Secure Object Sharing Development Kit for Java Card

Daniel Perovich
`Daniel.Perovich@sophia.inria.fr`

INCO - PEDECIBA
Facultad de Ingeniería
Universidad de la República
Uruguay
`http://www.fing.edu.uy/inco`

Project Lemme
Sophia Antipolis
INRIA
France
`http://www-sop.inria.fr/lemme`

February 22, 2002

Abstract

Nowadays, Java Card platform-based Smart Cards are multi-application and support inter-applet collaboration. The Java Card framework enforces applet isolation by means of the applet Firewall to prevent highly sensitive data in one applet to be leaked to another. The framework provides the Shareable Interface Object mechanism to allow developers to share services through the Firewall protection. The working of the mechanism presents serious flaws, which have been addressed and partially solved in work we shall in turn discuss in this paper. We present the Secure Object Sharing Development Kit, which constitutes a programming setting for the formulation of inter-applet collaboration. Its conception elaborates on the solutions proposed for improving the Shareable Interface Object mechanism, which can be applied, and even enriched, when implementing cooperating applets using the framework provided by the kit. We also discuss challenge/response authentication mechanisms, which are a basic ingredient of the various sharing mechanisms presented in this work.

1 Introduction

A Java Card is a Smart Card capable of running programs developed for a subset of the well-known Java platform, which presents additional technology-specific features. The Java Card architecture consists of the Java Card Virtual Machine (JCVM), the Java Card Application Programming Interface (JCAPI) and Industry Add-On classes. The JCVM is built on top of a specific integrated circuit and a native operating system implementation, and it is made of two separate components, namely the *converter* and the ByteCode *interpreter*. The converter runs on a workstation. It loads and pre-processes the class files that make up a Java package and outputs a cap (converted applet) file, which is then loaded on a Java Card Smart Card. The ByteCode interpreter runs on the Java Card and interprets the ByteCode in the cap files. The JCAPI[10] provides framework classes and interfaces for the core functionality of a Java Card application. The Java Card Runtime Environment (JCRE) [11] consists of the Java Card system components that run inside a Smart Card, namely the JCVM, JCAPI, system classes, the installer application and industry-specific extensions. The JCRE is responsible for card resource management, communication, applet execution, and on-card system and applet security [2]. Java Card applications are called *applets*. They are implemented by extending the `Applet` base class provided in the Java Card framework. Applets are loaded and installed in a Java Card. When an applet is registered (during installation) to the JCRE, it indicates the *applet identifier* (AID) it will use. The AID is unique within a Java Card, so if the AID is already in use the applet registration will fail [11].

Applets from different vendors can coexist in a single card and additional applets can be loaded after card manufacture. An applet usually stores highly sensitive information, so sharing this information among applets must be carefully limited. In the Java Card platform applet isolation is achieved through the applet Firewall [11] mechanism. It confines an applet to its own designated area, thus each applet is prevented from accessing fields and operations of objects owned by other applets. The applet Firewall partitions the Java Card object system into separate protected object spaces called *group contexts*. The FIREWALL can be considered as the boundary between one context and another. Each JAVA package is assigned a group context, thus applets in the same JAVA package share the group context. Applets are allowed to access objects of applets in the same group context. However, applets residing in different group contexts will not be able to access each other's objects.

To support cooperative applications on a single card, the JAVA CARD technology provides well-defined sharing mechanisms. These mechanisms are the JCRE Privileges, JCRE Entry Point Objects, Global Arrays and the Shareable Interface Object mechanism [11]. The first three mechanisms are used for JCRE-applet interaction, while the last one is intended to provide inter-applet collaboration.

The working of the Shareable Interface Object mechanism presents serious flaws, which make it possible to develop inter-applet cooperation risking applet impersonation or unauthorized use of resources. In addition to this, the mechanism prevents the number of recipients of a server applet services from being incremented after deployment. These problems have been put forward in [6], and two approaches to solve them have been presented in [6, 8]. These approaches base their solution on a challenge/response mechanism for client authentication. This mechanism relies on a shared knowledge between the client and the server. However, the challenge/response mechanism does not fit well in commercial JAVA CARD applications where services are sold to clients and the server cannot trust that this client will keep secret the shared knowledge. Furthermore, these two approaches present some drawbacks, namely, changes needed in the JCRE specification in one of them, and the lack of dynamic checks at service request in the other.

This paper begins by presenting an overview of the disadvantages of the Shareable Interface Object mechanism. After, it comments on the two approaches to address the problems and their drawbacks. Then, it focuses on the Secure Object Sharing Development Kit. The Kit represents a new approach to overcome the problems and enriches the JAVA CARD inter-applet collaboration mechanism. It is strongly based on the approach introduced in [8] and solves the drawbacks that it presents.

Section 2 introduces the JAVA CARD Shareable Interface Object mechanism and its disadvantages. Section 3 presents overall comments on these drawbacks while Section 4 overviews the two approaches already proposed. Section 5 makes considerable remarks on the challenge/response authentication mechanism and proposes a methodological solution for solving the problem it presents. Later, Section 6 presents the Secure Object Sharing Development Kit. Finally, Section 7 concludes.

2 Shareable Interface Object Mechanism

The Shareable Interface Object mechanism is the only sharing mechanism in the JAVA CARD platform intended for inter-applet collaboration. The Global Array mechanism can be used to pass data from one applet to another, but service request cannot be implemented using it.

For two software components to interact, it is needed to define in which way one of them will request services, and also how this component will get access to the provider which resides in the other component. The first requirement is met by defining the interface (with the set of operations, i.e., services) that the client component should use. For the second requirement, static fields and/or operations can be used. Instead of this, a global component which knows all components in the system can be used. The client requests a provider of the desired services by asking the global component for it.

In the JAVA CARD platform there is an extra requirement: the FIREWALL mechanism must be bypassed so as to share services among applets. The Shareable Interface Object mechanism provides this functionality.

Subsection 2.1 presents the working of the Shareable Interface Object mechanism, while Subsection 2.2 indicates it disadvantages.

2.1 Working of the SIO Mechanism

The `javacard.framework` package provides a tagging interface called `Shareable`, and any interface which extends the `Shareable` interface will be considered as a `Shareable` interface. Requests for services to objects whose class implements a `Shareable` interface are allowed by the FIREWALL mechanism.

When a server applet wants to provide services to other applets within the JAVA CARD, it must define the services it wants to export in an interface tagged as `Shareable`. Using this kind of interface, two of the software interaction requirements are solved, i.e., a client applet must use this interface to request services from the server applet, and the FIREWALL protection will be bypassed as the interface is tagged as `Shareable`.

Within the JAVA CARD, only instances of classes are owned by applets (i.e., are confined to a group context), classes themselves are not. No runtime check is performed when a static field is accessed or when a static operation is invoked. This means that static fields and operation are accessible from any applet; however, objects stored in static fields belong to the applet which instantiates them.

The server applet may decide whether to publish its Shareable Interface Objects (SIOs) in static fields, or return them in static operations. Additionally, the JAVA CARD platform provides a special component called `JCSystem`. This component provides *white pages*[1] functionality for services exported by applets. The `JCSystem.getAppletShareableInterfaceObject` operation can be used by client applets to obtain a reference to a service provider from a server applet, i.e., a reference to an object implementing the desired `Shareable` interface. To share services using the white pages functionality the server applet must override the `getShareableInterfaceObject` operation of the base `Applet` class, and return, within this method, the object implementing the requested interface. Client and server identification is based on their AID. The `JCSystem.getAppletShareableInterfaceObject` operation receives the server's AID and the `getShareableInterfaceObject` receives the client's AID.

Applet Interaction. Figure 2.1 shows the UML[7, 9] Sequence Diagram for getting the desired reference using the `JCSystem.getAppletShareableInterfaceObject` service. The client applet `ca` invokes the `getAppletShareableInterfaceObject` static operation of the `JCSystem` class passing the server's AID and a option parameter indicating the desired service. The `JCSystem`, in turn, invokes the `getShareableInterfaceObject` of the corresponding server applet `sa` passing the client's AID and the option parameter received. The `JCSystem.getAppletShareableInterfaceObject` operation may return `null`, so it is not warranted that the desired object will be obtained after an invocation to this service. What is more, the returned object is such that its class implements the `Shareable` interface, so the client applet has to cast this reference to the interface defined by the server applet. Neither is it warranted that the object returned can be successfully cast into the specific `Shareable` interface the client applet expects.

Figure 2.2 shows the UML Sequence Diagram that describes how a client applet `ca` gets the desired reference using a public static operation. Note that this diagram does not show the server applet making public the SIO. This mechanism has an advantage on the previous one: the server applet `getSIO` operation can receive as many parameters as needed, and also it can have a more adequate return type (`SI` in the Figure). Notice, however, that `null` can be also returned.

2.2 Drawbacks of the SIO mechanism

Improved security is one of the most relevant characteristics for preferring JAVA CARD systems. Even though, the working of the Shareable Interface Object mechanism presents important drawbacks, and some of them are considerable security problems.

[1]The white pages communication mechanism is such that the client knows the service it needs (denoted by the server applet's AID and the `byte` parameter), but not where this service is located (which is the instance of the `Shareable` Interface that provides this service) [5].

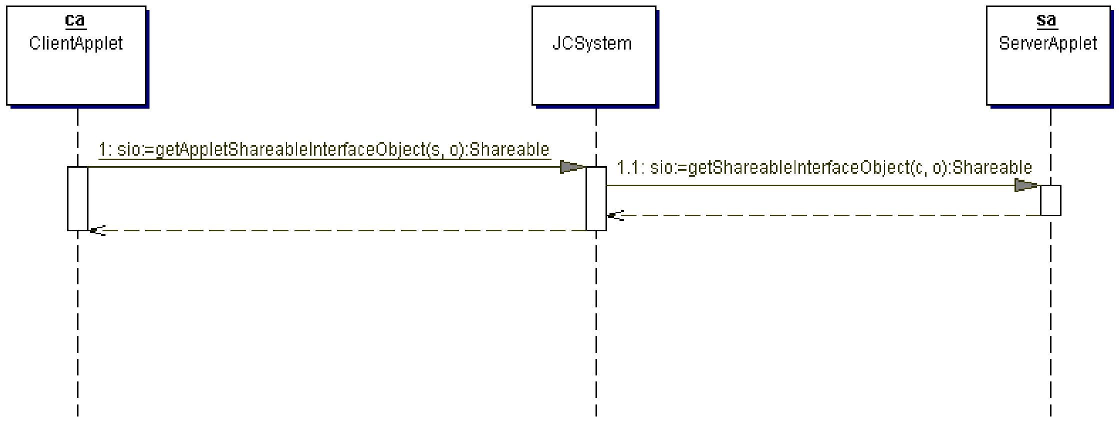

ca
ClientApplet
JCSystem
sa
ServerApplet
1: sio:=getAppletShareableInterfaceObject(s, o):Shareable
1.1: sio:=getShareableInterfaceObject(c, o):Shareable

Figure 2.1: Service Request using the `JCSystem` class

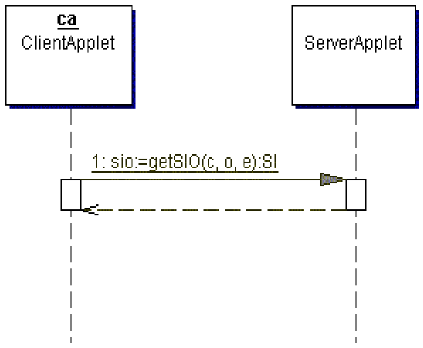


Figure 2.2: Service Request using a static operation

These problems were initially put forward in [6]. This work introduces four disadvantages, namely Impossibility for Passing Objects as Parameters, Inappropriate Casting, Applet Impersonation and Limited Clients. We present them in what follows. Section 3 presents overall comments on them while Section 4 overviews the two approaches [6, 8] for overcoming these problems.

Impossibility of Passing Objects as Parameters. If an applet gets a reference to an object that belongs to an applet in another group context, it will not be able to use its services due to the FIREWALL. If the operations within a `Shareable` interface passes objects as parameters or returns an object, it is practically useless as the other applet will not be able to use the services provided by these objects. In spite of this, preserving the reference and checking equality with other references can be a valid use.

Inappropriate Casting. It is also possible for a client applet to access a SIO for which it is not authorized. This may happen only if a class implements more than one `Shareable` interface, namely `SIA` and `SIB`. A reference to an instance of `SIA`, which has been legitimately got by a client applet, can be cast to a reference of `SIB`. The FIREWALL will allow the use of services provided by `SIB` as these services are defined in a `Shareable` interface.

Applet Impersonation. When requesting a SIO, client authentication is based on its AID. Then, a malicious applet could be installed with the same AID of a valid client, in a compromised card, and thus gain access to restricted data. Other proposals have been made to restrict applet loading so as to prevent this kind of attacks [4]. These proposals are based on annotated code for checking information flow by static or dynamic analysis.

Limited Clients. Another problem with this selection criterion is that the server applet must know the AID of every possible client, so as to authorize access to its SIOs. This would make it impossible to allow access to new valid clients once the server applet has been deployed and widely used.

3 Addressing the Drawbacks

The Applet Impersonation and Inappropriate Casting problems represent important security breaches. A malicious applet might make use of this problems to gain illegitimate access to restricted data or services. Neither the Limited Client problem nor the Impossibility of Passing Objects as Parameters are security matters. However both represents a considerable disadvantage of the mechanism. Limiting the number of client applets once the server applet is deployed implies re-deploying the server applet when a new collaborating client applet is developed. Moreover, the developer must be aware of passing objects as parameters as the FIREWALL restrictions will apply when using these objects. This problem is detected at runtime when a `SecurityException` is thrown, and usually debugging applets is a very hard task.

In what follows, we give overall comments for addressing the drawbacks of the Shareable Interface Object Mechanism presented in the previous Section.

3.1 Objects as Parameter

As we mentioned before, FIREWALL restrictions applies to object references passed or received in operations defined within a `Shareable` interface. Nevertheless, it is not impossible to pass usable objects as parameters. It can be done by defining `Shareable` interfaces for the services provided by the objects which are passed as parameters, and making their classes implement these interfaces. Notice that, as in usual applications, the callee (or the caller when considering the returned object) can hold a reference to the objects it receives. This means that these objects could be used at any time by the other applet, and also could be passed to a third applet as a parameter, and the FIREWALL would allow the usage of them. For security reasons, the client should first be authenticated using any of the approaches presented in Sections 4 and 6. When a complex data structure (i.e., non primitive data) needs to be passed as parameters, the standard solution is to develop a class only with query operations[2]. These classes should implement a `Shareable` interface and their instances should be reused for each invocation in which they are passed as parameters. It would be a good practice to put the fields to their default value, because the callee can hold a reference to this object, and, in this case, it would be able to read the information passed as a parameter to another applet.

As an example of passing object as parameters, consider two applets communicating using an Event Triggering mechanism. The triggerer applet provides a `Shareable` interface for listener registration. A listener applet will register an object using this interface. When the event is triggered, the triggerer applet notify all registered listeners, i.e., invokes an operation of the object which has received as a parameter. Both register and listener interfaces must be defined as `Shareable`.

Figure 3.1 shows the UML Sequence Diagram in which the `ListenerApplet` registers a listener to the `TrigererApplet`. First, the listener applet `la` requests a reference to the `Registerer` instance of the `TrigererApplet`. Once it gets the reference, it simply calls the `register` operation passing

[2]A query operation is such that has no side effects.

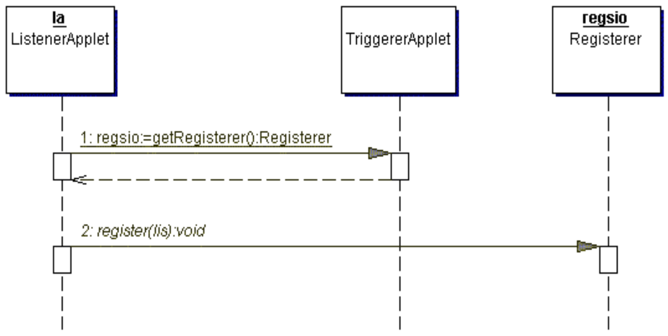


Figure 3.1: Listener Registration

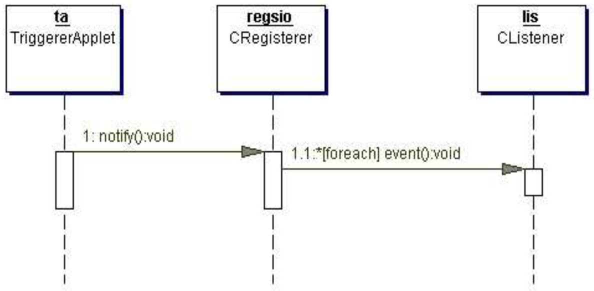


Figure 3.2: Event triggering (notification)

its `Listener` instance as a parameter. Notice that `Listener` is a `Shareable` interface in order to allow the `TrigererApplet` to call its operations.

Figure 3.2 shows the UML Sequence Diagram for notifying all listeners that an event takes place. The `TrigererApplet` invokes the `notify` operation of the `CRegisterer` instance (`regsio` in the Figure is an instance of `CRegisterer`; this class implements the `Registerer Shareable` interface and manages all registration issues), and this method invokes, in turn, the `event` operation of all registered listeners.

3.2 Inappropriate Casting

The Inappropriate Casting problem is present when a class implements more than one `Shareable` interface. Notice, however, that it is not enough to be sure that a class does not directly implement two `Shareable` interfaces. There are other cases in which a class can be implementing more than one `Shareable` interface. These cases are discussed in what follows.

Super class implements a Shareable interface. The Inappropriate Casting problem is present also when one of the super classes implements a `Shareable` interface other than the one being implemented by the class itself. Suppose we have two `Shareable` interfaces `SIA` and `SIB`, and two classes `CA` and `CB` implementing `SIA` and `SIB` respectively. If `CA` and `CB` are in the sub-classing relation, i.e., `CA` is a direct or indirect subclass of `CB` or vice-versa, we face the Inappropriate Casting problem. Figure 3.3(a) sketches out this scenario.

Extension of a Shareable interface. Another scenario when the problem is present is when an interface extends a `Shareable` interface. Suppose we have a `Shareable` interface `SIC`, and another interface `SID` extending it. All classes implementing `SID` will be implementing both `SIC` and `SID`.

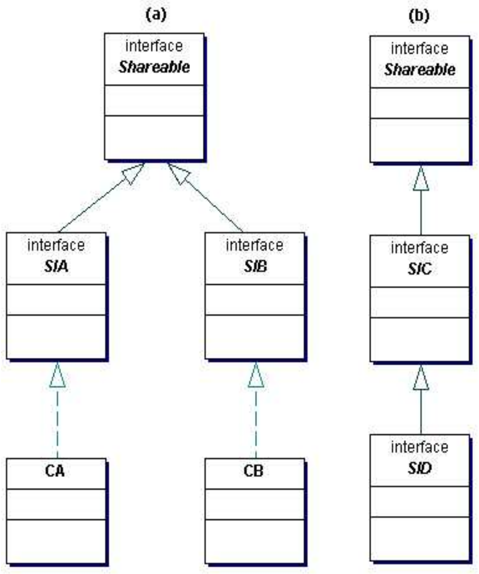


Figure 3.3: Inappropriate Casting Problem

The Inappropriate Casting problem is also found in this scenario as these classes can be casted from `SIC` to `SID` gaining access to extra services. Figure 3.3(b) illustrates it.

3.3 AID-Based Authentication

The Applet Impersonation and Limited Client problems are a consequence of basing applet authentication on the AID. The JCRE Specification [11] indicates that:

> *"A server applet to be invoked from another applet needs to override this method*[†]*. This method should examine the* `clientAID`[††] *and the* `parameter`*. If the* `clientAID` *is not one of the expected* AID*s, the method should return* `null`*. Similarly if the* `parameter` *is not recognized or if it is not allowed for the* `clientAID`*, the the method also should return* `null`*. Otherwise, the applet should return a SIO of the* `Shareable` *interface type that the client has requested."*

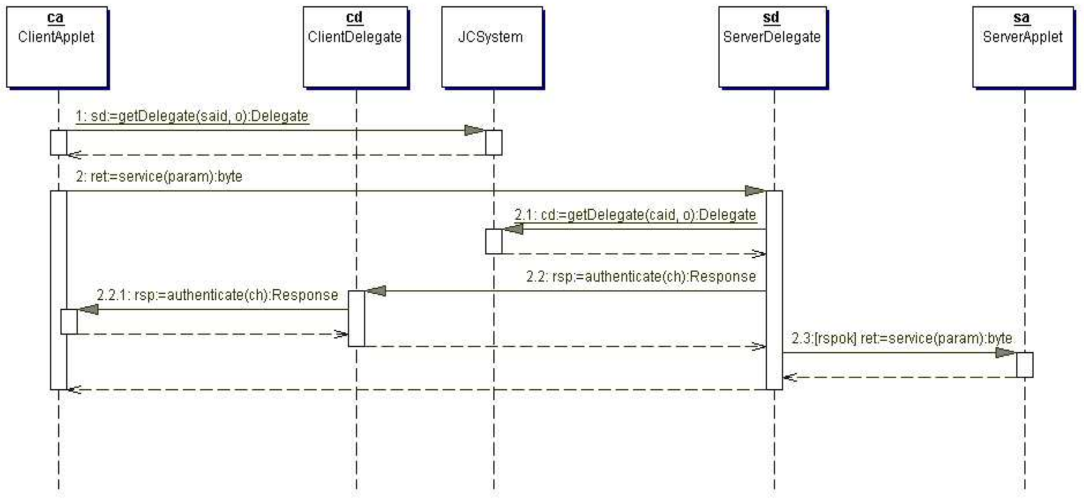
[†] *"this method"* refers to the `Applet.getShareableInterfaceObject`operation in the base `Applet` class.

[††] *"*`clientAID`*"* refers to the `AID` parameter of this operation.

Thus, the intended use of the `AID` parameter is for validating the client applet's rights for accessing the desired services. However, this problems can be overcome by using a challenge/response sequence for client authentication. Section 4 presents an overview of two already proposed approaches which have based their solution in the challenge/response mechanism. Later, Section 5 introduces the problem of using such a mechanism in a commercial environment.

4 Approaches to Solve the Problems

This Section details two proposed solutions for addressing some of the drawbacks of the Shareable Interface Object mechanism. These approaches focus mainly on Applet Impersonation and Limited Clients.



ca
ClientApplet
cd
ClientDelegate
JCSystem
sd
ServerDelegate
sa
ServerApplet
1: sd:=getDelegate(said, o):Delegate
2: ret:=service(param):byte
2.1: cd:=getDelegate(caid, o):Delegate
2.2: rsp:=authenticate(ch):Response
2.2.1: rsp:=authenticate(ch):Response
2.3:[rspok] ret:=service(param):byte

Figure 4.1: Interaction using the Delegate Object Approach

4.1 Delegate Object Approach

The solution introduced in [6] relies on the existence of a *Delegate Object*. A delegate object is an object tagged as delegate, and it is registered when the applet registers itself. The JCRE preserves the reference to the delegate object of each applet. This object provides all applet's services, and they can be both fields and operations. Access to delegate objects would be granted to any applet requesting it, and the delegate object handles all security concerns. The security mechanism proposed is the use of a challenge/response sequence occurring at each method invocation. Using challenge/response both the Applet Impersonation and Limited Client problems are solved as client authentication is not based on AIDs. The challenge/response mechanism can differ in the function it uses to convert challenges to responses on a per method basis.

Additionally, within this framework all shared services are managed by the delegate object, so a client cannot gain unauthorized access by Inappropriate Casting, as no `Shareable` interfaces is longer needed.

Interaction. Figure 4.1 shows the UML Sequence Diagram for the interaction of a client applet `ca` requesting a service from a server applet `sa`. The client applet gets the server's delegate object (`sd`) and requests the service. The server's delegate object gets the client's delegate object (`cd`) and goes through the authentication process, i.e., issues a challenge (`ch`) and gets the response (`rsp`). The client's delegate object asks the client applet for the response to the challenge and returns it. The server's delegate object checks the response, and if it is correct, requests the desired service (`byte service(byte)` in the Figure) from the server applet.

Changes to the JCRE. The Delegate Object approach requires changes to the JCRE specification. A delegate object is registered with the applet so the `register` operation of the base `Applet` class must be changed. Also the `JCSystem` must preserve the reference to each delegate object. The delegate object can be used by any client applet, thus the applet FIREWALL must be changed so as not to restrict the usage of services of delegate objects. Note that the current FIREWALL does a similar task with objects implementing a `Shareable` interface.

Drawbacks. This approach presents compatibility issues with the existing JAVA CARD hardware and software. It eliminates the Shareable Interface Object mechanism as legitimate access through separate contexts is provided by the delegate object. Therefore, current JAVA CARD applications that rely on the Shareable Interface Object Mechanism will not be functional on hardware based on this approach. In addition, JAVA CARD applications developed on this approach will not be able to run on the current hardware as the approach needs to change the current JCRE specification.

Another drawback, from Software Engineering, is that all exported services in one applet must be provided in one object, namely its Delegate Object. Additionally, the approach limits the possibility of passing complex data structures from one applet to another, as only Delegate Objects can be used through the FIREWALL.

Furthermore, this approach requires that the client goes through the authentication process at each service request. This leads to a more obscure client source code and to a performance penalty.

4.2 Methodological approach

The Methodological approach has been presented in [8], and has been proposed as an alternative to the Delegate Object approach. Its main goal is to solve the same problems but using the current JCRE specification. It is based on similar basic concepts of the Delegate Object approach, as it uses challenge/response authentication, and provides fine-grained control over access to the applet's services.

Main concepts. This methodology is based on the existence of an object in the server applet's package, called `SecureSIO`. This object is an instance of a class which implements a `Shareable` interface called `SecureSI`. The `SecureSI` interface provides operations for authentication, namely `getChallenge` and `setResponse`. The case study in [8] presents an authentication mechanism using a `short` challenge/response, but it is clear that a more complex mechanism can be used instead.

The server applet contains an `AuthorizationManager`, which keeps record of all registered clients together with the SIOs they can access during a session. Both the `SecureSIO` and the `AuthorizationManager` manage all security concerns within the server applet. The registration lasts for one session as the `AuthorizationManager` stores the information of the registered clients on a transient `Object` array.

Development and Interaction. The methodology suggests to create the `SecureSI` interface which defines the authentication mechanism, and to create a class `SecureSIO` which implements it. An `AuthorizationManager` class must be implemented for maintaining the set of registered clients. The `getShareableInterfaceObject` operation of the server applet must be implemented in order to return the `SecureSIO` if the client applet is not registered to the desired service. If the client applet has already registered the corresponding SIO is returned. Figure 4.2 shows the UML Sequence Diagram for the interaction using the Methodological approach. The client applet `ca` requests an object whose class implements the `SecureSI` interface. Then the client applet asks for the challenge and issues the response. If the response is correct the `SecureSIO` registers the client in the server applet. After registration, the client applet requests for an object whose class implements the desired `Shareable` interface.

Remarks. The Applet Impersonation and the Limited Clients problems are solved, using this approach, as client authentication is based on the challenge/response mechanism. The Inappropriate Casting is solved by taking into account the remarks presented in Subsection 3.2.

The advantages of the Methodological approach are that it does not require any changes to the JCRE, so it runs on existing hardware. Also, the Shareable Interface Object mechanism is preserved, so passing objects as parameters can be done as explained in Subsection 3.1.

The drawbacks of this approach are that the developer must implement the mechanism for each server applet, so there is extra code (and a bigger CAP file size) in each package. In addition, extra

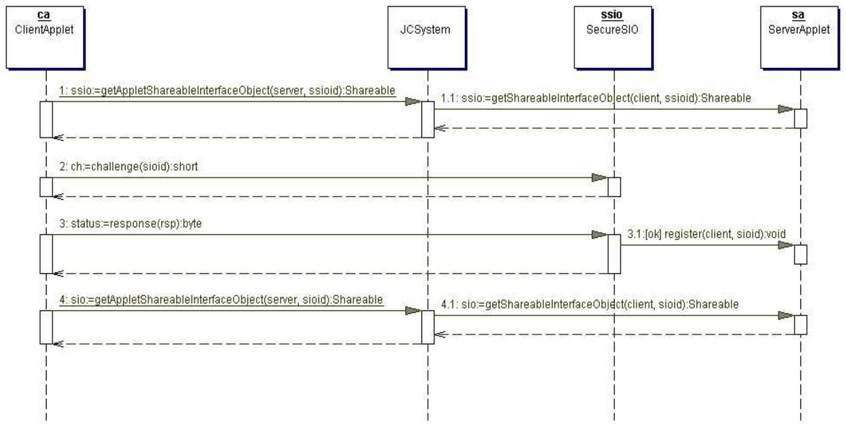

ca
ClientApplet
JCSystem
ssio
SecureSIO
sa
ServerApplet
1: ssio:=getAppletShareableInterfaceObject(server, ssioid):Shareable
1.1: ssio:=getShareableInterfaceObject(client, ssioid):Shareable
2: ch:=challenge(sioid):short
3: status:=response(rsp):byte
3.1:[ok] register(client, sioid):void
4: sio:=getAppletShareableInterfaceObject(server, sioid):Shareable
4.1: sio:=getShareableInterfaceObject(client, sioid):Shareable

Figure 4.2: Interaction using the Methodological Approach

memory is needed for administrative information within the `AuthorizationManager`. Moreover, the `AuthorizationManager` is just used to check if a client can get a SIO, but it is not used to check the usage of a SIO after the client applet gets the reference. Once a client applet gets the reference to a SIO, the client can hold this reference and use it later, even after card reset.

5 Challenge/Response Problem

The authentication mechanism used by the two approaches presented in Section 4 are based on a challenge/response sequence. For simplicity, it can be used just a shared key, i.e., a constant function from challenges to responses. Therefore, the client applet must answer the secret key, independently of the challenge. More complex functions can be used to convert challenges to responses, and also the data structure of both challenges and responses can be made very complicated.

Independently of the function and the data structure used, the challenge/response mechanism is secure enough if in the environment in which it is used, the server can trust the client. In what follows, we introduce the problem that this mechanism presents when confidentiality cannot be expected in the client. Later, we propose a methodological solution to overcome it. This solution is an alternative client authentication procedure which solves the problem, even though it is still vulnerable.

5.1 The Problem

The challenge/response mechanism relies on a shared knowledge between both the client and the server. This knowledge is the way a challenge is computed to generate a response. However, this authentication mechanism is not adequate in contexts where server's services are being sold to clients, or when the server cannot trust the client.

If the company developing the server applet sells the secret to a company which develops the client applet, the former need to be sure that only this client company is going to use this secret, i.e., that the client company will not make public (or resell) the secret. The main problem here is that the server applet can be already deployed, so it may be too late to include a new secret in the server applet when the company detects that the secret was made public.

The following Subsection presents a methodological solution for this problem, in which at most one client in each card will use the service sold.

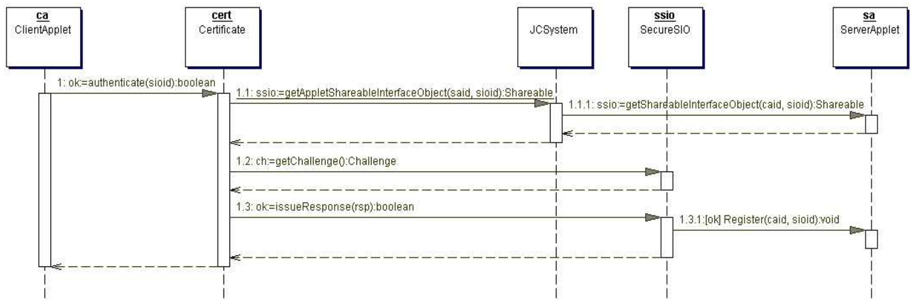

ca
ClientApplet
cert
Certificate
JCSystem
ssio
SecureSIO
sa
ServerApplet
1: ok:=authenticate(sioid):boolean
1.1: ssio:=getAppletShareableInterfaceObject(said, sioid):Shareable
1.1.1: ssio:=getShareableInterfaceObject(caid, sioid):Shareable
1.2: ch:=getChallenge():Challenge
1.3: ok:=issueResponse(rsp):boolean
1.3.1:[ok] Register(caid, sioid):void

Figure 5.1: Challenge/Response Authentication Procedure

5.2 Methodological Solution

A simple solution to this problem is by limiting the amount of clients, i.e., if the service is sold twice, it accepts at most two clients to this service. However, using this solution we face again the Limited Client problem mentioned before.

The solution we propose is a bit more complex. The developer of the server creates a class called `Certificate`. If a client applet holds a certificate of this kind, the server applet will allow access to its exported services. This certificate shares a secret with the server applet (using a challenge/response mechanism). Notice that the `Certificate` class is developed by the server applet's developer, and its secret is never published nor given explicitly to the client applet's developer. The special consideration here is that this class must be instantiated at most once, so as to prevent more than one client from using this kind of certificates. This is simply done by using the Singleton Design Pattern [3] in the `Certificate` class. This pattern suggests to have one private constructor in the class, and a `getInstance` static operation that returns the value of a private static field. If the field is `null`, the `getInstance` operation creates an instance and stores it in this field. Then, in any case, the field value is returned. Using this pattern, the only possibility to get an instance is by this operation, and at most one instance will be created. Notice also that as the `getInstance` operation is static, the `Certificate` instance will be created in the client's context. When the client applet creates the certificate, the latter registers the client applet's AID. Future authentication requests will be allowed only from the client applet context, as the `Certificate` instance preserves the original AID.

A package with the `Certificate` class is sold to the client's developer. This class is final, so no subclasses can be made of this class, and also it does not implement the `Shareable` interface. Because of that, only the client applet will be able to use this instance as it will be created in its own context. The client's developer can resell or make public this package, but only one of them (the seller or the buyer) will be allowed to use the certificate within a Java Card[3].

Using this approach, the authentication mechanism is different to the one used in the Methodological approach. First of all, the client applet creates an instance of the certificate by invoking the `getInstance` static operation of the `Certificate` class. At registration time, the client applet just requests the certificate to authenticate to the server. The certificate is responsible for the whole authentication procedure, usually a challenge/response sequence with the server. Figure 5.1 shows the authentication procedure when using this approach. Later, the client applet can get the desired services as it is already registered.

Notice that the security mechanism in the server applet is still needed so as to prevent any developer from developing fake certificates.

[3]Note that it is possible, in this scenario, that in one Java Card a client applet uses this certificate package and in another Java Card a different client uses it.

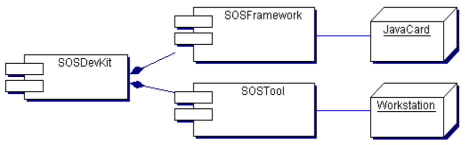


Figure 6.1: SOS Development Kit

Remarks. The challenge/response solution is still vulnerable. The possibility of inspecting a CLASS or CAP file makes it possible for the client developer to extract the shared secret used by the server and the certificate. The proposed solution can be reformulated in order to sell the client only a package with interfaces. In this case, the CLASS file would not contain the secret, so it would be useless to inspect its internal structure. This package will be then linked to another package provided and managed by the server's developer. Future work is to reformulate the solution taking into account this remark.

6 Secure Object Sharing Development Kit

The Secure Object Sharing Development Kit for the Java Card platform provides the developer a set of tools for using a Secure Object Sharing mechanism in his/her Java Card applications. The main goal of the Kit is to lighten the developer's work by providing an Object Sharing mechanism in which the security concerns are already handled.

The SOS Development Kit consists basically of a framework for Object Sharing in Java Card, and a set of tools which statically checks desired properties of the developer's applets. Figure 6.1 shows the overall structure of the Kit, and the intended use for each of the subcomponents.

Subsection 6.1 introduces the Secure Object Sharing Framework, while the set of tools is presented in Subsection 6.2.

6.1 SOS Framework

The Secure Object Sharing Framework is a software component which resides in a Java Card (see Figure 6.1). It is based on the Methodological approach and its main goal is to take care of all security concerns when an applet needs (wants) to export some services. It addresses the problems found in the Methodological approach as the SOS Framework needs to be loaded only once in the Java Card, and the developer's applet will be linked with it inside the card.

Figure 6.2 shows the internal design of the Secure Object Sharing Framework software component. A new applet class called `SOSApplet` is the most important class within the component. This class manages the security information in its `AuthorizationManager` and provides services to the subapplets[4] by the `ISIOResource` interface and to other loaded applets in the card by the `ServerAdminSI` interface.

The Framework can be understood as a redefinition of the Shareable Interface Object mechanism. Notice however, that what is redefined is the way a server applet manages incoming requests for Shareable Interface Objects (SIOs), not the way a client applet requests them.

The following provides a detailed account of each member of the Secure Object Sharing Framework.

6.1.1 SOSApplet base class

The `SOSApplet` class is a new base class for Java Card applet development. It extends the `javacard.framework.Applet` class and enriches the Shareable Interface Object mechanism.

[4]The term *subapplet* is used here to refer to the classes which extends the `SOSApplet` class.

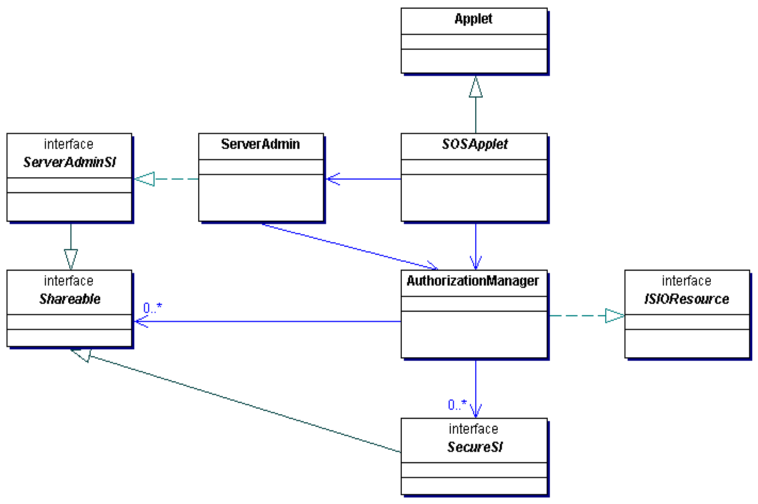


Figure 6.2: Secure Object Sharing Framework Internal Design

Figure 6.2 shows the internal design of the Secure Object Sharing Framework. The `SOSApplet` maintains an instance of the `AuthorizationManager` class to record client authentication, and an instance of the `ServerAdmin` class to provide services to other applets in the card.

This base applet class redefines (overrides) the `getShareableInterfaceObject` operation of the `Applet` class. This method is declared as `final` so subapplets will not be allowed to override it. Figure 6.3 shows the source code of the `getShareableInterfaceObject` method of the `SOSApplet` class. `lvAM` is the `SOSApplet` instance of the `AuthorizationManager` class, while `lvServerAdmin` is the `SOSApplet` instance of the `ServerAdmin` class.

Each time a client applet asks for a specific SIO, the `SOSApplet` checks first if the requested SIO is the administrative one, and in this case returns the corresponding instance. If not, i.e., the parameter is not `0`, it checks if the SIO is provided by the underlying applet (subapplet). We will show later how the subapplet registers all the SIOs when it is constructed. Then it is checked whether the client has been previously blocked, and if the corresponding SIO is available at this moment. Finally, it is checked if the client has already registered for the SIO; if so, the instance is returned, if not, the corresponding security mechanism for this SIO is returned. Note that in all cases, the rejection reason is set in the `ServerAdmin` instance. It is useful for the client applet to know why the SIO it requested was not returned.

6.1.2 Authorization Manager

The `AuthorizationManager` class is in charge of recording client authentication and administrative information. It registers all the SIOs exported by the underlying applet, as well as the security mechanism that must be used for each of them. It also holds the information about which client is registered to which SIO, as well as the list of blocked clients.

The list of registered clients is stored in an `AID array`, and when the instance is constructed, this array is built up in persistent memory or in transient memory, depending on the `CleanUpMode` parameter received by the constructor. This flexibility allows the subapplet to preserve the registration information by using the `SOSApplet.CLEAR_NEVER` clean up mode. This mode may be useful in some specific applications where the environment is quite secure. However using the `SOSApplet.CLEAR_ON_DESELECT` or the `SOSApplet.CLEAR_ON_RESET` clean up mode is highly recommended.

```
public final Shareable getShareableInterfaceObject(AID clientAID, byte parameter) {
  if (parameter == 0) {
    return lvServerAdmin;
  }
  else {
    if (lvAM.canBeProvided(parameter)) {
      if (!lvAM.isBlocked(clientAID)) {
        if (lvAM.isAvailable(parameter)) {
          if (lvAM.isRegistered(parameter, clientAID)) {
            lvServerAdmin.setRejectionReason(ServerAdmin.NOT_REJECTED);
            return lvAM.getSIO(parameter);
          }
          else {
            lvServerAdmin.setRejectionReason(ServerAdmin.CLIENT_NOT_REGISTERED_TO_SIO);
            return lvAM.getSecureSI(parameter, clientAID);
          }
        }
        else {
          lvServerAdmin.setRejectionReason(ServerAdmin.SIO_IS_UNAVAILABLE);
          return null;
        }
      }
      else {
        lvServerAdmin.setRejectionReason(ServerAdmin.CLIENT_IS_BLOCKED);
        return null;
      }
    }
    else {
      lvServerAdmin.setRejectionReason(ServerAdmin.SIO_NOT_PROVIDED);
      return null;
    }
  }
}
```

Figure 6.3: `getShareableInterfaceObject` method of `SOSApplet` class

6.1.3 Administrative Information Services

The `SOSApplet` provides a `Shareable` interface in which administrative information can be requested by client applets. These services are specified in the `ServerAdminSI` interface, and are implemented by the `ServerAdmin` class.

Figure 6.4 shows the declaration of the `ServerAdminSI` interface, which provides three operations (services). Recall that the `AuthorizationManager` uses an array to hold the information of the registered clients, and that it is built in the constructor. The `canRegisterNewClient` operation allows a client to know beforehand if there is space for registering itself to the server applet. The `unregister` operation is provided to unregister a client to a specific SIO, so as to free space in the registration array. This operation receives just the SIO because the client AID is obtained by the `JCSystem.getPreviousContextAID`, so as to prevent a client from unregistering another client. Lastly, the `getRejectionReason` is provided so as to let a client know the status of the last request for a SIO.

```
public interface ServerAdminSI extends Shareable  {

  public boolean canRegisterNewClient();

  public void unregister(byte pSIOID);

  public byte getRejectionReason();

}
```

Figure 6.4: Server Administrator `Shareable` interface

```
public interface ISIOResource {

  public void provideSIO(byte pSIOID, Shareable pSIO, SecureSI pSecureMechanism);
  public void provideSIO(byte pSIOID, Shareable pSIO, SecureSI pSecureMechanism,
                         boolean pAvailability);

  public void makeAvailable(byte pSIOID);
  public void makeUnavailable(byte pSIOID);
  public boolean isAvailable(byte pSIOID);

  public boolean isAuthorized(byte pSIOID);
  public boolean isAuthorized(byte pSIOID, AID pClient);

  public void block();
  public void block(AID pAID);
  public boolean isBlocked(AID pAID);

  public void register();
  public void register(byte pSIOID, AID pClient);
  public boolean isRegistered(byte pSIOID, AID pClient);

}
```

Figure 6.5: `ISIOResource` interface

6.1.4 Services provided to the subapplets

All the services provided by the `SOSApplet` class to the underlying applet are declared in an interface called `ISIOResource`. In the current implementation the `AuthorizationManager` class is the only class which implements this interface. In spite of that, exporting these services through a public interface is better as it allows extra flexibility for the framework implementation, i.e., future versions of the framework may use another class to implement this interface, while the applets based on the current version will remain unchanged.

The `SOSApplet` class provides the `getSIOResource` operation, which returns the `Authorization-Manager` instance of the `SOSApplet`. The functionality provided by the `ISIOResource` interface is shown in Figure 6.5.

The services provided to the underlying applet can be categorized in five groups. The `provide-SIO` group is used to register SIO instances and an associated security mechanism to the `SOSApplet`. The second group is to deal with the availability of SIOs. The subapplet can make unavailable a SIO and then make it available again. This feature is very important when an event mechanism is used. For example, when the server applet triggers an event, it can first stop some services (make SIO(s) unavailable) and after the event is finished, resume them (make SIO(s) available). The third group allows the subapplet to know if a client is authorized to use a specific service. The usage of this service is mandatory for the implementation of a SIO, as it allows the subapplet to check dynamically if the client requesting the SIO can use it at the moment of the request. The dynamic check is needed to solve one of the problems present in the Methodological approach: even if the client applet preserves a reference to the SIO, the server applet has the means to prevent this client from using the service, for example by blocking the client, unregistering it, or making the SIO unavailable. The last two groups of services make it possible to block and check if a client is blocked, and to register and check if a client is registered.

6.1.5 Open Security Mechanism

The framework provides a tagging interface `SecureSI` intended for Security Mechanisms. For all SIOs provided by the subapplet it must be indicated which security mechanism will be used. Recall that the `provideSIO` operation in the `ISIOResource` interface (see Figure 6.5) receives the SIO identifier, the SIO instance, and an instance of a class which implements a Security Mechanism. The framework provides three simple Security Mechanisms, namely `SecureSIFree` intended for free services, `SecureSIByShort` which authenticates using a `short` number, and `SecureSIBy-`

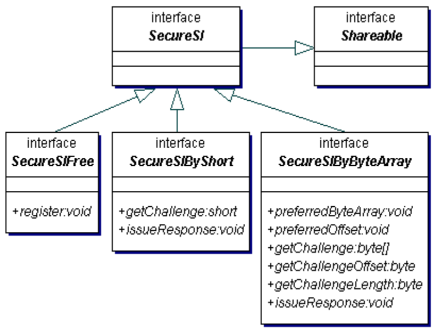


Figure 6.6: Security Mechanism Class Diagram

`ByteArray` which allows longer challenges and responses. In addition, the developer can create new authentication mechanisms just by defining an interface which extends the `SecureSI` tagging interface, and implementing it in a class.

Figure 6.6 presents the UML Class Diagram for the Security Mechanism interfaces within the framework.

6.1.6 Subapplet Implementation

Applets developed under the SOS Framework should extend the `SOSApplet` base class. The only constructor of the `SOSApplet` class is shown in Figure 6.7. It receives four `byte` arguments. The first three arguments deal with the maximum capacity of the arrays which will hold administrative information: the maximum amount of provided SIOs, the maximum amount of client registrations and the maximum amount of blocked clients. The last argument refers to the type of array to hold client registration; as it was explained before, the `AuthorizationManager` decides on this parameter if the array will be placed in persistent or transient memory, and in the latter case when the array must be cleaned up.

After invoking `super` in the subapplet's constructor, it should indicate which SIOs it is going to provide. This is simply done by invoking `getSIOResource` and using the `provideSIO` operation of this object. The subapplet should call this operation for each SIO it wants to provide.

Furthermore, the subapplet must not implement the `getShareableInterfaceObject` operation, because it would lead to a compile time error as this operation is declared as `final` in the `SOS-Applet` class.

Additionally, as any other applet, it must implement the abstract operation `process` defined in the base `javacard.framework.Applet` class. It should implement also the `install`, `select` and `deselect` operations in order to provide the applet's functionality.

Figure 6.8 presents the implementation of a Loyalty Applet which allows other applets to consult its balance. The `LoyaltyApplet` extends the `SOSApplet` class. It has two fields, one for storing the points of the customer (`lvPoints`), and the other for providing services to other applets (`lvSIO`). The constructor first invoke the constructor of the `SOSApplet` class; Figure 6.7 indicates

```
public SOSApplet(byte pMaxAvailableSIOs,
                 byte pMaxRegistrations,
                 byte pMaxBlocks,
                 byte pCleanUpMode)
```

Figure 6.7: `SOSApplet`'s Constructor signature

```
package com.loyalty;

import javacard.framework.*;
import com.sosframework.*;

public class LoyaltyApplet extends SOSApplet {

  // Identifier of the service
  static final byte GET_BALANCE_SIOID = (byte) 0xCC;

  private short lvPoints;

  private GetBalSIO lvSIO;

  // Constructor
  private LoyaltyApplet() {

    // Constructs the server applet
    super((byte) 1, (byte) 1, (byte) 10, SOSApplet.CLEAR_ON_RESET);

    // Initialize fields
    lvPoints = (short) 0;
    lvSIO = new GetBalSIO(this);

    /**
     * Register the SI it wants to export indicating:
     *   - the SIO identifier
     *   - the instance of the class implementing the service
     *   - the instance of the class which implements the security mechanism
     */
    getSIOResource().provideSIO(GET_BALANCE_SIOID, lvSIO, new LoyaltySecure(this));

    register();

  }

  // Install method
  public static void install(byte[] bArray, short bOffset, byte bLength) {
    new LoyaltyApplet();
  }

  short getBalance() {
    return lvPoints;
  }

  // ...

  public void process(APDU pAPDU) {
    // ...
  }

}
```

Figure 6.8: Subapplet Example

which parameters are expected by the `SOSApplet` class. Later, fields are initialized. The next step is to indicate the `SOSApplet` class which services will be provided to other applets. Finally, the applet is registered by invoking the `register` operation. Notice that the `process` method and other operations of the `LoyaltyApplet` class were omitted for brevity.

6.1.7 Exported Services Implementation

The developer must be particularly careful when developing the SIO classes. Each SIO should be implemented in a separate class, as it is recommended in the remarks made for the Inappropriate Casting problem. The framework relies on dynamic checks for providing services. Each service provided by a SIO should first check whether it is possible to provide the service to the client who places the request. This is simply done by using the `isAuthorized` operation of the `ISIOResour-ce` interface, providing the correct SIO identifier. This operation checks if the client is not blocked,

```
package com.loyalty;

import javacard.framework.*;

public interface GetBalSI extends Shareable {

  // Returns the amount of points of the loyalty applet
  public short getBalance();

}


class GetBalSIO implements GetBalSI {

  private LoyaltyApplet loyapp;

  GetBalSIO(LoyaltyApplet loyapp) {
    this.loyapp = loyapp;
  }


  public short getBalance() {

    // Checks if the client is authorized to query the balance
    if (loyapp.getSIOResource().isAuthorized(LoyaltyApplet.GET_BALANCE_SIOID)) {

      return loyapp.getBalance();

    }
    else {
      // Returns an invalid value
      return (short) -1;

    }

  }

}
```

Figure 6.9: SIO Example

if the SIO is available and also if the client has previously registered to the SIO.

Figure 6.9 shows the `Shareable` interface `getBalSI` defined to provide the `getBalance` service to other applets within the card. The class `getBalSIO` implements this interface. For returning the balance to the client applet it simply consult the `LoyaltyApplet` instance it received when the `getBalSIO` instance was created. Notice that before returning the balance, it is checked that the client is authorized to request this service. In order to do so, it invokes the `isAuthorized` operation defined in the SOS Framework. As it was explained before, this operation returns a `boolean` value indicating if the client (which is requesting the service) is allowed to access the service.

6.2 Tool-Kit

The Tool component is supposed to be used on a workstation, not in a Java Card (see Figure 6.1). It is made of two different tools: one of them checks statically if any class or interface present the Inappropriate Casting problem, while the other checks statically if each SIO implementation does the necessary dynamic check of a client authorization to use services.

The tools of the SOS Development Kit are work in progress. In what follows we present the general idea of their possible implementation. Future work will suggest the complete development of such tools.

Inappropriate Casting. When using the Secure Object Sharing Framework, by implementing correctly the SIO classes the Inappropriate Casting can be avoided. At each service request the SIO class must check if the client is authorized to use a specific SIO by providing the SIO identifier. Even if the client casts a reference to another, each method of the SIO class checks for the corresponding SIO identifier. So, the client must be registered for each SIO for which the class can be used. Nevertheless, this tool is provided for those developers who base their JAVA CARD applications on the Shareable Interface Object mechanism of the JAVA CARD platform, who prefer to develop their own secure object sharing mechanism, or who want to use the Methodological approach.

The tool checks each class and interface in the application package. For each class it is built a set, beginning with the class itself, and adding all super-classes (transitively) and all interfaces (including its super-interfaces) that these classes implement. If this set contains more than one interface which extends the `Shareable` interface, then this package presents the Inappropriate Casting problem. For each interface it is checked that if it extends the interface `Shareable`, then the `Shareable` interface is it direct super-interface. If this property does not hold, then the implementation is not secure in the sense that it allows Inappropriate Casting.

Dynamic Checks. The framework relies on dynamic checks when a client requests a service from a SIO. As the service request is placed directly to the SIO instance, which is implemented by the subapplet's developer, there is no possibility for the framework to check by itself if the client is authorized or not to use the service. To overcome this problem, the developer must consult the `SOSApplet` whether the client is authorized or not, as was explained in Subsection 6.1.7. This tool checks the SIOs (all classes which implements a `Shareable` interface) to see if the corresponding dynamic check is done at the beginning of the method.

It is also possible to centralize service requests in an operation `doService` defined in a general `SIO` class within the Framework. This operation should check first if the client is authorized to request the service, and in case it is authorized, the service should be requested to the underlying `SIO` class. This class would act as a proxy for services. It must be considered how services are identified and how different kind of parameters are going to be passed from the client to the proxy, and from the proxy to the underlying class (subclass). This mechanism is not present in the current framework implementation since it would lead to a larger CAP file, and to a non-intuitive service usage by clients.

The dynamic check can be done at the source code level. For each method implementing an operation defined in a `Shareable` interface, it should be checked that the corresponding `if` clause is used. Figure 6.9 shows an implementation example of a class implementing a `Shareable` interface. The `if` clause invoking the `isAuthorized` operation must be present in all methods which implement an operation defined in a `Shareable` interface.

7 Conclusions

We presented an overview of the drawbacks of the JAVA CARD Shareable Interface Object mechanism, namely Impossibility of Passing Object as Parameters, Inappropriate Casting, Applet Impersonation and Limited Clients. We indicated the scenarios where Object as Parameters and Inappropriate Casting take place, and gave general suggestions to solve these problems. Applet Impersonation and Limited Client problems have been previously addressed by the Delegate Object approach and the Methodological approach. We put forward these approaches in Subsection 4.1 and 4.2 respectively.

We introduced the Secure Object Sharing Framework for the JAVA CARD platform. The Framework provides the `SOSApplet` class, which extends the base `Applet` class. This class enriches the Shareable Interface Object mechanism by solving the problems and providing extra functionalities to the subapplets. The Framework is based on the Methodological approach, and improves it by:

- Allowing the server applet to block clients
- Allowing the server applet to change the availability of SIOs,
- Automatically blocking clients that do an incomplete authentication procedure
- Providing an extensible security mechanism by the `SecureSI` interface.

It solves the problems that are present in the Methodological approach by using dynamic checks at every service request. These checks just require consulting the `AuthorizationManager` if the client is authorized for a specific service, not going through the authentication procedure again as in the Delegate Object approach.

We also introduced the problem of the challenge/response mechanism for commercial JAVA CARD applications. An approach to overcome this flaws was also presented. It constitutes an interesting alternative, still vulnerable, but perfectible. This approach allows the server applet's developer to sell services to untrusted clients. Additionally, the authentication mechanism is simpler for the client, as it is already implemented in the `Certificate` class.

The tool for detecting the Inappropriate Casting problem, described in Subsection 6.2, can be used even when using the plain object sharing mechanism of the JAVA CARD platform. This tool ensures the developer that no client applet will be accessing services by illegitimate casting of Shareable Interface Objects. The tool for detect the absence of dynamic checks is an important companion component of the Secure Object Sharing Framework, which strongly relies on checking at runtime client authorization for service request.

The current implementation of the SOS Framework, containing all classes and interfaces shown in Figure 6.2, was generated into a CAP file whose size is 4 KB. Hence, it fits very well in current JAVA CARD hardware. Recall that the SOS Framework has to be loaded only once in a card, as it is implemented in a separate package rather than in all server applets. However, extra memory will be needed when each subapplet is instantiated.

Future work. Future work includes the development of both tools proposed in the Secure Object Sharing Development Kit. The tool for Inappropriate Casting can use the *introspection* capability of JAVA, and seems not to be a difficult task. On the other hand, developing the tool for checking Dynamic Checks might result in a very tedious job.

Another interesting issue is to implement a Case Study based on the Secure Object Sharing Development Kit. The Electronic Purse JAVA CARD implementation proposed in [1] can take advantage of the capabilities of the Kit for solving the problem it faces.

The challenge/response solution is still vulnerable. The possibility of inspecting a CLASS or CAP file makes it possible to extract the shared secret used by the server and the certificate. Future work is to analyze the alternative solution of providing two packages. One of them would contain interfaces and would be sold to the client. The other would be provided and managed by the server's developer. The combination of both packages represents the solution proposed in this paper.

8 Acknowledgements

The help and comments of Gustavo Betarte and the anonymous referees are gratefully acknowledged.

References

[1] P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Electronic purse applet certification: extended abstract. In S. Schneider and P. Ryan, editors, *Proceedings of the Workshop on Secure Architectures and Information Flow*, volume 32 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, 2000.

[2] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's guide.* The Java Series. Addison-Wesley, Reading, MA, USA, June 2000.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, USA, 1995.

[4] P. Girard. Which security policy for multiapplication smart cards? In *Workshop on Smartcard Technology (Smartcard '99), USENIX*, Chicago, USA, May 1999.

[5] H. Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, Reading, MA, USA, January 2000.

[6] M. Montgomery and K. Krishna. Secure object sharing in Java Card. In *Workshop on Smartcard Technology (Smartcard '99), USENIX*, Chicago, USA, May 1999.

[7] Object Management Group. *OMG Unified Modeling Language Specification*, March 2000. `http://www.omg.org/uml`.

[8] Daniel Perovich, Leonardo Rodríguez, and Martín Varela. A simple methodology for secure object sharing. In *Proceedings of the Gemplus Developer Conference 2001 (GDC 2001)*, CNIT Conference Center, Paris, France, June 2001. Gemplus.

[9] Rational Software Corporation. *UML Notation Guide*, September 1997. `http://www.rational.com/uml`.

[10] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1.1 API Specification*, 2000. `http://www.javasoft.com/products/javacard`.

[11] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1.1 Runtime Environment Specification*, 2000. `http://www.javasoft.com/products/javacard`.