

**Hauptseminar (WS 2001/2002):**

## **Applet Firewall und Freigabe der Objekte**

Nachweis von Sicherheitseigenschaften  
für JavaCard durch approximative Programmauswertung

**Veranstalter: Prof. T. Nipkow, Dr. M. Strecker**

**Autor: Jin Zhou ([zhou@in.tum.de](mailto:zhou@in.tum.de))**

**10. Dezember 2001**

### **Inhaltverzeichnis:**

1 Applet Firewall.....	2
1.1 Kontexte.....	2
1.2 Besitz der Objekte.....	3
1.3 Der Zugriff zum Objekt .....	3
1.4 Statische Felder und Methoden .....	3
2 Objektfreigabe über den Kontexten .....	4
2.1 Kontextumwandlung.....	4
2.2 JCRE Privilegien.....	5
2.3 JCRE Entry Point Objekte .....	5
2.4 Globale Arrays .....	6
2.5 Objekt Shareable Interface Mechanismus.....	7
2.5.1 Shareable Interface.....	7
2.5.2 Shareable Interface Objekte .....	7
2.5.3 Gedanke hinter dem Shareable Interface Mechanismus .....	8
2.5.4 Ein Beispiel von Objektfreigabe zwischen Applets .....	9
2.5.5 Ein Shareable Interface erzeugen.....	9
2.5.6 Aufforderung an einen Shareable Objekt.....	10
2.5.7 Anwendung mit einem SIO.....	11
2.5.8 Kontextumwandlung während Objektfreigabe .....	12
2.5.9 Typen von Parametern und Rückwerten in SI-Methoden .....	13
2.5.10 Beglaubigung eines Client-Applets.....	14
2.5.11 Die getPreviousContextAID Methode.....	16
2.5.12 Zusammenfassung.....	17
Literatur.....	18

## **Applet Firewall und Freigabe der Objekte**

Die Java Card Plattform ist eine Umgebung der Multiapplikationen. Mehrere Applets für verschiedene Anwendungen können auf einer einzigen Karte existieren und auf dieser Karte können noch zusätzliche Applets hinzugefügt werden, nachdem sie hergestellt worden ist. Aber häufig behält ein Applet sehr sensitive Informationen wie zum Beispiel: das elektronische Geld, das Fingerabdruck, das private Geheimnispassword und so weiter. Offensichtlich muss die Freigabe solcher Daten zwischen Applets sehr sorgfältig beschränkt sein.

Auf der Java Card Plattform wird solche Isolierung zwischen Applets durch „Applet Firewall“ Komponent durchgeführt. Das Applet Firewall beschänkt den Datenzugriff eines Applets nur auf seinem eigenen Bereich.

Um eine Zusammenarbeit zwischen Applets auf derselben Karte zu verwirklichen– zum Beispiel: Telefonkartefunktion –hat die Java Card Technologie ein klares und sicheres Mechanismus zur Objekt-Freigabe angeboten.

In diesem Artikel wird das Verhalten der Objekte, die Exceptions und Applets hinter dem Firewall erklärt und auch diskutiert, wie die Applets die Freigabe der Daten durch die Java Card APIs sichern können.

### **1 Applet Firewall**

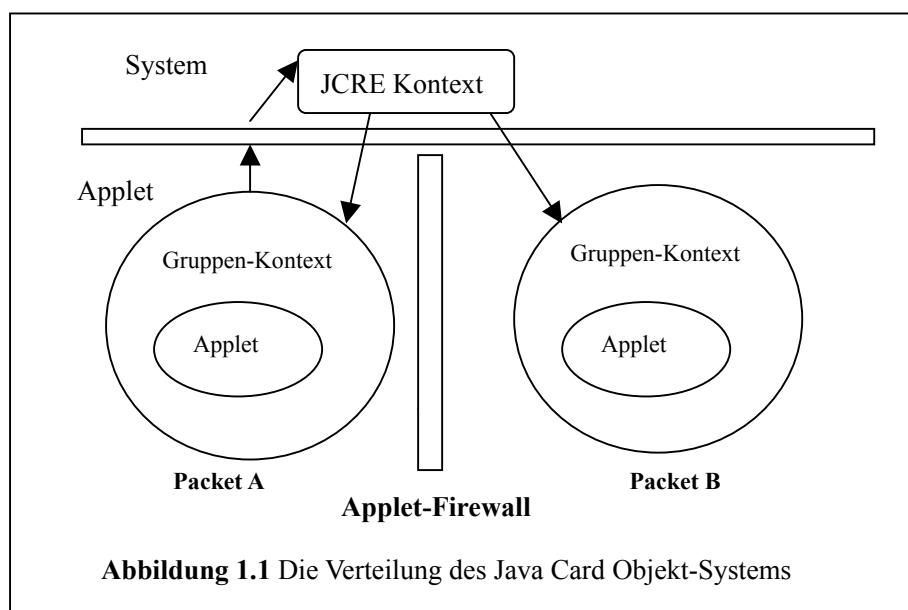
Mit der Applet-Isolierung kann das Applet-Firewall die Sicherheit schützen: Der Entwickler hat z.B. falsch getan und entworfen, dass das Applet den empfindlichen Daten erlaubt, zu einem anderen Applet gelect zu werden. Also muß hier ein Schutz gegen Hacking angeboten wird. Ein Applet kann ein Objekt bekommen von einem öffentlich zugänglichen Ort, aber wenn das Objekt einem anderen Applet in einem anderen package gehört, verhindert das Firewall den Zugriff zu diesem Objekt.

#### **1.1 Kontexte**

Das Applet-Firewall trennt das Java Card Objekt System in verschiedene geschützte Objektplätze, die wir Kontexte nennen. Und das Firewall ist die Grenze zwischen einem Kontext und den anderen. Wenn ein Applet initialisiert ist, teilt ihm das JCRE einen Kontext zu. Dieser Kontext ist wesentlich ein Gruppenkontext. Alle Applets in demselben package sind in demselben Kontext. Es gibt kein Firewall zwischen den Applets im gleichen Gruppenkontext. Der Objektzugriff zwischen den Applets in demselben package ist erlaubt. Aber der Zugriff zum Objekt in den verschidenen Gruppenkontexten ist von dem Firewall verweigert.

Außerdem befindet sich das JCRE(Java Card Runtime Enviroment) in seinem eigenen Kontext. Der JCRE-Kontext ist ein System-Kontext und hat ein Sondervorrecht: Der Zugriff von dem JCRE-Kontext zu allen anderen Applets-Kontexten ist erlaubt, aber andererseits wird der Zugriff von einem Applet-Kontext zum JCRE-Kontext vom Firewall verboten.

Die Verteilung des Java Card Objekt-Systems ist wie in Abbildung 1.1 gezeichnet:



## 1.2 Besitz der Objekte

Zu jeder Zeit gibt es nur einen aktiven Kontext innerhalb der Virtual Maschine: entweder den JCRE Kontext oder einen Appletsgruppen-Kontext. Wenn ein Objekt neu gebaut wird, wird ihm ein Kontext zugeteilt – der momentan aktive Kontext. Und er kann von allen Applet, die im gleichen Kontext sind, erreicht werden. Also wir sagen, dass dieses Objekt von dem aktiven Applet besessen wird, wenn es erstellt wird. Wenn der JCRE-Kontext der aktive Kontext ist, ist das Objekt von der JCRE besessen.

Statische Arrays der primitiven Art im Applet können initialisiert werden, wenn sie deklariert werden. Solche statische Arrays werden durch den Konverter erstellt und initialisiert. Weil sie statisch hergestellt werden, bevor noch keine Applets auf der Karte initialisiert werden, wird der Besitz dieser Arrays irgendeinem Applet in ihrem Definierenpackage zugewiesen werden. Jedes Applet in ihrem Package kann auf diese Arrays zugreifen. Mit anderem Wort, der Kontext von diesen Reihen ist nichts anderes als der Gruppenkontext von dem Packet.

## 1.3 Der Zugriff zum Objekt

Wenn ein Objekt zugegriffen wird, soll eine Zugriffskontrolle durchgeführt werden. Zum Beispiel: Eine private Methode darf nicht ausgeführt werden außerhalb des Objektes. Außerdem wird der Kontext dieses Objektes zum aktiven Kontext verglichen. Wenn die beiden Kontexte nicht gleich sind, wird der Zugriff verweigert und wird ein `SecurityException` geworfen.

## 1.4 Statische Felder und Methoden

Keine Kontextprüfung zur Laufzeit wird durchgeführt, wenn ein statisches Feld zugegriffen wird oder eine statische Methode aufgerufen wird. Mit anderen Worte, die statischen Felder und Methoden sind zugreifbar für alle Kontexte. Zum Beispiel: jedes Applet kan die statische Methode

in der Klasse ISOException aufrufen:

```
If (apdu_buffer[ISO07816.OFFSET_CLA] != EXPECTED_VALUE)
    ISOException.throwIt(ISO07816.SW_CLA_NOT_SUPPORTED);
```

Freilich müssen alle Regeln für normale Java-Programme hier auch gelten. Zum Beispiel: Statische Felder und Methoden, die mit *private* erklärt wird, sind auch sichtbar nur für ihre definierte Klasse.

Wenn eine statische Methode aufgerufen wird, wird sie nur im Kontext des aufrufenden Programms ausgeführt. Und die Objekte, die innerhalb einer statischen Methode erstellt werden, werden zugewiesen mit dem Kontext des aufrufenden Programms (der momentan aktive Kontext).

Statische Felder sind zugänglich für alle Kontexte. Aber jedoch sind die Objekte (einschließlich Arrays) auf statischen Feldern wie normale Objekte. Solche Objekte gehören zu dem Applet (oder JCRE) , das sie herstellt hat, und müssen die Standardfirewallzugreifsregeln passen.

## 2 Objektfreigabe über den Kontexten

Der Applet-Firewall beschränkt die Tätigkeit eines Applets auf seinem gekennzeichneten Kontext. Der Applet kann nicht über seinen Kontext auf ein Objekt von JCRE oder von einem anderen Applet mit einem unterschiedlichen Kontext zugreifen. Aber in der wirklichen Anwendung muß das Applet manchmal mit dem anderen kooperieren. Die Java Card Technologie hat für solche Situation ein sicheres Freigabemechanismus angeboten. Und zwar folgende:

- JCRE Privilegien
- JCRE Entry Point Objekte
- Globale Arrays
- Shareable Interfaces

Dieses Freigabemechanismus erlaubt im Wesentlichen, dass ein Kontext auf einen zu einem anderen Kontext gehörenden Objekt unter spezifischen Bedingungen zugreifen kann.

### 2.1 Kontextumwandlung

Es gibt nur einen aktiven Kontext zu jeder Zeit innerhalb der Ausführung auf der Java Virtual Maschine. Alle Objektzugriffe werden von der VM überprüft, ob sie erlaubt sind. Normalerweise wird ein Zugriff verweigert, wenn der Kontext des zuzugreifenden Objekts nicht gleich als der momentan aktive Kontext ist. Aber wenn ein Freigabemechanismus durchgeführt ist, erlaubt die Java Card VM diesen Zugriff durch eine Kontextumwandlung.

Komponenten in einem Objekt besteht aus Methoden und Feldern. Für den Zugriff zu einem Feld von einem Objekt mit dem anderen Kontext wird die Kontextumwandlung nicht durchgeführt.

Nur das JCRE kann ein Feld von einem Objekt in anderem Kontext zugreifen.

Bei der Anforderung einer Methode an der Kontextumwandlung wird zuerst der aktuelle Kontext gespeichert und der neue Kontext wird der aktuelle aktive Kontext. Die aufzurufende Methode ist schon im neuen Kontext und hat das Zugriffsrecht zum aktuellen Kontext. Wenn die Methode durch ein normales *return* oder eine Exception endet, wird der ursprüngliche Kontext als der aktuell aktive Kontext wiederhergestellt. Zum Beispiel: Wenn ein Applet eine Methode von einem JCRE-entry-point-Objekt aufrufen will, wird dann eine Kontextumwandlung von dem Kontext des Applets zum Kontext von JCRE durchgeführt.

Da der Aufruf der Methoden verschachtelt sein kann, kann die Kontextumwandlung auf der Java Card VM auch verschachtelt sein. Wenn die VM anfängt zu laufen nach dem Card Reset, ist der JCRE-Kontext für immer der aktuelle aktive Kontext.

## 2.2 JCRE Privilegien

Auf der Java Card Plattform dient das JCRE als das Leitprogramm. Und der JCRE-Kontext ist auch der „System“-Kontext, deshalb hat der JCRE-Kontext spezielle Privilegien. Er kann alle Methode von allen Objekten aufrufen und auf alle Felder von allen Objekten auf der Karte zugreifen. Solche System-Privilegien erlauben das JCRE, die System-Betriebsmittel zu kontrollieren und die Applets zu steuern. Zum Beispiel: Wenn das JCRE einen APDU-Befehl erhält, kann es die Methoden des ausgewählten Applets aufrufen wie *select*, *deselect*, oder *process*.

Wenn das JCRE eine Methode vom Applet aufruft, wird der Kontext vom JCRE zum Kontext des Applets umgewandelt. Das Applet nimmt dann die Steuerung und verliert die JCRE Privilegien. Alle Objekte, die nach der Kontextumwandlung erstellt werden gehören zu dem Applet und haben den gleichen Kontext wie das Applet. Beim Return der Methode vom Applet wird der JCRE-Kontext wiederhergestellt.

## 2.3 JCRE Entry Point Objekte

Das JCRE kann auf alle Appletskontexte zugreifen, aber die Applets dürfen nicht auf den Kontext vom JCRE zu zugreifen. Ein sicheres System muß auch einen Weg für die normalen Benutzer (die, die keine Privilegien haben) anbieten, die Systemservices vom privilegierten Systemprogramm zu verlangen. Auf der Java Card Plattform ist solche Anforderung durch *JCRE Entry Point Objekte* erreicht.

Die JCRE Entry Point Objekte sind normale Objekte und gehören zum JCRE-Kontext, nur sie sind markiert als Enthalten von Entry Point Methoden. Normalerweise schützt das Firewall solche Objekte vorm Zugriff von allen Applets. Die Entry Point Kenzeichnung erlaubt die public Methoden von solchen Objekten, von allen Kontexten zuzugreifen. Wenn der Zugriff auftritt, ist eine Contextumwandlung zum JCRE-Kontext durchgeführt. So sind solche Methoden das Gateway, wodurch Applets JCRE-Services anfordern können. Aber man muß achten, dass nur die public Methoden von JCRE entry point Objekten zugriffbar durch das Firewall dürfen. Alle anderen methoden bleiben noch vom Firewall geschützt zu sein.

Das APDU Objekt ist vielleicht der am häufigsten genutzte JCRE entry point Objekt.

Es gibt zwei Kategorien vom JCRE Entry Point Objekt:

- *Temporäre JCRE Entry Point Objekte* — Wie alle JCRE Entry Point Objekte können Methoden von temporären JCRE Entry Point Objekten von allen Kontexten zugegriffen werden. Aber die Referenzen zu diesen Objekten können nicht in Klassevariablen, Instanzvariablen oder Reihensfeld von einem Applet gespeichert werden. Das JCRE ermittelt und schränkt die Versuche zum Speichern der Referenzen in diese Objekte. Es dient zum Teil der Funktion des Firewalls, damit die nicht autorisierte Wiederverwendung verhindert werden kann. Der APDU Objekt und alle JCRE Exception-Objekte sind schon Beispiele von temporären JCRE Entry Point Objekten.
- *Ständige JCRE Entry Point Objekte* — Wie alle JCRE entry point Objekte können Methoden von temporären JCRE entry point Objekten von allen Kontexten zugegriffen werden. Außerdem können die Referenzen zu diesen Objekten gespeichert und frei wiederverwendet werden. Das JCRE AID-Instance ist schon ein Beispiel vom ständigen JCRE Entry Point Objekt. Das JCRE erzeugt ein AID für jedes Applet, wenn es erzeugt wird.

Nur das JCRE selbst kann Entry Point Objekte bestimmen und auch bestimmen, ob sie temporär oder ständig sind. Der JVRE Implementor ist dafür verantwortlich, welche JCRE Entry Point Objekte bestimmt werden und ob sie temporär oder ständig sein sollen.

## 2.4 Globale Arrays

Die JCRE Entry Point Objekte erlauben den Applets das Zugriff zu speziellen JCRE Services, indem sie ihre jeweiligen Entry Point Methoden aufrufen. Die in den JCRE Entry Point Objekten eingekapselten Daten sind nicht direkt von den Applets zugreifbar. Aber auf der Java Card Plattform sind die globalen Arrays von allen Applets und vom JCRE Kontext zugreifbar.

Um globale Dateien flexibel zugreifen zu können, erlaubt das Firewall das JCRE, primitive Arrays als global zu bestimmen. Die globalen Arrays bieten im Wesentlichen ein freigegebenes Memorybuffer, dessen Daten von anderen Applets und vom JCRE zugegriffen werden können.

Globale Arrays sind ein spezieller Fall von JCRE Entry Points Objekten. Das Applet-Firewall erlaubt die Publicfelder (Inhalte der Arrays und Länge der Arrays) von solchen Arrays zugreifbar von allen Kontexten. Die Publicmethoden von globalen Arrays sind gleich behandelt wie die Methoden von den JCRE Entry Point Objekten. Die einzige Methode in der Arraysklasse ist die `equals` Methode, die von der Root-Klasse `Objekt` vererbt wird. Wie Methoden von einem JCRE Entry Point Objekt aufgerufen werden, wird bei dem Aufruf auch eine Kontextumwandlung von dem aktuellen Kontext zum JCRE-Kontext durchgeführt.

Nur primitive Arrays können als globale ernannt werden, und nur das JCRE selber kann globale Arrays bestimmen. Alle globalen Arrays müssen auch temporäre JCRE Entry Point Objekte sein. Deshalb dürfen die Referenzen zu diesen Arrays nicht gespeichert werden in Klassevariablen, in

instance Variablen, oder in Arraykomponenten.

Die einzigen globalen Arrays, die in den Java Card APIs benötigt sind, sind das APDU-Buffer und die Byte-Arrays-Parameter in der `install` Methode der Applets. Ein typisches Beispiel ist folgendes: Eine JCRE-Implementierung führt den APDU-Buffer als der Byte-Arrays-Parameter zur `install` Methode. Da die globalen Arrays überall sichtbar und zugreifbar sind, kann das JCRE den APDU-Buffer löschen, sobald ein Applet ausgewählt ist, oder bevor das JCRE einen neuen APDU-Befehl übernimmt, damit keine sensitiven Daten von den Applets durch den globalen APDU-Buffer möglicherweise ausgelaufen werden.

## 2.5 Objekt Shareable Interface Mechanismus

Noch mal zusammen, das Freigabemechanismus zwischen dem JCRE und den Applets sind:

- Wegen seiner Privilegien kann das JCRE auf alle Objekte zugreifen.
- Ein Applet kann auf Systemservices zugreifen durch die JCRE Entry Point Objekte.
- Das JCRE und die Applets können primitive Daten zusammenbenutzen, indem Sie die Daten als globale Reihen ernennen.

Die Java Card Technologie erlaubt die Objektfreigabe zwischen Applets noch durch das Shareable Interface Mechanismus.

### 2.5.1 Shareable Interface

Ein Shareable Interface ist einfach nur ein Interface, das direkt oder indirekt von dem Interface `javacard.framework.Shareable` vererbt wird.

```
public interface Shareable {}
```

Dieses Interface definiert weder Methoden noch Felder. Sein alleiniger Zweck ist von anderen Interfaces zu vererben. Und die anderen Interfaces können auch eigene Methoden oder Felder weiterdefinieren.

Ein shareable Interface definiert Methoden, die von anderen Applets zugreifbar sind. Eine Klasse darf beliebig viel Shareable Interfaces implementieren und kann auch von andere Klassen vererbt sein, die shareable Interfaces implementieren.

### 2.5.2 Shareable Interface Objekte

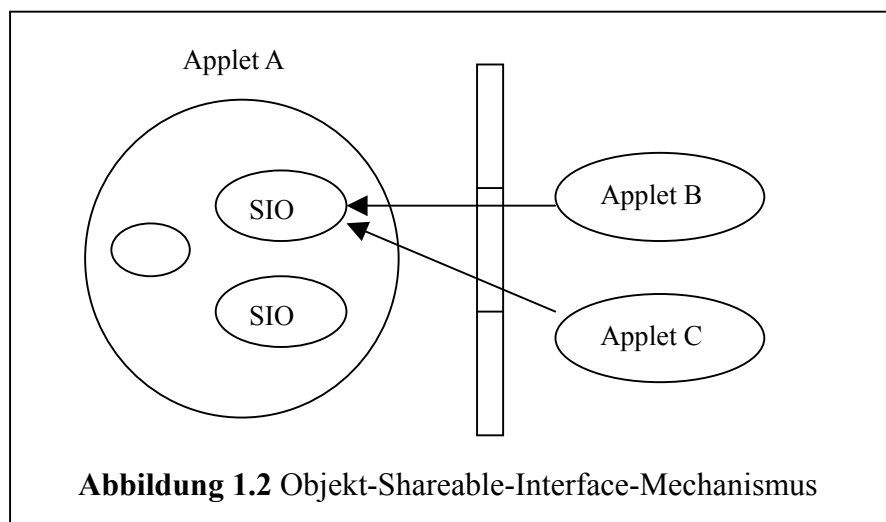
Ein Objekt von einer Klasse, die ein Shareable Interface implementiert, heißt ein Shareable Interface Objekt (SIO). Zum eigenen Kontext ist ein SIO wie ein normaler Objekt, und alle seine Methoden und Felder können zugegriffen werden. Zu anderen Kontexten sind nur die Methoden zugreifbar, die in dem Shareable Interface definiert sind. Alle anderen Methoden und Felder vom SIO sind versteckt hinter dem Firewall.

### 2.5.3 Gedanke hinter dem Shareable Interface Mechanismus

Die Applets speichern Daten in Objekten. Datenfreigabe zwischen den Applets bedeutet, ein Applet gibt ein Objekt von ihm zum anderen Applets frei, und die freizugebenen Daten sind im Objekt eingekapselt.

Auf der Objektorientierten Welt ist das Verhalten eines Objektes durch seine Methoden ausgedrückt. Messagesendung und Methodeanforderung ermöglichen die Kommunikationen zwischen Objekten. Das shareable Interface Mechanismus erlaubt die Messagesendung zwischen Applets ohne vom Firewall kontrolliert zu werden. Ein Applet erzeugt ein SIO und implimentiert Methoden, die im shareable Interface definiert werden. Die Methoden stellen das public Interface von ihrem Applet dar, wodurch die anderen Applets Messages senden und infolgedessen die bereitgestellten Services zugreifen können von diesem Applet.

Beispiel: Wie es in der Abbildung 1.2 gezeichnet wird, das Applet A ist ein Server ( bietet SIOs an), und das Applet B und C ( benutzen die SIOs vom A) sind Clients. Ein Applet kann ein Server und gleichzeitig auch ein Client sein.



In der Java-Programmiersprache definiert ein Interface eine Bezugsart, die einige Methodensignaturen und Konstanten enthält. Für ein Client-Applet ist nur das shareable Interface sichtbar. Das heißt, nur methoden, die im shareable Interface definiert werden, sind überreicht zum Client-Applet, andere Felder und Methoden sind nicht freigegeben. Und damit kann das Server-Applet Ihre Freigabe schon kontrollieren.

Wenn das Server-Applet mit einem anderen Applet zusammenwirkt, muß es auch einen anderen Hut tragen. Das bedeutet, dass das Server-Applet seine Services mit dem Respekt vom Client-Applet anpassen kann, ohne das Tür immer weit zu öffnen. Ein Server-Applet kann es erledigen durch Definition von mehreren Interfaces, jedes Interface deklariert Methoden, die einer Gruppe von Client-Applets passen. Wenn Methoden in Interfaces deutlich sind, kann ein Server-Applet Klassen erzeugen, die jeweils ein Interface implementiert. Aber oft sind die Services eine Überschneidung, ein Server-Applet kann eine Klasse definieren, die mehrfache Interfaces implementiert. Deshalb kann ein SIO mehrfache Rollen spielen.



```
}
```

Und dann erzeugt das Server-Applet eine Serviceversorgerklasse (eine Serviceversorgerklasse darf auch diese Applet-Klasse selbst sein), die das shareable Interface implementiert. Das Server-Applet kann jetzt einen oder mehrere Objekte von dieser Serviceversorgerklasse erzeugen und diese Objekte (SIOs) den anderen Applets in unterschiedlichen Kontexten freigeben.

```
package com.fasttravel.luftmeilen;
import javacard.framework.*;
public class LuftMeilenApp extends Applet implements LuftMeilenInterface{
    private short meilen;
    public void grantMeilen(short amount){
        meilen = (short)(meilen + amount);
    }
}
```

Bevor ein Client ein SIO auffordern kann, muß es zuerst den Server identifizieren. Auf der Java Card Plattform ist solches Applet eindeutig mit einem AID indentifiziert.

In anderem Kapitel wird schon erklärt, wenn ein Applet erzeugt wird, ist es auch gleich registriert durch Aufruf von einer oder zwei **register** Methoden vom JCRE. Die Methode ohne Parameter registriert das Applet mit dem in der CAP-File definierten Default-AID vom JCRE. Eine andere **register** Methode registriert das Applet mit einem spezifischen AID als das Default. Das JCRE kapselt die AID-bytes in einem AID-Objekt ein und verbindet dieses AID-Objekt mit dem Applet. Das AID-Objekt ist also verwendet, wenn ein Client-Applet den Server spezifiziert.

### 2.5.6 Aufforderung an einen Shareable Objekt

Bevor das Client-Applet einen SIO vom Server-Applet auffordert, muß es zuerst eine Verbindung zum AID-Objekt vom Server-Applet errichten. Um es tu tun, ruft das Client-Applet eine **lookupAID** Methode in der Klasse **JCSystem**:

```
public static AID lookupAID (byte[] buffer, short offset, byte length)
```

Das Client-Applet muß erst die AID-Bytes vom Server-Applet und schreibt die in den Parameterbuffer. Die **lookupAID** Methode gibt einen AID-Objekt vom Server-Applet zurück oder ein null, wenn das Server-Applet nicht auf der Karte installiert wird. Da der AID-Objekt auch ein ständiger JCRE entry point Objekt ist, braucht das Client-Applet den AID-Objekt nur einmal aufzufordern und kann ihn in einer ständigen Stelle für spätere Anwendung speichern.

Und dann ruft das Client-Applet die Methode **JCSystem.getAppletShareableInterfaceObject** auf, um den Server zu identifizieren.

```
Public static Shareable getAppletShareableInterfaceObject(AID server_aid, byte
parameter)
```

Das zweite Parameter ist für das Server-Applet geeignet, wenn das Server-Applet mehr als ein SIO-Objekt zur Verfügung stellen kann.

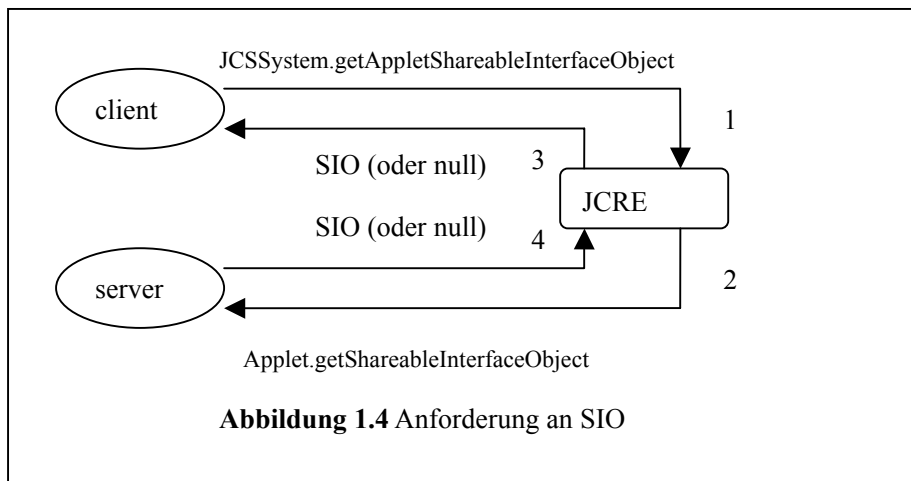
In der `getAppletShareableObject` Methode vergleicht das JCRE das `server_aid` und die AIDs von den vom JCRE registrierten Applets. Wenn das Server-Applet nicht gefunden ist, gibt das JCRE null zurück. Andernfalls ruft das JCRE die `getAppletShareableObject` Methode von dem Server-Applet.

```
public static Shareable getAppletShareableObject(AID client_aid, byte parameter)
```

Die Methode `getShareableObject` ist definiert in der Klasse `java.framework.Applet`. Die vorgegebene Implementierung gibt ein null zurück. Eine Appletklasse muß diese Methode überschreiben, wenn sie die SIOs freigeben will. Folgende Codes werden zeigen, wie das Luft-Meilen-Applet die `getShareableObject` Methode implementiert.

```
public class LuftMeilenApp extends Applet implements LuftMeilenInterface{
    short meilen;
    public Shareable getShareableObject(AID client_aid, byte parameter){
        return this; // das SIO zurückgeben
    }
    public void grantMeilen(short amount){
        meilen = (short)(meilen + amount);
    }
}
```

Wenn das Server-Applet das SIO zurückgegeben hat, schickt es das JCRE zum Client-Applet. Der Prozeß von der Aufforderung an ein SIO ist in Abbildung 1.4 gezeichnet:



### 2.5.7 Anwendung mit einem SIO

Um es zu ermöglichen, dass das zurückgegebene SIO vom Server allen SIO-Typen passen kann, sind alle zurückgegebenen SIOs der beiden Methoden vom Typ `Shareable` – der Basistyp von allen SIOs. Ein Client-Applet muß das zurückgegebene SIO erst zu einem passenden Interface wechseln, das das Client-Applet braucht. Wie z.B. :

```
LuftMeilenInterface sio = (LuftMeilenInterface)JCSystem.
getAppletShareableObject(AID server_aid, byte parameter);
```

Nachdem das Client-Applet das SIO bekommen hat, ruft es die Shareable-Interface-Methode auf zum Zugriff zu den Services vom Server. Aber auf jeden Fall sind nur die im shareable Interface definierten Methoden sichtbar vom Client-Applet.

Folgendes ist ein Beispiel, wie das Brieftaschen-Applet das Meilenaddirungsservice vom Luftmeilen-Applet auffordert.

```
package com.smartbank.brieftasche;
import javacard.framework.*;
import com.fasttravel.luftmeilen.LuftMeilenInterface;
public class BrieftascheApp extends Applet{
    private short balance;
    private byte[] luft_meilen_aid_bytes = SERVER_AID_BYTES;
    //Geld ausgeben
    private void debit(short amount){
        if(balance < amount) ISOException.throwIt(SW_EXCEED_BALANCE);
        balance = (short)(balance - amount);
        // das Meilenaddirungsservice auffordern
        requestMeilen(amount);
    }
    private void requestmeilen(short amount){
        //einen Server AID-Objekt bekommen
        AID luft_meilen_aid = JCSystem.lookupAID(luft_meilen_aid_bytes,
(short)0, (byte)luft_meilen_aid_bytes.length);
        if(luft_meilen_aid == null)
ISOException.throwIt(SW_LUFT_MEILEN_APP_NOT_EXUST);
        //das SIO vom Server auffordern
        LuftMeilenInterface sio = (LuftMeilenInterface)
(JCSystem.getAppletShareableInterfaceObjekt(luft_meilen_aid, SECRET));
        if(sio == null) ISOException.throwIt(SW_FAILED_TO_OBTAIN_SIO);
        // das Meilenaddirungsservice vom Server auffordern
        sio.grantMeilen(amount);
    }
}
```

Wenn ein Fehler aufgetreten ist, kann das Applet ein ISOException auswerfen, indem es die statische Methode throwIt aufruft. Diese throwIt Methode wirft einen JCRE ISOException-Objekt, der JCRE entry point Objekt gehört und zugreifbar von allen Applets sein kann.

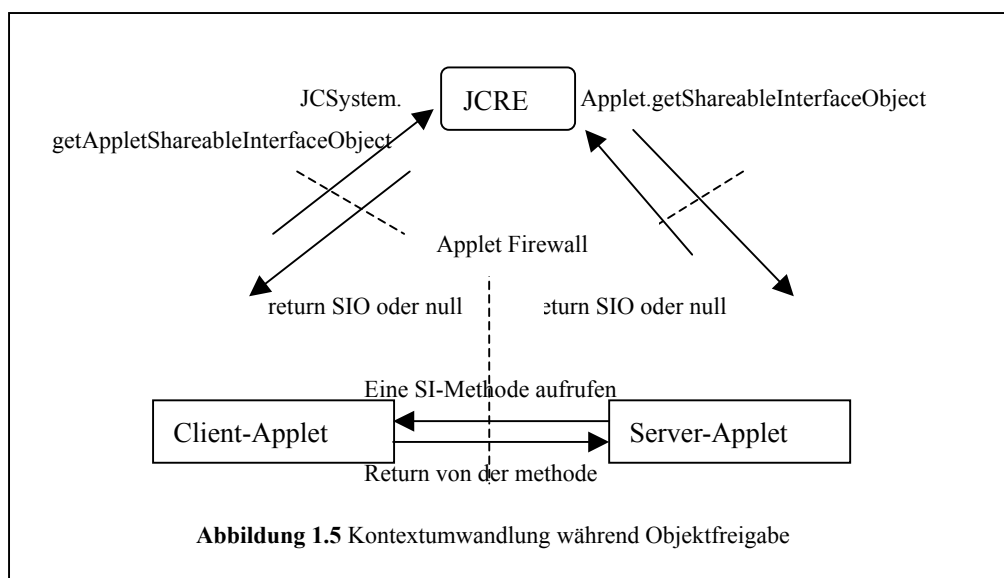
### 2.5.8 Kontextumwandlung während Objektfreigabe

Das JCRE, das Client-Applet und das Server-Applet bleiben in verschiedenen Kontexten. Deshalb muß eine Kontextumwandlung bei der Objektfreigabe durchgeführt werden. Das Client-Applet ruft die JCSystem.getAppletShareableInterfaceObjekt Methode zur Aufforderung von einem

SIO auf. Ein internes Mechanismus in der Methode wird dann ausgeführt, um den Kontext des Clients zum Kontext des JCRE umzuwandeln. Und dann ruft das JCRE die `getAppletShareableInterfaceObject` Methode vom Server-Applet auf. Solche Aufforderung führt noch zu einer Kontextumwandlung, so daß der Kontext vom Server-Applet das aktuelle wird. Am Ende der Aufforderung wird der Kontext vom Client-Applet wieder zurückgesetzt.

Und dann kann das Client-Applet die Services vom Server auffordern, indem es die shareable Interface Methoden vom Server mit dem bekommenen SIO aufruft. Während der Aufforderung führt die Java Card Visual Maschine eine Kontextumwandlung aus. Der Kontext des Server-Applets wird der aktuell aktive Kontext sein.

Die Kontextumwandlung während der Objektfreigabe ist veranschaulicht wie in der Abbildung 1.5:



### 2.5.9 Typen von Parametern und Rückwerten in SI-Methoden

In der Java-Programmiersprache können die Parameter von Methoden und die Rückwerte von allen Typen sein, aber auf der Java Card Plattform sind die beschränkt worden. Zum Beispiel: Wenn das Brieftasche-Applet einen eigenen Objekt als Parameter in der `grantMeilen` Methode gibt, wird die Firewall den Zugriff zu diesem Objekt verhindern. Ebenso, wenn das Luft-Meilen-Applet einen eigenen Objekt zurückgibt, wird die Firewall das Brieftasche-Applet auch am Zugriff zu diesem Objekt verhindern. Um das Problem zu vermeiden, können wir die Werte von folgende Typen verwenden:

- *Primitive Werte* – Die primitiven Typen auf der Java Card Plattform sind `boolean`, `byte`, `short` und (beliebig) `int`.
- *Statische Felder* – Public statische Felder sind zugreifbar von allen Kontexten.
- *JCRE entry point Objekte* – Public Methoden von diesen Objekten sind zugreifbar von allen Kontexten.
- *Globale Reihen* – Sie können von allen Kontexten zugegriffen werden.
- *SIOs* – Shareable Interface Methode von diesen Objekten können von allen Kontexten zugegriffen werden.

## 2.5.10 Beglaubigung eines Client-Applets

Damit kein nichtauthentifiziertes Client die geschützten Daten zugreifen kann, soll das Server-Applet das Client-Applet authentifizieren, bevor es einen SIO bekommen und eine Service-methode durch SIO aufruft.

Um zu bestimmen, wer einen SIO bekommen kann, muß der Server das AID vom Client-Applet überprüfen. Zum Beispiel, das Luft-Meilen-Applet kann folgende Überprüfung in der `getShareableInterfaceObject` Methode ausführen.

```
public class LuftMeilenApp extends Applet implements LuftMeilenInterface{
    public Shareable getShareableInterfaceObject(AID client_aid, byte parameter){
        if(client_aid.equals(brieftasche_app_aid_bytes, (short)0,
(byte)(brieftasche_app_aid_bytes.length)) == false)
            return null;
        if(parameter != SECRET)
            return null;
        return (this);
    }
}
```

Wenn eine shareable Interface-methode wieder aufgerufen wird, muß der Server das Client-Applet nochmal überprüfen. Diese Vorsorge ist sehr nützlich für die Sicherheit. Denn z.B. ein Client-Applet hat einen SIO bekommen und verbricht dann die Verbindung und gibt den SIO einem dritten Applet ohne korrekte Erlaubnis. Das Server-Applet muß solche Situation verhindern, damit keine geschützten, sensitiven Daten vom unzuverlässigen Applet erhalten werden. Um das AID vom aktuellen Aufrufer herauszufinden, kann das Server-Applet die Methode `JcSystem.getPreviousContextAID` benutzen. Die folgenden Codes zur Überprüfung der Identität des Aufrufers müssen in die `grantMeilen` methode hinzugefügt werden:

```
public void grantMeilen(short amount){
    //das AID des Aufrufers bekommen
    AID client_aid = JcSystem.getpreviousContextAID();
    //überprüfen, ob die methode richtig vom Brieftasche-Applet aufgerufen wird
    if(client_aid.equals(brieftasche_app_aid_bytes, (short)0,
(byte)( brieftasche_app_aid_bytes.length)) == false) ISOException.throwt
(SW_UNAUTHORIZED_CLIENT);
    meilen = (short)(meilen + amount);
}
```

In den vorangehenden Codes wird ein Authentisierungsschema zur Sicherung ausgeführt. Aber ein solches Schema ist nicht hinreichend sicher für ein Applet, das eine höhere Sicherheit verlangt. In diesem Fall muß eine zusätzliche Authentisierung hinzugefügt werden.

Die folgenden Codes implementieren ein Authentisierungsschema, das sogenannte challenge-response Schema in dem Brieftasche und Flugmeilen- Beispiel. Wenn die Servicemethode `grantMeilen` aufgerufen wird, erzeugt das Luft-Meilen-Applet eine gelegentliche

Prüfungsphrase und sendet diese Prüfung zum Briefetasche-Applet. Das Briefetasche-Applet verschlüsselt die Prüfung und schickt eine Antwort zum Luft-meilen-Applet. Durch das Überprüfen der Antwort authentifiziert das Luft-Meilen-Applet das Briefetasche-Applet und addiert die aufgeforderte Meilen zu seiner Bilanz.

Um das Schema zu verwirklichen müssen zuerst in der `grantMeilen` Methode zwei zusätzlichen Parameter hinzufügen – ein AuthentisierungsObjekt und ein Buffer.

```
public interface LuftMeilenInterface extends Shareable{
    public void grantmeilen(AuthenticationInterface authObject, byte[] buffer, short amount);
}
```

Das Luft-Meilen-Applet authentifiziert das Briefetasche-Applet durch Aufruf der Methode `challenge` vom Authentisierungsobjekt. Der Buffer ist verwendet für Transport der prüfung und der beantworteten Daten.

```
public class LuftMeilenApp extends Applet implements LuftmeilenInterface{
    public void grantmeilen(AuthenticationInterface authObject, byte[] buffer, short amount){
        // eine gelegentliche Prüfungsphrase im Buffer erzeugen
        generateChallenge(buffer);
        //prüft das Client-Applet
        //die Antwort ist in den buffer gespeichert
        authObject.challenge(buffer);
        //die Antwort prüfen
        if(checkResponse(buffer) == false) ISOException.throwIt
(SW_UNAUTHORIZED_CLIENT);
        meilen = (short)(meilen + amount);
    }
}
```

Der Authentisierungsobjekt ist vom Briefetasche-Applet erzeugt und gehört auch ihm. Das Applet-Firewall verlangt, dass solcher Objekt ein SIO werden soll:

```
public interface AuthenticationInterface extends Shareable{
    public void challenge(byte[] buffer);
}

public class BriefetascheApp extends Applet implements AuthenticationInterface{
    public void challenge(byte[] buffer){
        //Antwort bekommen
        //die prüfung und die beantworteten Daten sind in den Buffer gespeichert
        getResponse(buffer);
    }
    public void process(APDU apdu){
        if(getCommand(apdu) == DEBIT)
            debit(apdu);
    }
}
```

```

    }
    private void debit(APDU apdu){
        short amount 0 getDebitAmount(apdu);
        //die Bilanz updaten
        balance = (short)(balance – amount);
        //dem Luft-Meilen-Applet eine Meilenaddierung anfordern
        requestMeilen(apdu.getBuffer(), amount);
    }
    private void requestMeilen(byte[] buffer, short amount){
        //den AID-Objekt bekommen
        AID luft_meilen_aid = JCSYSTEM.lookupAID(luft_meilen_aid_bytes, (short)0,
(byte) luft_meilen_aid_bytes.length);
        //das SIO vom Luft-Meilen-Applet auffordern
        LuftMeilenInterface sio = (LuftMeilenInterface)
(JCSYSTEM.getAppletShareableInterfaceObject(luft_meilen_aid, SECRET));
        //dem Luft-Meilen-Applet eine Meilenaddierung anfordern
        sio.grantMeilen(this, buffer, amount);
    }
}
}

```

In diesem Beispiel ist der APDU-Buffer verwendet für Transport der prüfung und der beantworteten Daten. Der APDU-Buffer ist global und kann von allen Kontexten zugegriffen werden.

### 2.5.11 Die `getPreviousContextAID` Methode

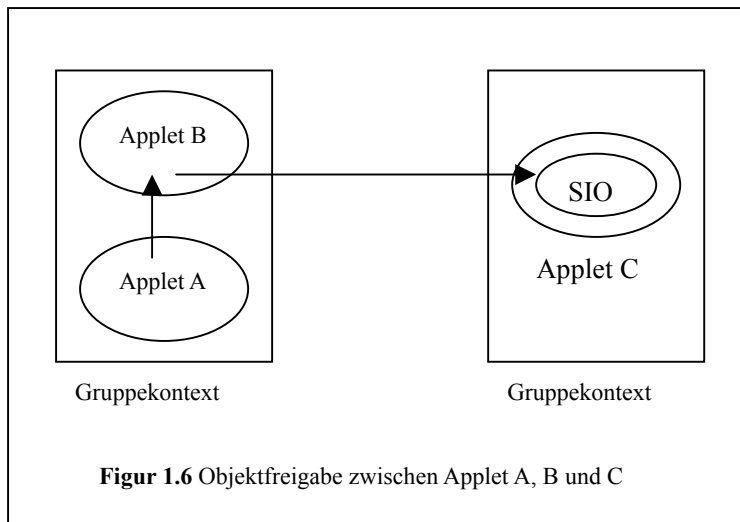
Während der Objektfreigabe kann der Server das AID vom Client-Applet durch Aufruf der Methode `JCSYSTEM.getPreviousContextAID` Methode herausfinden:

```
public AID getPreviousContextAID()
```

Diese Methode gibt das JCRE-AID zurück, das sich mit dem bei der letzten Kontextumwandlung aktiven Applet verbindet. Im letzten Beispiel, wenn das Brieftasche-Applet die shareable Interface methode `grantMeilen` aufruft, wird eine Kontextumwandlung durchgeführt. Die `getPreviousContextAID` Methode gibt das AID vom Brieftasche-Applet, das vor der Kontextumwandlung aktiv war.

Wir betrachten jetzt einen komplizierteren Fall, wie es in der Figur 1.6 gezeigt wird. Es gibt zwei Applets A und B mit einem gemeinsamen Gruppenkontext. Keine Kontextumwandlung wird durchgeführt, wenn Applet A eine Methode vom Objekt b ( gehört dem Applet B) aufruft. Diese Tätigkeit wird unabhängig davon erlaubt, ob Objekt b ein SIO ist oder nicht.

Aber jetzt, wenn Objekt B einen SIO vom Applet C zugriff, wird eine Kontextumwandlung ausgeführt. Das aktive Applet bei der letzten Kontextumwandlung war das Applet B, und deshalb gibt die `getPreviousContextAID` Methode das AID vom Applet B zurück.



### 2.5.12 Zusammenfassung

1. Wenn das Server-Applet A ein Objekt mit anderem Applet freigeben will, muß es zuerst ein Shareable Interface SI definieren. Ein shareable Interface ist vom Interface `javacard.framework.Shareable` vererbt. Die Methoden, die im SI repräsentiert werden, sind diejenigen, die das Applet A den anderen Applets zugreifbar lassen will.
2. Das Applet A definiert dann einen Serviceanbieter, nämlich Klasse C, die das SI implementiert. C bietet eine tatsächliche Implementation von den in SI definierten Methoden. C kann auch andere Methoden und Felder definieren, aber die sind vom Applet Firewall geschützt. Nur die im SI definierten Methoden sind zugreifbar für die anderen Applets.
3. Das Applet A erzeugt eine Objektinstanz 0 von der Klasse C. C gehört zum Applet A und das Firewall erlaubt den Zugriff von A an allen Felder und Methoden von C.
4. Wenn ein Client-Applet B den Objekt 0 vom Applet A zugreifen will, ruft es die `JCSystem.getAppletShareableInterface` Methode zur Aufforderung einen SIO vom Applet A auf.
5. Das JCRE sucht in seiner internen Applettabelle nach dem Applet A. Wenn es A gefunden hat, ruft es dann die `getShareableInterfaceObject` Methode auf.
6. Das Applet A bekommt die Aufforderung und prüft, ob A dem Applet B den Objekt freigeben will. Wenn A mit der Freigabe mit B einverstanden ist, antwortet A dann B mit einer Referenz zu 0.
7. Das Applet B bekommt die Objektreferenz vom Applet A, castet sie zum Typ SI, und dann speichert in den SIO. Obwohl das SIO wirklich den Objekt 0 vom A anspricht, ist es noch vom Typ SI. Nur die im SI definierten shareable Interface Methoden sind sichtbar von B. Das Firewall verhindert, dass B auf andere Felder und Methoden von 0 zugreifen kann. Das Applet B kann Service vom Applet A auffordern, indem B eine von den SI-Methoden vom SIO aufruft.
8. Vor der Ausführung des Services kann die SI-Methode das Client(B) autehtisieren, um festzustellen, ob der Service bewilligt wird.

## Literatur

[1] Lit.: Chen: JavaCard Technology for SmartCards. Addison-Wesley 2000.

[2] Lit.: Bruce Eckel: Thinking in Java. 04.1999

[3] Lit.: Uwe Hansmann: Smart Card Application Development using Java. Springer Verlag  
20.Oktober.1999

[4] Lit.: Die Spezifikation von Java Card Technologie. <http://java.sun.com/products/javacard/>