

A Semi Formal Model of Java Card 2.1 in UML

Olivier Carre, Hugues Martin, Jean-Jacques Vandewalle

Gemplus Research Lab.,
BP 100 13881 Gémenos Cedex, France
{ocarre, hmartin, jeanjac}@research.gemplus.com

Abstract. This paper presents a part of a UML model of the Java Card 2.1 specification that describes the security mechanisms. Our goal is to provide developers with a comprehensive documentation and an expressive notation for the security mechanisms of Java Card.

1 Introduction

Due to an important number of standards and specific products, it was very difficult to develop portable applications for different smart cards. In the mid-90's, a new programming language and environment, available for many platforms, has been designed: Java [1], which is based on a virtual machine [2]. In terms of application design and software production, Java Card brings the object oriented programming model that is enforced by the usage of the Java Programming language. Object orientation brings to card programming the same advantages than anywhere else, by encouraging the development of code in small and self contained units. This approach allows modularity, encapsulation and information hiding to card applications. Then it leads to more independence between objects, and a restriction of the interactions between objects to well defined interfaces.

The Java environment is of great interest for the portability and the security of applications thanks to its virtual machine which executes type-safe bytecodes in an interpreted mode. Then, Java was regarded as a possible environment that could be embedded in smart cards. Nevertheless, the Java virtual machine needs a lot of resources to run and smart cards are not powerful enough to support a pure Java virtual machine. So, a stripped down version of Java has been specifically designed for smart cards, the Java Card standard. This standard, in its latest 2.1 release, is defined by a set of three specifications: the Java Card Virtual Machine (JCVM) [3], the Application Programming Interfaces (API) [4], and the Java Card Runtime Environment (JCRE) [5]. With Java Card, developing smart card applications is almost as easy as developing classic Java applets.

Thanks to a wide set of functionality, Java Card 2.1 can be regarded as a complex information system. Moreover, the system formed by the smart card server (the JCRE), the card applets, and the terminal application can rapidly become difficult to comprehend in its entirety. Designing complex information systems, with a lot of interacting parts, can benefit from the use of modeling techniques. The Unified Modeling Language (UML) [6], standardized by the OMG [7], is a language for modeling a system with multiple views expressed in a unified notation. This language is oriented towards the modeling of object-based information systems. Therefore, UML is useful to express views that can help the development and the deployment of such systems. That is why the idea of using UML for modeling Java Card came naturally in order to provide developers with a comprehensive documentation and unified notation for Java Card.

This paper does not provide a complete UML model of the Java Card system. It just outlines the security features of such a system. We provide and discuss some sketches of our model including diagrams for the card applet life-cycle, the object sharing mechanism, and the applet isolation mechanism.

The remaining part of the paper is structured as follows: we first discuss the basic functionality of Java Card 2.1 we have first isolated, and then, the security mechanism of Java Card 2.1. We conclude by discussing about the security of the system.

2 Basic Functionality of the Java Card

One of the main features of the Java Card is to be open. That is to say that it is possible to load and install several applets on a card at any time, even after it is delivered. Thus each applet to be executed must be selected first.

One of the interest of having multiple applets installed into the card is that it is possible for those applets to communicate with each other by exchanging data. In order to perform this, the Java Card provides an object sharing mechanism.

However, communication between two applets may be the source of a security risk. To prevent from unexpected data leaking and references passing from applet to applet, the Java Card has a dynamic security mechanism named firewall, which controls the object sharing. Each time an access to a resource is required, the firewall checks it. If the access is not allowed, the firewall throws a security exception.

In UML, we describe this subset of Java Card with a use case diagram (Figure 1).

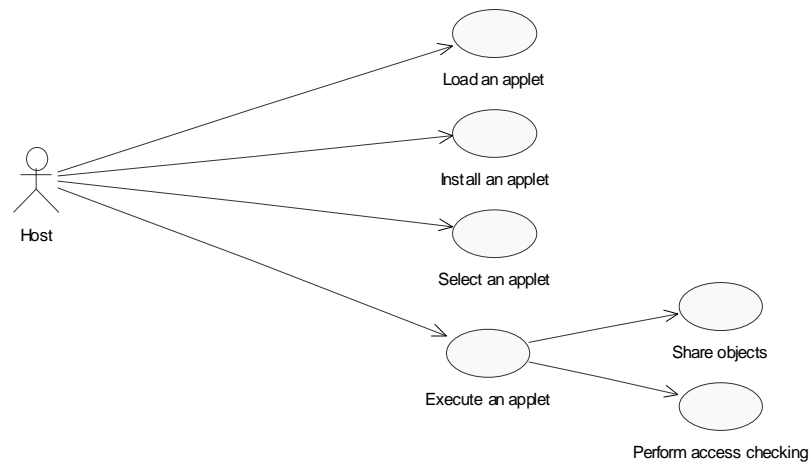


Figure 1: Use Cases involved in the Java Card

From these use cases diagrams, we identify several objects (Figure 2), that may be part of an implementation of the Java Card, and that enable us to implement the previous use cases.

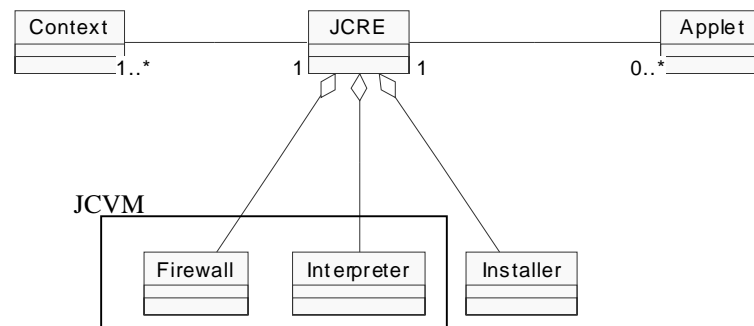


Figure 2: Partial Class Diagram of the Java Card

Classes shown in Figure 2 form a sample of the whole class package of the Java Card. Only the needed classes are described in this article.

The JCRE class is the main class of the Java Card system. This class has a firewall, an interpreter and an installer. The installer is in charge of loading, linking and installing an applet within the card. During the linking process, the installer performs a static checking, which is out of the scope of this article. The applet is loaded into the card as a set of bytecodes. By interpreting the bytecode, the firewall performs the dynamic security checking at runtime.

The rest of this article is devoted to the object sharing mechanism which involves some of the previous identified classes. The firewall mechanism is particularly highlighted.

3 Object sharing in the Java Card

In order to make communicate applets through a strong security system (firewall), the Java Card specification defines sharing mechanisms: the JCRE Entry Point Objects, the global arrays, and the shareable interfaces.

3.1 JCRE Entry Point Object

Without any particular procedure, an object cannot be accessed by objects of different contexts. But, some system features of the JCRE need to be accessed from any context and any object. The JCRE context is not shared, but some methods have been flagged as entry point methods and thus become accessible from any context. There are two categories of entry point objects: temporary and permanent. Due to the design of the Java Card, some data might be stored in persistent memory or in volatile memory. The temporary entry point methods are stored in volatile memory, and the permanent JCRE entry point objects are persistent across card sessions.

Applets and other objects are not allowed to store references to temporary JCRE Entry Point Objects.

3.2 Global arrays

Some objects may require to be accessible from any context and are not owned by the JCRE. It is the case of the APDU buffer. When an APDU command is received by the JCRE, it is stored in a particular array, the APDU buffer. This buffer is accessible from any context. It is a global array.

3.3 Shareable Interfaces

In some cases, objects may require to communicate with other objects. A complete mechanism is provided by Java Card.

Considering two applets in the card: a purse and a loyalty applet. The goal of these applets is that each time a debit is made on the purse, points are granted in the loyalty applet. Thus, the loyalty applet has to share methods with the purse applet in order for the purse to be able to credit the loyalty. The applet loyalty builds a shareable interface. The purse obtains the shareable interface object and then requests services from the loyalty applet. In this case the service is a grant service.

We are not interested in the way the purse and the loyalty applets are implemented. We only need to know is that the loyalty applet makes available its `grantPoints` method for other applets.

We detail hereafter the different steps of this sharing.

3.3.1 Step one: The loyalty applet builds the shareable interface object.

To make its object available for sharing to another applet in a different context, the loyalty applet first defines a shareable interface named `JavaLoyaltyInterface` which extends `javacard.framework.Shareable`. The method defined in the shareable interface represents the service that the applet makes accessible to other applets. The applet then defines a class `JavaLoyalty` that implements the shareable interface. `JavaLoyalty` may also define other methods and fields, but these are protected by the firewall. Only the methods defined in `JavaLoyaltyInterface` are accessible to other applets.

Using UML, we represent this architecture with a class diagram (Figure 3).

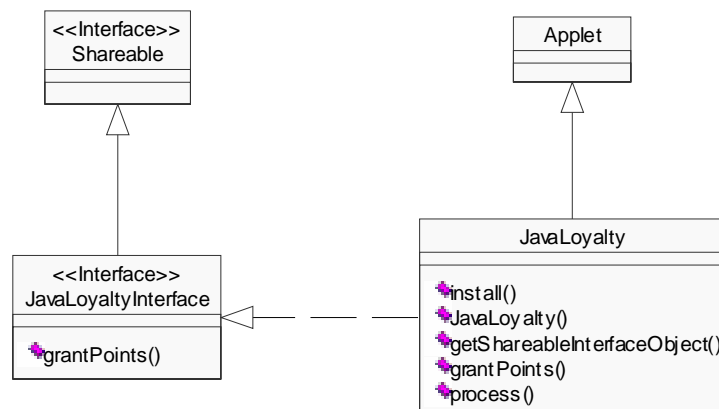


Figure 3: Class diagram of the shareable interface utilization

In this example, the shared method is `grantPoint` which grants points in the loyalty applet. This method is accessible from any context. A context which is a protected space in which the applet is executed is detailed later in this article. The other methods are the constructor (`JavaLoyalty`), and the common methods of every applet (`process` and `install`).

3.3.2 Step two: The purse obtains the shareable interface and requests services from the loyalty

To access loyalty's method, the purse needs to get the reference of the shareable interface. First it creates an object reference of type `JavaLoyaltyInterface`,

and secondly, it gets the reference via the special JCRE method `JCSYSTEM.getAppletShareableInterfaceObject`. When this invocation is received by the JCRE, it invokes the `Applet.getShareableInterfaceObject` method of the loyalty applet, with the AID of the requester (the purse) as parameter. Then the loyalty applet determines if it allows access to the purse. The way the access is allowed is entirely implementer free.

With UML, we can represent this procedure with a sequence diagram (Figure 4).

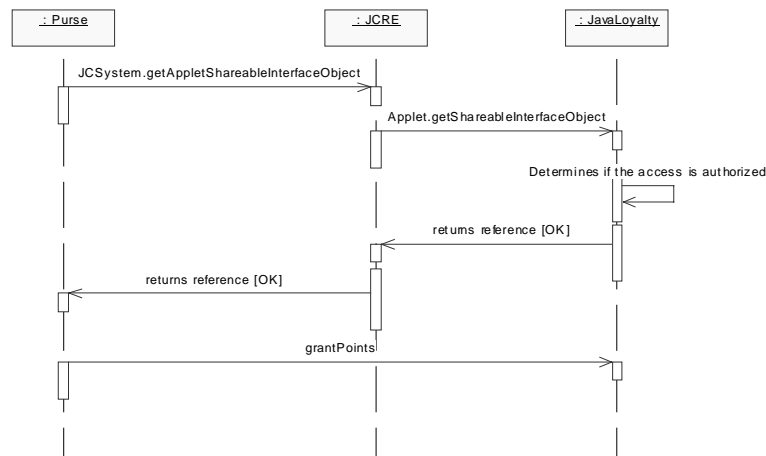


Figure 4: Sequence diagram of accessing a shareable interface

In order to process the object sharing in a secure manner, the Java Card specification defines a security mechanism, the firewall, which supervises and checks the integrity of the sharing at runtime.

4 The Firewall

The firewall is a runtime enforced protection which partitions the Java Card platform's object into separate object spaces called contexts. The firewall is the boundary between a context and another. There is a single context for each applet. The firewall creates and manages each applet context. At any time, only one context is active because the Java Card platform is a single-threaded platform. All bytecodes that access resources are checked in order to determine if the access is allowed. An access to a resource is an access to an object, or to a feature of an object.

When certain well defined conditions are met during the execution, the firewall allows the access from one object to another. In this case, a context switch is performed from the object caller's context to the called object's context. Upon exit from the called

method, a restoring context switch is performed, the original context (of the caller of the method) is restored as the currently active context.

4.1 States of the Firewall

During its lifetime, the firewall has different states. Assuming that the firewall is an object of the JCRE, we use an UML state diagram to represent the different states of the firewall (Figure 5).

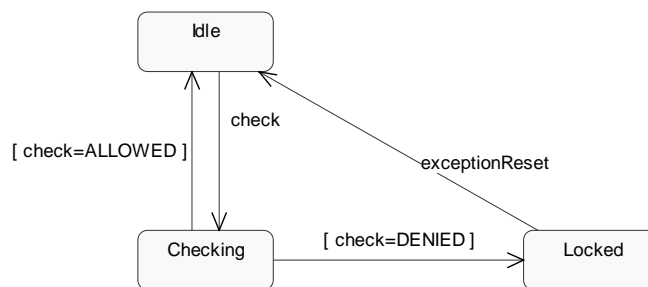


Figure 5: State diagram of the Firewall

The initial state of the firewall is idle. It is waiting for a command. When an applet is executed, the bytecode is interpreted by the virtual machine. For each access to a resource, the interpreter of the virtual machine asks the firewall to check the access. The firewall goes in the checking state in which its activity is to verify if the access is allowed or denied. The checking mechanism is detailed later in this paper. There may be two result of the checking. The access may be allowed or denied. If the access is allowed, the firewall goes back in the idle state and returns a status variable to the interpreter in order to indicate that the access is allowed. If the access is denied, the firewall goes in a locked state and throws a security exception to the JCRE.

Once the firewall is locked, no access can be done until it is reset by the JCRE. If the security exception is caught by an applet, the JCRE resets the firewall to its idle state and the interpretation continues; if the security exception is not caught, the JCRE indicates it to the host application.

4.2 Mechanisms of the firewall

The execution of an applet is made by the interpretation of the bytecode. All accesses made within the card are checked. There are two levels of checking. The first one is a static checking which is made at the loading of the applet by the installer. The second

one is dynamically made bytecode by the firewall during the interpretation of the bytecode.

When the interpreter encounters one of the following bytecode, it asks the firewall to check if the access is allowed. These bytecodes are:

Static field access: `getstatic`, `putstatic`.

Object access: `getfield`, `putfield`, `invokevirtual`, `invokeinterface`, `athrow`, `<T>aload`, `<T>astore`, `arraylength`, `checkcast`, `instanceof`.

In this section, we describe the firewall mechanisms which involve collaborations and interactions between the firewall itself and the different objects within the card. We give an example of the activity done for controlling a bytecode.

The bytecode checking can produce a context switch or not. Thus we isolate two corresponding types of collaboration between the firewall and the other objects: one concerning an access to a static field or a feature within the same object with no context switch produced (Figure 6) and one concerning an access to a shared method producing a context switch (Figure 7).

4.2.1 Checking without a context switch produced

If no context switch is performed, the complete collaboration of the firewall is as

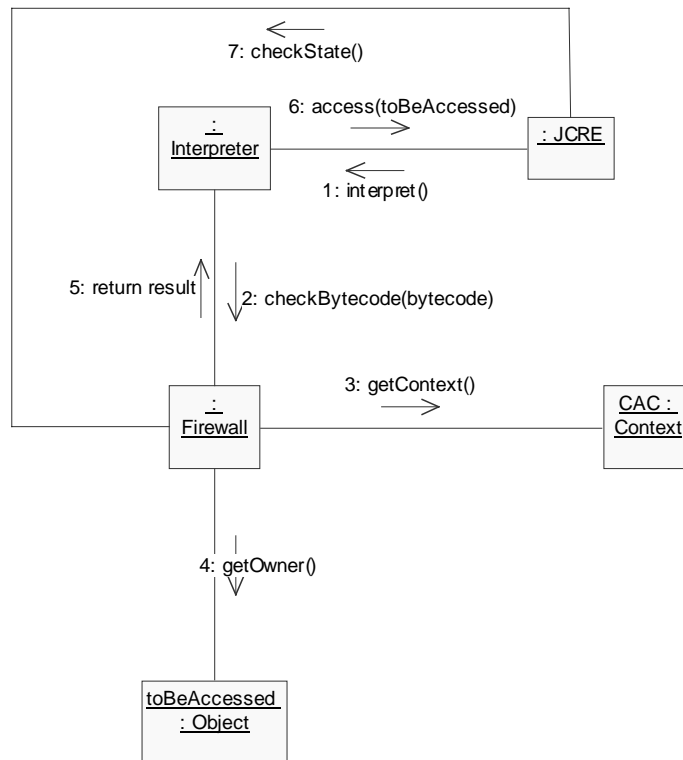


Figure 6: Collaboration diagram of the firewall during a verification without context switching

follow:

The JCRE asks the interpreter to execute an applet. During this interpretation if the interpreter encounters a bytecode which has to be checked, it contacts the firewall to perform the access check. To perform the check, the firewall needs to know which context is the current active context, and the owner of the object to be accessed. A status is returned to the interpreter. The interpreter asks the JCRE to perform the access. Then the JCRE checks the state of the firewall.

4.2.2 Checking with a context switch produced

When a shared interface method has been implemented, applets can access some objects they do not own. For such accesses, the JCRE needs to perform a context switch.

The bytecode `invokevirtual` is used to invoke an interface method. If the current active context is the JCRE, a context switch is required in order to execute the method in its own context, that is the context of the method's owner. If the bytecode `invokeinterface` is used to invoke a shared interface and if the firewall allows the access, a context switch is performed to the called method context.

The next collaboration diagram we describe (Figure 7) illustrates the `invokeinterface` interpreted bytecode. It is the same than the previous collaboration diagram, but a context switch is performed after the firewall allows the access, and before the access is performed. This diagram describes the interactions

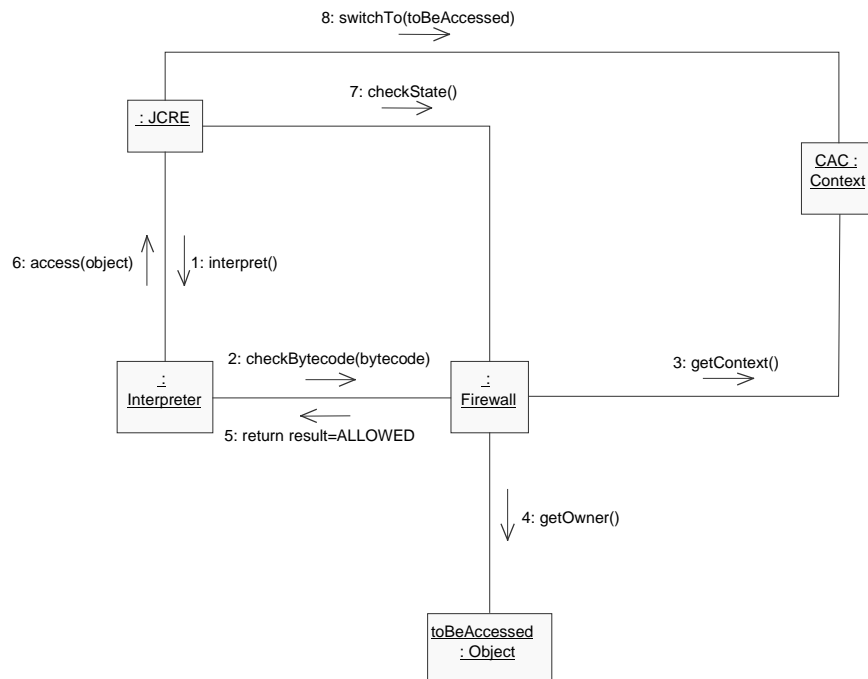


Figure 7: Collaboration diagram of the firewall during a verification with a context switching

between the firewall, the interpreter, the JCRE and the main objects used by the firewall to allow or to deny an access.

The JCRE asks the interpreter to execute some bytecode. During this interpretation, a bytecode that need a special check is encountered. The interpreter asks the firewall to check this bytecode. Before to allow or to deny the access, the firewall needs to know about the currently active context and the owner of the object to be accessed. If the access is allowed (which is the case in this example), the firewall returns a status to the interpreter which indicates that the access is allowed. Then, the interpreter asks the JCRE to perform the access. Before to perform the access, the JCRE checks the state of the firewall. If the firewall is not locked, the JCRE performs a context switch.

4.3 An example of an access checking

We have described previously the interactions between the firewall and other objects when performing an access. The way mechanism by which the access check is made by the firewall is not provided in this paper for the whole bytecode set because there are too many activity diagrams, almost similar. However, each operation is precisely described within the Java Card specification and can be represented with this kind of diagram. In figure 8, we give a sample for one bytecode: the `invokeinterface` bytecode.

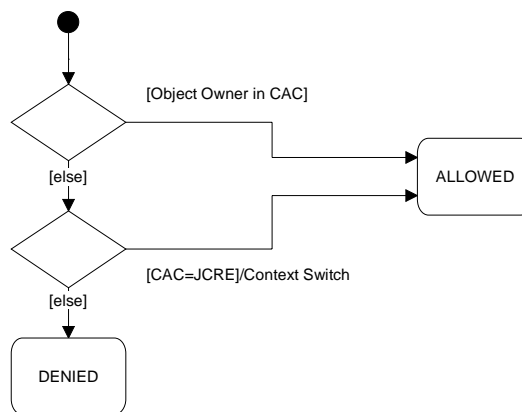


Figure 8: Accessing Standard Interface Methods

The behavior is as follow: if the object is owned by an applet in the current active context, then the access is allowed. If the JCRE is the current active context, then the access also is allowed but in this case, the context is switched to the object owner's context. Otherwise, the access is denied.

5 Conclusion

In this paper, we gave a part of a more complete work done at Gemplus, which is to use UML to provide a more comprehensive vision of the Java Card 2.1. The aim of this paper was to present and to detail the security mechanisms, which are involved in the object sharing and in the firewall. This work helped us clarify these points in order to facilitate their implementation. A next step is to build a formal specification in order to prove properties on these security mechanisms.

The security mechanisms presented in this paper do not cover how an applet provider implements its own security policy while making a shareable object accessible from other applets. Currently, we are working on an extended JCRE that registers the security policy specified by an applet. Then, the JCRE can use this information in order to take into account the defined policy when the firewall is controlling the access from one applet to another applet's object. For this latter design, we also use UML as an expressive notation to represent the objects, their states, and how they interact.

References

- [1] ARNOLD K., GOSLING J.. *The Java Programming Language*. Addison Wesley, May 1996.
- [2] LINDHOM T., YELLIN F.. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley, Sept. 1996.
- [3] SUN MICROSYSTEMS, INC. *Java Card 2.1 Virtual Machine Specification*. Final Revision 1.0 ed., Feb. 1999.
[<http://java.sun.com/products/javacard/JCVMSpec.pdf>]
- [4] SUN MICROSYSTEMS, INC. *Java Card 2.1 Application Programming Interface*. Final Revision 1.0 ed., Feb. 1999.
[<http://java.sun.com/products/javacard/html/doc/index.html>]
- [5] SUN MICROSYSTEMS, INC. *Java Card Runtime Environment 2.1 Specification*. Final Revision 1.0 ed., Feb. 1999.
[<http://java.sun.com/products/javacard/JCRESpec.pdf>]
- [6] BOOCH G., RUMBAUGH J., JACOBSON I. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.
- [7] OBJECT MANAGEMENT GROUP. *Unified Modeling Language*. Version 1.3 R5. March 1999.
[<http://www.rational.com/uml/>]
- [8] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *International Standard ISO/IEC 7816: Integrated circuit(s) cards with contact, parts 3 and 4*. 1987-1998.