Universidade da Madeira

Manual
for HealthCard_JavaCard-JML using Eclipse

Ricardo Rodrigues
Master in Informatics Eng.
University of Madeira
Funchal, Portugal
ricardo.rodrigues@max.uma.pt

1. Introduction

This HealthCard prototype was developed under a master thesis work made Ricardo Rodrigues from University of Madeira. The thesis work consists on a strategy to incorporate JML formal specifications into software development processes. The strategy is applied to formalize the informal functional requirement by passing through a middle stage during the process in which we produce semi-formal functional requirements, class invariants and system invariants. As JML is a Design-by-Contract tool which besides serving as a support to formalize the specifications, it has also a documentation aspect that supports the definition of contracts between clients and servers. The development of this HealthCard prototype had the purpose of validating that strategy as well as applying the concept of contracts with the support of the written JML specifications. We developed a small client application having in mind that all calls to the card side had to be respected. That is, when the client side had to make a remote call to a card's method, we had to validate the preconditions in the client side.

The available prototype is divided into Running Version, and JML Version. The Running Version is already prepared to be used after compiling the files and converting the card side image. It contains a client application prototype. The JML Version contains only the card side code and its JML specifications. It is an incomplete version, and it only serves to test the Java Card code against its JML specification. This version is ready to be processed and tested to check the JML specifications against the Java Card code.

This document describes how to install and use the *HealthCard_JavaCard-JML* prototype. The prototype can be downloaded from http://sourceforge.net/projects/healthcard/ website. Then the files must be uncompressed to a directory.

Also a compressed file containing the images of HealthCard's class diagrams are also available in the website for download.

2. The HealthCard Application

In the following, we describe HealthCard, the application used to validate our strategy proposed in our thesis. HealthCard stores people's medical information. It is named HealthCard because it runs on a smart card, a pocket-sized plastic card with embedded integrated circuits that process data. A typical smart card application includes on-card applets (the applets running on the card), a card reader-side, and off-card applications (e.g., a computer program communicating with the card applets). HealthCard is written in Java Card, a subset of Java used to program card applets. We used the Java Card Method Invocation (JCRMI) model for communication between off-card applications and on-card applets. This model implements a client-server setting with the HealthCard acting as server, and off-card applications as clients, communicating via APDU (Application Protocol Data Unit) messages. Figure 1 shows the structure of the HealthCard smart card.

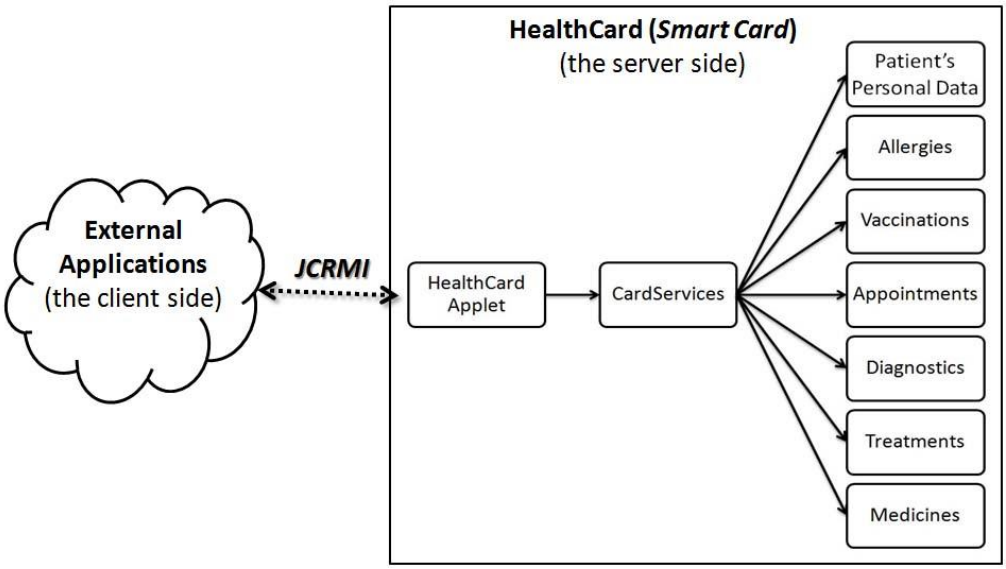

Figure 1. HealthCard application structure

A patient can use his HealthCard to furnish accurate medical information to general practitioners in medical centres with the appropriate system to read it. The HealthCard manages the patient's personal details (*Personal* module), his historical record of allergies (*Allergies* module), vaccines (*Vaccines* module), diagnostics (*Diagnostics* module), treatments (*Treatments* module) and prescribed medicines (*Medicines* module). Also it is possible to schedule appointments with the HealthCard (*Appointments* module), where it is recorded the date and time, local and doctor or type of appointment. The HealthCard is divided in several modules for managing the medical information. Each module has a remote interface, and an implementation class that serves the appropriate services. All the remote interfaces are referenced in a single remote interface named *CardServices* whereby an external client can invoke services. For example, if an external client calls the method *getAppointments*() in *CardServices*, he gets a reference to the *Appointments* remote interface. This reference can then be used to invoke appropriate methods implementing services.

3. Java Card Installation and Usage

We start by referencing some of the tools needed for developing a Java Card application. The following tools were used to develop the Health Card application (card side):

- Eclipse version 3.2.2. [1]
- Java Card Development Kit version 2.2.2. [2]
- *EclipseJCDE*, Eclipse plug-in [3]

To install the Eclipse, you just need to download it and extract it to a system directory, for instance *"C:\"*. The Eclipse will be ready to use afterwards.

[1] Eclipse IDE: http://www.eclipse.org/downloads/
[2] Java Card Development Kit: http://java.sun.com/products/javacard/
[3] EclipseJCDE, Eclipse plug-in: http://sourceforge.net/project/showfiles.php?group_id=176931

We have also used the Java Card Development Kit (JCDK) that comes with tools and libraries necessary for the development and testing of Java Card applications. To install it, one has to download it and extract it to a system directory, for instance it could also be "*C:\"*, and next one has to setup the environment variables of the OS and add the following lines (assuming that JCDK is installed in *"C:\"*):

- *JC_HOME : C:\java_card_kit-2_2_2*
- *Path: %JC_HOME%\bin*

We also installed the *EclipseJCDE* plug-ins (http://eclipse-jcde.sourceforge.net/) for the Eclipse IDE. The Eclipse plug-in allow to apply AID (Application Identifiers) to Java Card class packages and to applets, generate scripts from those packages and execute them by wrapping their APDU commands to communicate with a smart card simulator. Without this plug-in the process of writing the scripts would take much longer and would be tiresome. To install the plug-in, one should download it and uncompress it into the plug-in directory of the Eclipse installation directory. Then, one has to initiate Eclipse and setup the Java Card directory in "Java Card → Preferences" then fill it in "Java Card Home". This has to be the directory where we installed the Java Card Development Kit.

3.1. Using the Java Card

After installing and setting up the Java Card environment and its tools, we describe the Java Card usage in the card side and client (external) side.

2.1.1.1. The card side (server)

Using the *Eclipse* and after installing the EclipseJCDE, for creating a new Java Card project one must choose the option "Java Card Project" and give it a name. The needed libraries for developing Java Card applications are automatically added to the project. Afterwards, we import the files from the "*runningversion\healthcard_server\src"*, uncompressed before from HealthCard_JavaCard-JML.rar, to the project created. You can simple drag the packages and left into the project on Eclipse.

Then, it's necessary to apply an AID to the applet and packages to uniquely identify them. This can be done through the use of *"Set Package AID"* and *"Set Applet AID"* from the EclipseJCDE plug-in. One must notice that the external applications calling the card services from a certain applet must have the same applet AID in their code mechanisms to select a remote applet. For the running version of the HealthCard, you must setup the AID for the Applet as 0x01:0x02:0x03:0x04:0x05:0x06:0x07:0x08:0x09:0x01. The AID is used by external applications to select the respective card applets. For each multiple packages, each one of them must have a unique AID, otherwise it will cause conflicts and it won't be possible to convert them into CAP files.

Having applied the AIDs, one compiles the Java Card files like normal Java files. To generate the APDU scripts one must select each package and by clicking the right button one chooses "Java Card Tools → Generate Script". This procedure must be done for each package, beginning by the least dependent packages, in our work we start with Commons, Allergies,

Appointments, Personal, Vaccines, Diagnostics, Treatments, Medicines, CardServices and Card packages.

After generating the script files we have to edit manually the *"cap-download.script"* of each package. This is basically needed for completing the script that is necessary for sending installation APDU commands into the smart card. For the script files from the package containing the applet(in our project Card package), in the *"create- HealthApplet.script"*, we have to copy the line referent to the applet creation, i.e., the APDU command line below the comment *"// create HealthApplet applet"*, and then we have to paste it in the "*cap-download.script"* of the same package just before the APDU command "0x80 0xBA 0x00 0x00 0x00 0x7F;" and the "powerdown" lines at the end of the file.

The next step, for each package's script file *"cap-download.script"*, and beginning from the least dependent package Commons, we have to copy all the APDU commands between the "powerup" and "powerdown" lines and then we have to paste it in the same script file where we pasted the create line, i.e., in the *"cap-download.script"* from the *HealthApplet applet* in the package Card. We paste it after the "powerup" line and before the *"// select the installer applet"* line, moreover those pasted scripts must be ordered by the dependency level of each respective package, starting with Commons, Allergies, Appointements, Personal, Vaccines, Diagnostics, Treatments , Medicines and finishing with CardService package. For example, we have to paste the script lines of Commons after the "powerup" line and then we have to paste the script lines of Allergies after the script lines of Commons, and so on for the rest of the packages.

After creating and editing the scripts we end up with a single script file containing the APDU commands from all other scripts. Now we can execute this script to install the Java Card applet into a smart card or a Java Card environment simulation (i.e., image file simulating a smart card). To simulate a Java Card environment, we have decided to use CREF.

Using the CREF tool through command line we create an image file which waits for the script downloading. First we write by command line the following command *"cref –o <image file name>"*, and then the execution of CREF waits for the script downloading. Next, through Eclipse we select the script containing the APDU commands and by right clicking we choose *"Java Card Tools → Run Script"* to download the script into the image file to install the Java Card application. If all ends well, after downloading the script, we have created the image file simulating a smart card that contains the Java Card application.

When running external Java applications to access the image file, like it was a smart card, we use the CREF tool, but the command line given is *"cref –i <image file name>"*. The execution of CREF waits for external APDU commands sent to the simulated smart card. The command *"-i"* makes the image file readable, but we can make the image file permanently writable (i.e., to hold information after the execution) by adding the command "-o", making possible to store permanently the information inserted in the image file, during the simulation (i.e., *"cref –i –o <image file name>"*).

Next we describe the use of Java Card in external applications, i.e., the client side.

2.1.1.2. The client side

The client side or external applications that will communicate with the card can be implemented in normal Java language.

First of all, we recommend creating a project with the name Healthcard_Client. The project has to include the following Java Card Development Kit libraries: *apduio.jar*, *jcclientsamples.jar* and *jcrmiclientframework.jar*.

After creating the project we have to import the source code from *"runningversion\healthcard_client\src"* to the project and the *images* from the *"runningversion\healthcard_client\images". You* just simply have to drag the images files to the project and each package from source to current source from the project.

As this is a system implementing the Java Card RMI (Remote Method Invocation), for each remote interface of the card application, it must be created a stub of their implemented remote objects. This stub must be present in the card side applications to be possible a communication between the client side and the card side (i.e. server side). To create a stub we present the following command line example: - *"rmic -classpath C:\java_card_kit-2_2_2\lib\javacardframework.jar;. card.Personal_Impl"* – where following the –classpath we write the path to the needed Java Card library and *";."* to refer the directory of *card* where the compiled class *Personal_Impl* is. The directory *card* is the package where *Personal_Impl* is and *Personal_Impl* is the implementation of the remote interface *Personal* from the card side. The result of this command line is the creation of a stub file of *Personal_Impl*.

After creating the stubs, we copy them and the remote interfaces to the client side. In our case we have done a *.bat file (rmicHealthCard.bat) to execute all this command lines and copy the files automatically, as it is a tiresome task. The rmicHealthCard.bat file is in *"runningversion\healthcard_server\bin"*. The file must be copied and pasted in the \bin directory of the current project from Java Card. Don't forget to check the paths from bat file are correct them if needed. You must check if the *.class files were created in the HealthCard_Server directory \bin. At this point we have all prepared to execute the external application and establish a connection with the card side, the only thing necessary before running the client applications is to use the CREF tool to execute the image file to simulate the insertion of a real smart card. Then to execute the client application you have to launch HealthCardGUI.

3. JML Installation and Usage

In this section we describe the tools needed for installing and using the Java Modelling Language as well as some necessary steps of its use.

3.1.1. Installation and setup of JML

The following tools are necessary to install and use de JML:

- Java Runtime Environment version 1.4.* (JML only works with this version)
- Eclipse version 3.4.1. (The only compatible version with the existing JML plug-ins for Eclipse.)

- JML Common Tools [4] (we used version 5.4.)
- JML2 Eclipse Plug-in Project
- JUnit 3 [5]

This Eclipse IDE used for working with JML is a different version to that used for working with Java Card. This is because of each plug-in of Java Card and JML being neither compatible with the same Eclipse version. We had to write the JML specifications and Java Card code in the Eclipse version with the JML plug-in installed, and the Eclipse version with the Java Card plug-in installed was only used for writing the Java Card applet and security coding, and running the Java Card application. To install the Eclipse we only have to download it and extract it in a directory.

To install the JML Common Tools we need to download them and extract them in a directory (for example, *"C:\JML"*) and next we have to setup the necessary environment variables of the OS as it follows:

- *Path: %JML_HOME%\bin*
- *JML_HOME : C:\JML*

And, in the *classpath* environment variable we have to add the following:

- …JML\bin\jml-release.jar; …j2re1.4.2_18\lib\rt.jar; …j2re1.4.2_18\lib\sunrsasign.jar; …j2re1.4.2_18\lib\jsse.jar; …j2re1.4.2_18\lib\jce.jar; …j2re1.4.2_18\lib\charsets.jar; …j2re1.4.2_18\classes; ...JML\specs\; …JML\org\jmlspecs\models; …java_card_kit-2_2_2\lib\api.jar

One must notice that we have also installed Java Runtime Environment version 1.4.2. -update 18, because it was the latest compatible version with the most recent version of the JML Common Tools (at the time it was 5.4.). At last, we have to make sure that the Eclipse has the JUnit 3. The JUnit 3 is needed for executing the Runtime Assertion Checking. The JUnit is a unit testing framework for the Java programming language and along with the *jmlunit* tool from JML Common Tools we can make runtime assertion checks

The JML2 Eclipse Plug-in Project is an Eclipse plug-in that integrates the JML Common tools into the Eclipse. Through this plug-in, the process of writing the JML specifications is easier, especially its functionality of checking the JML syntax while writing the specifications. To install this plug-in we have to start Eclipse and then we have to go to "help → software updates… → Available Software → Add Site". In there we insert the following website http://www.pm.inf.ethz.ch/research/universes/tools/eclipse/ to obtain the JML2 Eclipse Plug-in Project.

To setup the automatic JML checker for automatically make the static checking of JML specifications we have to right click on a project and select Properties, the we have to select "JML2 Plug-in → Automatically run JML2 Checker".

[4] The JML Common Tools: http://sourceforge.net/projects/jmlspecs/files/
[5] JUnit: http://www.junit.org/

Also, before using this Eclipse plug-in for working with JML we need to include the following libraries into the Eclipse project:

…JML\bin\jml-release.jar
… JML\specs
… java_card_kit-2_2_2\lib\apduio.jar
…java_card_kit-2_2_2\lib\api.jar

3.1.2. Using the JML Common Tools

To use the JML Common Tools we may execute the *jml-release.jar* to launch a graphical version, or we can use its tools through command lines.

Once we have Java code specified with JML we may use the *jml* tool from the common tools to make a static assertion checking. The alternative way of making a static assertion checking is to use the automatic tool supplied by the JML2 Eclipse Plug-in. This alternative is better because while one writes the JML specifications, the tool checks automatically.

To compile Java files with JML we can use the plug-in function "JML2 Compiler" or we can compile through the *jmlc* tool from the common tools. This compiles like the compilation of normal Java files but with the addition of JML. The compilation is necessary before executing a runtime assertion checking. If using the jmlc to compile, one has to copy and paste the resulting class files to the *"…/bin"* directory of the working Eclipse project and replace any previous compiled file. By default the compiled files are put in the bin directory and we have to replace those compiled files because normally the Eclipse compiles automatically as normal Java files when we save the changes in the code (or in JML specifications).

To make a runtime assertion checking of the JML specifications and implementation code we have to generate the testing files and provide some test data, after compiling the Java files with JML. 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 files are already created in the test package in the*"jmlversion\healthcard_jml\src"*, so it is only necessary to import all packages to the project created. The ones already generated by us have some data to be tested. At last, when *'*_JML_Test.java'* and *'*_JML_TestData.java'* are imported into the project and the compiled class files in bin directory, with the Junit we run the *'*_JML_Test.java'.* If all works fine, the test results will be presented.