we have covered in the JML specifications, the case where a type argument is equal to (byte) 0x00. One must know, that according with the design by contract principles, the *addAppointment* method must not validate any precondition. For this reason, it is assumed that the client respects his part of the contract when he calls for *addAppointment*, that is, it is assumed that the argument type has a value higher than (byte) 0x00.

```
public normal_behavior
…
&& 0x00 < type && type <= AppointmentsSetup.MAX_TYPE_CODES;
```

```
…
public exceptional_behavior
…
|| !(0x00 < type && type <= AppointmentsSetup.MAX_TYPE_CODES)
…
```

Code 20. Argument type JML specification for the method addAppointment

Both static and runtime assertion checking in the verification and validation phase can be used while implementing the system. By this iterative process, one can evolve the Java code as well the JML specifications according to the validations made through testing.

7. The HealthCard Application Metrics

After the HealthCard application development we did some measurements to determine the quantity of written code and specifications. In the following Table 12, it is described the number of classes (interfaces and concrete classes) and their number of Java source code and JML specification lines. From these measurements we conclude that almost half of the lines written are JML specifications (50,12%) for almost another half of source code (42,01%), being a small percentage (7,87%) the amount of mixed lines. We didn't make any study about an application of this type developed without the use of JML specifications, but certainly we would end with a large number of lines of code, due to the large number of validations performed within the methods implemented on the card side. The JML specifications describe the conditions that, according to the design-by-contract principles, which according to the design by contract principles, must be respected by the clients when calling the specified methods, so it is assumed the method's preconditions are valid and therefore no validation coding is necessary within these methods. When developing a smart card application, one has to consider the card's limited capacities. The HealthCard application has 36 classes, being 20 interfaces and 16 concrete classes. The total lines of code are almost 4300.

Table 12. The HealthCard application metrics per module

Modules	Modules Size (bytes)	Classes	Interfaces	Concrete Classes	Source Code Lines	Source Code Lines (%)	Specification Lines	Specification Lines (%)	Mixed Lines	Mixed Lines (%)	Total Lines
Allergies	17.768	5	3	2	151	37,75%	220	55,00%	29	7,25%	400
Appointments	37.158	5	3	2	325	37,27%	483	55,39%	64	7,34%	872
Diagnostics	22.400	5	3	2	268	45,50%	271	46,01%	50	8,49%	589
Treatments	23.551	5	3	2	262	42,26%	299	48,23%	59	9,52%	620
Medicines	32.027	5	3	2	359	44,21%	385	47,41%	68	8,37%	812
Vaccines	15.985	5	3	2	150	41,21%	189	51,92%	25	6,87%	364
Commons	12.871	2	1	1	61	17,18%	265	74,65%	29	8,17%	355
CardServices	10.433	3	1	2	183	85,51%	19	8,88%	12	5,61%	214
HealthApplet	1.572	1	0	1	34	80,95%	8	19,05%	0	0,00%	42
Totals	**173.765**	36	20	16	1793	42,01%	2139	50,12%	336	7,87%	4268

In the following Table 13 we describe the quantity of Java source code and JML specification lines per Java interfaces and concrete classes. From the values presented in the table, we can see that almost of the JML specifications are written within the Java interfaces (abstract classes), being 1605 specification lines against the 263 lines in the concrete classes. This is because of the way the system was designed to achieve a better reusability and

maintainability. If we write the specifications in the interfaces, we can write the implementation in concrete classes in several ways as long as we respect what is specified in the super-classes.

Table 13. The HealthCard application metrics per interface and concrete classes

Java Files	Number	Source Code Lines	Source Code Lines (%)	Specification Lines	Specification Lines (%)	Mixed Lines	Mixed Lines (%)	Total Lines
Interfaces	20	431	19,01%	1605	70,80%	231	10,19%	2267
Concrete Classes	14	1212	76,71%	263	16,65%	105	6,65%	1580

Also based on Table 13, we can see that most specification effort is done in the interfaces. Most of the JML specifications are written in the interfaces to add some level of abstraction to the implementation process and to make the specifications more reusable. By this way, we can implement the concrete classes in various ways while respecting the specifications written in the interfaces (i.e. reusability).

8. A Client-Side Small Example

After having fully specified and implemented the in-card HealthCard application we will show know how to write a simple method from a client that call methods of the application while adhering to its JML specifications (see Section 5). We will describe how to write the client-side Java code for the method `addAppointment(byte[] date, byte[] hour, byte[] local, byte[] doctor, byte type)` shown in Code 16 (see Section 5.3). The code client-side will be determined by the JML specification of the method, and the Design-by-Contract principles underlying JML (Section 3.2.1), which states that a method precondition has be respected by the client to be able to call the method, and the method postcondition must be ensured by the method supplier.

In the following, we describe the JML specification of the method `addAppointment`. The precondition specification below states that if the number of null appointments in the card is positive (so a free slot exists), a new appointment object element can be inserted into the list of appointments.

```
@     requires appointments_model.count(null) > 0;
```

The JML specification below makes clear that the date, hour, and local associated to an appointment cannot be null.

```
@     requires date != null
@           && hour != null
@           && local != null;
```

The method precondition specification below states well-formedness conditions about objects of type appointment.

```
@  requires date.length == DATE_LENGTH
@           && hour.length == HOUR_LENGTH
```

```
  @             && local.length == LOCAL_CODE_LENGTH
  @             && doctor.length == DOCTOR_CODE_LENGTH
  @             && 0x00 < type && type <= AppointmentsSetup.MAX_TYPE_CODES;
```

The client-side implementation below reflects the JML specifications for the method `addAppointment` above.

```
    ...
1       if(countAppointments() < AppointmentsSetup.MAX_APPOINTMENT_ITEMS){
2           if(date!=null && hour != null && local != null ){
3               if( _date.length == Common.DATE_LENGTH
4                   && _hour.length == Common.HOUR_LENGTH
5                   && doctor.length() == Common.DOCTOR_CODE_LENGTH
6                   && local.length() == Common.LOCAL_CODE_LENGTH
7                   &&  0<= Integer.parseInt(type)
8                   && Integer.parseInt(type) <= AppointmentsSetup.MAX_TYPE_CODES
9                   myAppointments.addAppointment(_date, _hour, _local , _doctor, b);
    ...
```

Code 21. From the client addAppointment method

9. A Prototype Tool for Generating JML Formal Specifications from Informal Software Requirements

In the following, we describe a prototype tool we built to automate the process of writing JML formal specifications from semi-formal specifications. We used João Pestana's software development strategy for implementing the HealthCard (see Section 5). In this strategy, semi formal specifications are transformed into JML specifications. The transformation relies on the idea that semi-formal specifications can be written in a "if <event/condition> then <restriction/rule>" way. This was an important factor that determined the way we implemented our prototype tool: after "if" and before "then" there is a method precondition; after "then" there is a method postcondition.

We further looked at the requirements written for the HealthCard and noticed some patterns that relate semi-formal requirements and JML formal specifications. Preconditions in the semi-formal requirements have logical conditions. In these conditions data structures, variables, constants or values, joint with relational and logical operators (EQUAL, NOT EQUAL, MORE THAN, LESS THAN, MORE OR EQUAL, LESS OR EQUAL, AND, OR) are found. For instance, in the informal requirement *"To schedule an appointment, a valid date and a valid type for the appointment must also be inserted"*, a logical *and* operation for the *date* and *type* of an appointment is found (see below). Furthermore, the "then" part outlines a JML method postcondition.

if *< date NOT EQUAL null AND date is an array that has length EQUAL length of array date_model AND type MORE THAN 0)>*

then *<date EQUALS date_model AND type EQUALS type_model>*

We show below the JML specification for the semi-formal specification above

/*@ public normal_behavior

```
@   requires date != null
@            && date.length() == date_model.length()
@            && type > 0
@*/
```

We show below a screenshot (Figure 17) implementing the reasoning introduced above. The prototype tool was based on Adobe Flex 3 [9] technology.

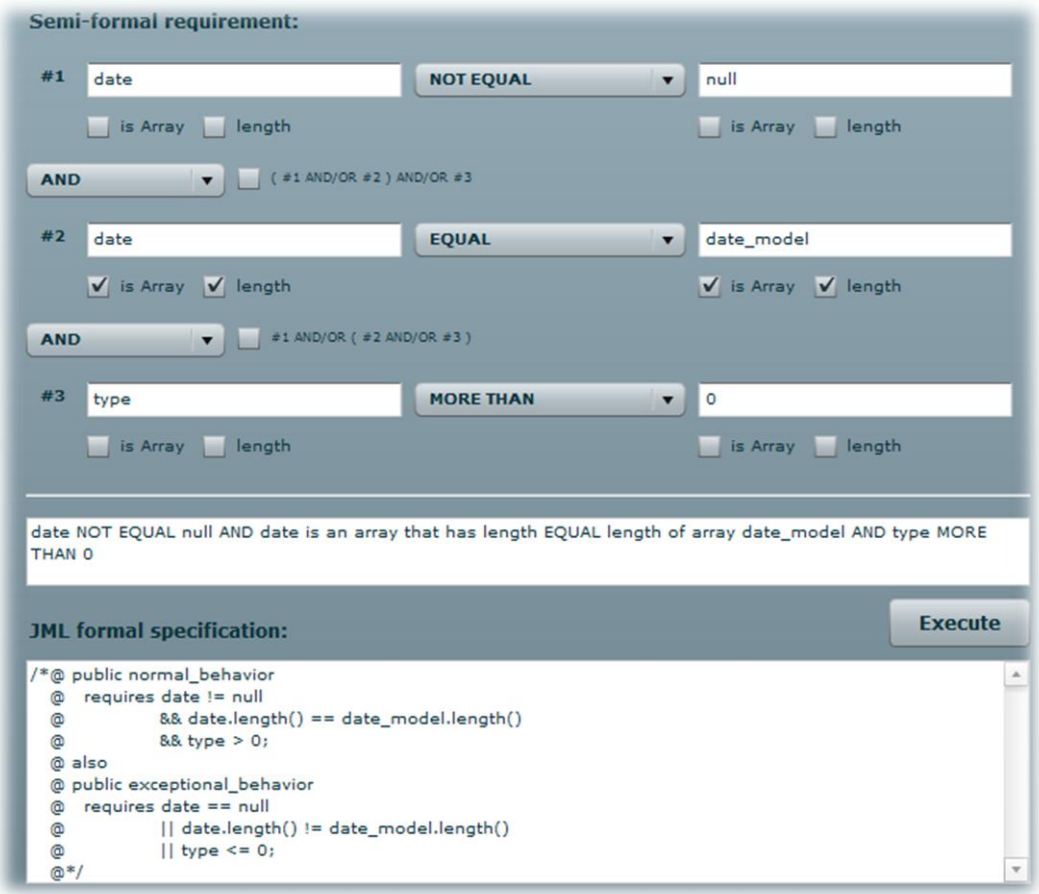

Figure 17. Screenshot from the prototype

[9] Adobe Flex is a software development kit released by Adobe Systems for the development and deployment of cross-platform rich Internet applications based on the Adobe Flash platform

10. Conclusion

This master thesis is about the JML driven formal development of a privacy sensitive Java application. It was developed following João Pestana's strategy, introduced in [25]. We consider that this strategy is suitable for developing correct applications implementing any client-server architecture with a strong need for a lightweight server specification in general. In this client-server setting, validations of methods' pre-conditions are not carried out within methods' implementations. On the contrary, it is the client's responsibility to ensure that methods are called with the appropriate parameters. This reduces the size of implemented methods. This is particularly important for smart cards whose generated byte-code cannot be bigger than a certain limit. JML ensures that methods are called with the right parameters. This prevents programmers from making validations both inside and outside methods, a common programming mistake.

One of the main goals of our work was to validate João Pestana's strategy. Our work benefits from using this strategy, hence bringing informal software requirements (as understood by clients and general stakeholders) into a final software product (as understood by developers and engineers) that fulfils the real needs of the client. Evolving informal requirements into formal specifications is not a linear process; it requires great ingenuity and experience. This evolving process is stepwise. Therefore, we transform informal functional requirements into semi-formal ones, before being expressed as JML specifications.

The use of JML facilitates the communication between requirements' teams, designers and programmers. JML specifications further serve as software documentation, likewise *JavaDocs*, but while the first can be used to check the source code against it, the latter cannot. We carried this checking with the JML Common tools, which automatically generated JUnit tests from JML specifications. These tests are checked in runtime.

We implemented a Smart Card application written in Java Card. This explains the use of JML for writing our specifications. The syntax of Java Card is limited compared to syntax of Java itself. For instance, Java Card does not support dynamic memory allocation, and some native data-types are missing. These limitations led us to use JML model types to abstract data structures. For instance, the *JMLObjectSequence* model type was used to represent a collection of objects, and the *JMLValueSequence* model type was used to represent a collection of values. The use of JML abstract variables keeps the specifications maintainable and reusable, giving a range of possibilities for the actual concrete data structure to be implemented.

JML specifications can be re-used straightforward. For instance, the more restricted version of the JML specification of the in-card implementation was re-used for the specification of the prototype of our Java client easily.

In Bertrand Meyer's design-by-contract principles, the specified methods oughtn't to implement validations of their preconditions in their method bodies, because those validations are of the client's responsibility, i.e., the method's preconditions must be assured by the clients when they call them. The employment 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 weighted, 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. In our case, when we developed the HealthCard, we needed to reduce the card side code, because we had to consider the smart card's memory limitation. We use JML to describe contracts, and we use

the JML Common tools to check against the code. Using JML specifications while developing the HealthCard, we didn't only make a correct system but also we have reduced considerably the card side code, leaving the precondition validations to the external client applications when they called for remote methods.

Other interesting aspect of using design by contract principles is that it also helps to reduce the redundant validation code, a common programming error. Removing redundant code potentially helps to improve the performance of applications. If we produce strong preconditions, and they are already respected in the client side, these preconditions become redundant. For instance, if the preconditions are describing that some arguments must be different from null, the client must guarantee the validation of those arguments before calling the method. The card side will have less code because there is no need to verify the null arguments twice and the application will not achieve an abnormal state. This will make the card application lighter.

We want to stress on the use of JML as specification language for applications that use the Java Card Runtime Method Invocation (JCRMI) as the model of communication between the server and the client side applications. JCRMI and the design-by-contract (as enforced by JML) can be used together in a way that the remote Java interfaces or abstract classes in the server side (card-side) become place where specifications can be written and respected when a client makes a call to the remote method. These JML-specified remote Java interfaces are shared with the client side. Hence developers can implement the client side supported by the specifications written for the shared Java interfaces.

In our work, some challenges became evident from the application domain itself, i.e. JCRMI and Java Card. Sometimes problem arose when we were using a client that tested methods of the Health Card. In those cases, the JCRMI threw an exception because we didn't know about JCRMI data transfer limit. Thus, every time the method was called, our application crashed. After some tests we found out that JCRMI does not provide support to messages of more than 150 bytes. All this put in evidence the fact that, even having some freedom to express specifications, we still needed to master the language used for implementing the HealthCard.

To introduce formal methods in software industry, we first have to educate people on the strategies for incorporating formal specifications, and formal methods in general, into software development. This not only counts for software developers, but at some extent, people leading groups. Formal methods provide ways document code and new ways to implement software. Different software development groups can implement an application in parallel from a common formal specification. For instance, in a client-server application, one team can work on the client side implementation, the other on the server side.

Regarding Madeira Island, we can envision having a central database with the information of medical data records of Madeira people. The full medical data records would be formally specified, for example using JML. Therefore, any company in the area of the healthcare that wants to develop a software application must comply with the common formal specification of the medical data records.

Finally, we want to emphasize that the CardSecurity class declared in the Java Card tools was not specified in JML. However, we re-used the CardSecurity class in our Java implementation of the HealthCard.

11. Bibliography

1. **Warnier, Martijn.** *Language Based Security for Java and JML.* 2006.

2. **Ortiz, C. Enrique.** Sun Developer Network. *An Introduction to Java Card Technology - Part 1.* [Online] May 29, 2003. http://java.sun.com/javacard/reference/techart/javacard1/.

3. **Cardlogix Corporation.** Smart Card Standards. *Smart Card Basics.* [Online] 2009. http://www.smartcardbasics.com/standards.html.

4. **Ort, Ed.** Writing a Java Card Applet. *Sun Developer Network (SDN).* [Online] January 2001. http://java.sun.com/javacard/reference/techart/intro/.

5. **Warnier, Martijn and Oostdijk, Martijn.** *Java Card Remote Method Invocation.* University of Nijmegen : s.n.

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

7. **Sommerville, Ian.** Formal Specification. *Software Engineering.* 2000, 9.1, pp. 159-169.

8. **Liu, Shaoying, et al.** Teaching Formal Methods in the Context of Software Engineering. *inroads — SIGCSE Bulletin.* 2, June 2009, Vol. 41, pp. 17-23.

9. **Kemmerer, Richard A.** Integrating Formal Methods into Development Process. September 1990.

10. **Fraser, Martin D., Kumar, Kuldeep and Vaishnavi, Vijay K.** Strategies for Incorporating Formal Specifications in Software Development. *Communications of the ACM.* 10, October 1994, Vol. 37.

11. **Priestley, Mark.** *The Logic of Correctness in Software Engineering.* Cavendish School of Computer Science, University of Westminster. London : s.n.

12. **Meyer, Bertrand.** Design by Contract: Building Reliable Software. *Object-Oriented Software Construction.* s.l. : Prentice Hall, 1997, 11, pp. 331-410.

13. **Eiffel Software.** DESIGN BY CONTRACT AND ASSERTIONS. *Eiffel Software.* [Online] http://archive.eiffel.com/doc/online/eiffel50/intro/language/invitation-07.html.

14. **Tucker, Allen and Noonan, Robert.** Program Correctness. *Programming Languages: Principles and Paradigms.* s.l. : McGraw-Hill Science/Engineering/Math, 2001, 12.

15. *Design by Contract com JML.* **Júnior, Rogério Dourado, Figuereido, Jorge César and Guerrero, Dalton Dario.** [ed.] UNISINOS. São Leopoldo : s.n., 2005. XXV Congresso da Sociedade Brasileira de Computação . pp. 1455-1499. published in portuguese language.

16. **Meyer, Bertrand.** Applying “Design by Contract". *Computer.* 1992, pp. 40-51.

17. *The ESC/Java 2 Tool.* [Online] http://secure.ucd.ie/products/opensource/ESCJava2/.

18. *The Jack Tool.* [Online] http://www-sop.inria.fr/everest/soft/Jack/jack.html.

19. *The Krakatoa Tool.* [Online] http://krakatoa.lri.fr/.

20. **van den Berg, J. and Jacobs, B.** The LOOP compiler for Java and JML. *In Proceedings of TACAS.* s.l. : Springer, 2001, pp. 299-312.

21. **Leavens, Gary.** JML Reference Manual. [Online] 20 May 2008. http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman.html.

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

23. **Leavens, Gary T.** Documentation. *The Java Modeling Language (JML).* [Online] 29 July 2008. http://www.eecs.ucf.edu/~leavens/JML//documentation.shtml.

24. **Leavens, Gary.** Downloading the JML Common Tools. *The Java Modeling Language (JML).* [Online] http://www.eecs.ucf.edu/~leavens/JML/download.shtml.

25. **Pestana, João.** *A JML-Based Strategy for Incorporating Formal Specifications into the Software Development Process.* Universidade da Madeira. 2009.

26. *Executing jml specifications of java card applications: A case study.* **Catano, N. and Wahls, T.** Waikiki Beach, Honolulu,Hawaii : ACM (SAC), 2009.

27. **ARC - Care Parent Network.** Information and Medical Forms. *CARE Parent Network Resource .* [Online] http://www.contracostaarc.org/html/careresources.html.

28. **Cornell University.** Making Appointments . *Gannett Health Services .* [Online] http://www.gannett.cornell.edu/accesstocare/appointments.html.

29. **Medical Assistant.net.** Medical Assistant Net - What is a SOAP Note? *Medical Assistant.* [Online] http://www.medicalassistant.net/soap_note.htm.

30. **NetBeans.** NetBeans IDE. *NetBeans.org.* [Online] 2009. Fully-featured Java IDE written completely in Java. http://www.netbeans.org/.

31. **Leavens.** Java Modeling Language. *SourceForge.net.* [Online] http://sourceforge.net/tracker/?func=detail&atid=510629&aid=2822469&group_id=65346.

32. **Swiss Federal Institute of Technology Zurich.** JML2 Eclipse Plug-In. *Department of Computer Science - Chair of Programming Methodology.* [Online] http://www.pm.inf.ethz.ch/research/universes/tools/eclipse.

33. **Leavens, Gary T. and Cheon, Yoonsik.** *Design by Contract with JML.* Dept. of Computer Science, Iowa State University; University of Texas at El Paso. 2006.

34. **Bell, Donald.** UML basics: The class diagram. *IBM.* [Online] 15 September 2004. http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/.

35. **Riehle, Dirk.** *Method Types in Java.* s.l. : SKYVA International, 2000.