Object Lifetimes in Java Card

Marcus Oestreicher
Zurich Research Laboratory, IBM Research Division

Ksheerabdhi Krishna
Austin Product Center, Schlumberger

Object Lifetimes in Java Card

Marcus Oestreicher
Zurich Research Laboratory
IBM Research Division
Rueschlikon, Switzerland
oes@zurich.ibm.com

Ksheerabdhi Krishna
Austin Product Center
Schlumberger
Austin, TX 78726
kkrishna@slb.com

Abstract

Java Card promises the ease of programming in Java to the world of smart cards. Java's memory model however is resource intensive especially for smart card hardware. Hence, adapting Java's memory model to Java Card must retain the easy programming paradigm while enabling Java Card applications to maximize the use of smart card memory. To this end, the Java Card 2.1 Specification [3] advocates an ad hoc persistent memory model that foists an unnatural programming paradigm and an inherently limited API.
In this paper we discuss memory model choices for Java Card in the context of persistent systems. We propose the concept of a transient and persistent environment for encapsulating the transient and persistent objects in Java Card applications. While offering a simple programming model, it allows efficient sharing of the memory resources among multiple applications and enables garbage collection for Java Card.

1 Introduction

The Java [1] environment possesses a number of features that make it's adoption to smart cards attractive. A reasonable subset of the Java environment can constitute the base for a multifunction smart card platform. Applications can rely on standardized APIs and may be compiled into an intermediate bytecode representation enabling execution by a Java virtual machine. The simplicity and wide acceptance of the Java programming language is attractive to the smart card community where no standard language suitable for developing multifunction smart card applications has yet been established. Finally, they can benefit from the platform independence and security features provided by the Java environment.

However, Java's platform independence places significant constraints on the specific target platform. Java provides automatic memory management and does not foresee any means of manual memory control. In particular, it does not provide explicit support for persistent objects. In contrast, a smart card application requires random access to both transient and persistent memory. The limited amount of the memory resources available and their physical characteristics require simple and transparent manipulation of objects in both types of memory by the application. Hence, one of the challenges in the design of the Java Card is adapting Java's memory model to the constraints of smart card hardware.

Systems that require an intimate composition of long-lived data and programs are called *persistent application systems* [6]. Since applications in a Java Card are coupled with data, it enables the card to be a persistent application system. Orthogonal persistent systems [6] specially provide an appealing programming model for application development. Current smart card hardware typically offers around 16K bytes of persistent memory and nearly 1K bytes of transient memory. Given such constraints a Java Card cannot provide the same degree of transparency and orthogonality as an orthogonal persistent system. While it may not be desirable to completely hide the lifetimes of objects in the Java Card, it could be done in tandem with handing some control to the programmer. The necessary restrictions must be carefully chosen to provide a high degree of programmer convenience while enabling a reasonable utilization of the resources available in a smart card.

In this paper, we present the choices underlying different Java Card memory models. In particular, the differences, advantages, and drawbacks of each are highlighted. The notion of the transient and persistent environment is introduced as a solution toward simplifying Java programming on smart cards, data sharing, and support for efficient garbage collection.

The paper is organized as follows. Section 2 describes the typical memory layout in smart cards and introduces the basic execution model of applications in a Java Card. Section 3 lists the different lifetimes of objects which result from the given execution model and presents plausible allocation and placement strategies.

Section 4 discusses concepts underlying persistent systems in general and their applicability to smart cards. Section 5 outlines the approaches to introduce transience and persistence in the Java Card which were discussed during the Java Card specification process. Section 6 introduces the basic concepts and usage of transient and persistent environments. Section 7 discusses the implications of the transient environment on security, memory reclaiming and object sharing contrasting it with the approach taken by the Java Card 2.1 Specification. Finally, we present our conclusions in Section 8.

2 Smart Card Memory and Java Card Basics

Current and upcoming smart card hardware provide very limited storage capabilities. The memory resources typically consist of Read Only Memory (ROM), Random Access Memory (RAM) and Electrically Erasable Programmable Read Only Memory (EEPROM). EEPROM is used to store long-lived data. In contrast, RAM loses its contents after a power loss and is thus only available for temporary storage. Both EEPROM and RAM can be read and written; however, write operations to EEPROM are typically thirty times slower than to RAM and the possible number of EEPROM writes over the lifetime of a card is physically limited. Another difference lies in the physical size of each of these memory types. The physical size constraints on a smart card dictate RAM/EEPROM ratios often resulting in considerably smaller RAM in comparison to EEPROM.

A typical Java Card design places the operating system, the virtual machine and one or more applications in ROM. EEPROM is used to store applications which have been loaded after a card has been issued. Java Card applications, also referred to as *applets*, correspond to Java packages and are not loaded as regular Java class files [2]. Instead, a converter coalesces all the class files that comprise the package into a compact representation with minimal symbolic information. The converted code is linked on the card against the system classes and other required packages. It is up to the converter and the virtual machine to assure the Java language protection rules for downloaded code.

Once downloaded the applet is installed in a separate step. The virtual machine calls the mandatory *install* method which allocates required resources and registers a persistent object, the applet instance, with the Java Card runtime environment for future invocations. Applet execution is tailored around the server centric nature of a smart card and takes place during sessions. When a card is placed in a card acceptance device (CAD) the runtime is initialized and awaits input. Communication is handled by the underlying operating system. An external application (the client) initiates a session with a specific applet (the server) by sending a *select* command to it. The runtime marks the selected applet active and forwards the command by invoking the applet's select method. Each command sent by the client hence is forwarded and handled by invoking the applet's *process* method. The applet processes the command and prepares a response which is sent after the applet has returned from its invocation. A session ends when a new select command is received. The runtime deselects the currently selected applet by invoking its *deselect* method and initiates a session with the newly selected applet.

During a session, an applet can access both the services of linked packages and services exported by other applets. A client applet can ask a server applet for a service by obtaining a reference to a shared object and invoke it freely during the session. When invoked, the server object can check the identity of the caller and grant the requested service. Note that the Java stack is completely unwound upon cessation of communication with the CAD.

3 Object Lifetimes

The object allocation, placement and invocation model influences the lifetime of objects in the system. Object lifetimes in persistent systems fall into one of the following general categories [4]:

1. *Transient (temporary) results in expression evaluations and local variables in procedure activation.*
   Data in this category resides in the individual bytecode frames and is of a primitive type. Java and Java Card do not allow the explicit allocation of objects on the stack which is especially limited on the Java Card. The stack contents must only be valid during applet invocation in a session.
2. *Instance variables, class variables and heap items whose extent is different from their scope.*
   Among these items are especially objects which must be accessible during applet invocation or the entire session. They must not be saved in case of power loss. For example, objects of this type are used to store the state of the communication, session keys or the communication buffer.
3. *Data that exists between two program executions.*
   Objects in this category cover data which must be stored in EEPROM to survive a power loss.
4. *Data that exists between different versions of a program or data that outlives the program.*
   The Java Card environment currently does not address this category.

The object lifetime categories and the memory types in smart cards give rise to the three following allocation strategies for objects in a Java Card:

1. *Objects are instantiated in RAM and are serialized by the applet into EEPROM via a file system etc.* This strategy resembles the regular Java environment and of early Java Cards [11]. The applet instantiates objects in RAM and stores their data with the help of specific API functions into the long-term store. Each applet has to implement the functionality required for serializing and deserializing its state. There are two drawbacks with this approach. First, the working set of an applet could be larger than the available RAM. Second, the underlying operating system must manage the contents of the long-term store, i.e., it must provide a name binding with the serialized state and a means of checking the access rights of applets.
2. *Object instantiation is always in EEPROM.* If objects are instantiated in EEPROM the language protection rules can be used to verify the integrity of the system.This provides an uniform access to all the objects of an applet. This strategy is attractive since most data manipulations are performed on long-lived data. However performing all allocations in EEPROM may never be feasible since writes to EEPROM are extremely expensive and it's life limited.
3. *Object instantiation is in RAM and EEPROM.* The instantiation of objects with limited lifetime in RAM saves space in EEPROM, increases the performance and adds additional security in case of sensitive objects like session keys which must get lost in case of power loss. Long-lived data is placed appropriately in EEPROM. The utilization of both stores for object allocation and manipulation should still aim at the benefits of object instantiations only in EEPROM, i.e., allowing uniform access to objects and relying on Java's language protection rules.

4 Persistent Systems

The Java environment is designed as a transient programming environment. The Java Card environment however must support access to both transient and persistent objects within Java language expressions. Persistent programming languages and environments strive to enable such manipulations *transparently* [5]. Language expressions manipulating persistent data are made to appear similar to expressions operating on data with shorter lifetimes. Other than transparency, persistent systems provide different degrees of data type *orthogonality*. Full orthogonality expresses the demand that any instance can be persistent regardless of its type and that its lifetime may not be expressed at instantiation time [6]. In any case, the lifetime of data must be easily expressible by the programmer and persistent data must be *identifiable* as such by a simple and consistent mechanism. The principles of transparency, orthogonality and identification serve as the basis for persistent systems.

Orthogonal persistent systems offer the highest degree of transparency and orthogonality and were hence chosen as the basic architecture for adding persistence to Java in the PJama project [7]. PJama allows any instance to be persistent regardless of its type and any object is identified as being persistent by verifying reachabilty from a persistent root set. Persistent objects are always manipulated in RAM and are lazily stabilized into the long-term store.

5 Proposed Approaches

In a Java Card objects allocated in RAM must be immediately copied into EEPROM when assigned to a persistent reference. Otherwise, unexpected power losses will lead to illegal references and loss of data integrity. Since the working set of the applet can exceed the available RAM it can only be used as a cache. However, the extremely limited resources on a smart card make it impossible to determine a suitable caching scheme which delivers sufficient results without assistance from the programmer. It is worth noting that even large orthogonal systems tend to be inefficient [10].

To counter inefficiency and provide control to the programmer, some persistent systems limit the extent of one or more principles of orthogonal persistent systems, i.e., they may restrict transparency or orthogonality [5]. The programming style is affected as little as possible. This is crucial for the Java Card which touts the simple and popular programming style of Java. Hence changes to the language to support persistence are prohibited. Introducing lifetime aware bytecode instructions to the virtual machine must be avoided as they would hinder upgrading to another memory model.

5.1 Transient Types

A common approach to introducing persistency in statically typed languages is to make persistency dependent on type. Persistent objects must be instances of classes inheriting from a specific superclass or must implement a specific interface. One proposal advocated a *Persistence* interface causing implementing class instances to be allocated in EEPROM, all other class instances would reside in RAM. Alternatively, another approach proposed a *Local* interface to mark class instances to be allocated in RAM, with unmarked instances allocated in EEPROM.

There are two problems with this approach. Firstly, references to transient objects in the persistent set lead to dangling pointers in case of sudden power losses. Since an applet acquires access to its state by its persistent instance at invocation time, it is required to store a transient reference in its persistent set as soon as it uses a transient object. Resetting dangling pointers at the beginning of a card session involves complex scanning and is time consuming due the writes required to EEPROM. Secondly, it requires having two type hierarchies for classes whose instances may be transient or persistent. This especially restricts the use of array objects which can either be persistent or transient but never both. Additional classes simulating the behavior of fixed built-in types maybe required as well. The resulting code bloat and the performance penalty incurred by wrapper classes makes this approach unacceptable for smart cards.

5.2 Transient Fields

The introduction of separate type hierarchies may be avoided by using or extending particular language features [8][9]. Changes to the language are forbidden in Java as it affects programming style, requires educating programmers and forces changes to the Java compiler. Java however provides a *transient* keyword, a field modifier that affects object serialization. Fields marked transient are not part of the persistent state of the encapsulating object and are not serialized. It seems natural to reuse the transient modifier in Java Card to mark fields whose data must reside in transient memory and must never be saved in the persistent image of the object. The advantage of this approach is that the persistent set is only connected with the transient object set at the location of the transient fields. This allows for simpler and efficient implementations for resetting the persistent set.

The main drawback of using transient fields is it's inability to express transience in a consistent manner. The value of a reference type transient field is the reference itself. Since the transient keyword does not indicate the lifetime of the referenced object it's meaning must be extended to include the transience of the referenced object. The extended definition fails to specify the lifetime of an object which is referenced by a transient and a persistent field as well. Changes to Java semantics also prevents future introduction of the transient keyword with uniform semantics in Java Card.

5.3 Transient Data (Java Card 2.1 Specification)

Since the main requirement is to disable storing sensitive and data requiring fast access to long-term store a memory model may allocate data associated with certain objects in short-term store. The current Java Card 2.1 Specification follows this approach and is based on two basic design decisions. Firstly, applets must be designed to not expect any form of memory reclamation on the card. Applets are required to instantiate needed data at installation time and reuse it throughout their lifetime. Unreferenced data cannot be expected to be reclaimed and therefore new allocations may fail. The second design decision is to avoid dangling pointers in case of sudden power loss by expecting all objects to be referenced persistently via EEPROM. Only the data of arrays of primitive types can be allocated in RAM. As the *new* bytecodes allocate objects in the persistent store special static factory methods *makeTransientBooleanArray*, *makeTransientByteArray*, and *makeTransientShortArray* are used for allocating only the object header in EEPROM and the data in RAM.

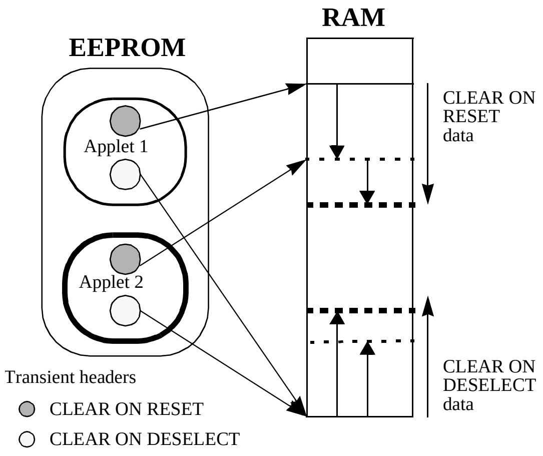

Fig. 1: RAM usage between Applets

Although the data part of such objects in transient memory is lost at power loss, the location and size information in the persistent header reserves its space over multiple card sessions. However, as RAM is shared by all available applets on the card it must be possible to limit the reservation of the transient data to less than a whole card session. Otherwise the entire RAM may be reserved after the installation of a few applets allocating transient data. The factory methods therefore take an additional argument, the duration identifier which either allows instantiations of arrays with ‘‘clear on reset’’ or ‘‘clear on deselect’’ duration. Arrays with the ‘‘clear on reset’’ duration will keep their values during the whole card session, i.e., their contents are not reset between multiple applet selections during a card session. The value of arrays with the ‘‘clear on deselect’’ duration are always cleared before their owning applet is selected. This coarse grained lifetime specification allows overlapping the ‘‘clear on deselect’’ RAM space across different applets. Figure 1 shows an organization of the transient memory using this overlap. RAM is split into two segments, the ‘‘clear on reset’’ and ‘‘clear on deselect’’ spaces which grow in opposite directions. Each newly allocated “clear on reset” array gets allocated from the globally reserved “clear on reset” space while each newly allocated “clear on deselect” array gets allocated at the beginning of the per applet “clear on deselect” space. The runtime must ensure that these two spaces do not overlap.

A consequence of this design is that the entire ‘‘clear on deselect’’ space must be cleared at once before a new applet is selected. If a package contains more than one applet the ‘‘clear on deselect’’ space has to be shared by all applets as they are allowed to share their transient arrays as per the Java Card 2.1 Specification. On the other hand, an applet shall allocate all its needed data, especially its worst case usage of transient data, at install time. This can easily lead to the situation where the first applet in a package, having access to a sufficient amount of memory, will install successfully whereas the installation of a cooperating applet in the same package may fail. The problem is compounded as the runtime is unable to verify the usage of two very important memory resources pertaining to the applet, the required stack space and the worst case usage of the transaction buffer. These resources cannot be pre-calculated as they depend on the card specific bytecode frames, implementation of the required packages, and the implementation of the transaction mechanism in case of the transaction buffer. This leads to a programming model where the programmer is forced to allocate some memory resources in advance but still needs to check for their unavailability at runtime.

The lack of transient objects and the restriction on transient arrays especially effects the convenient Java programming model. A programmer is forced to load and store values from persistent objects into transient arrays and vice versa. Additionally, since resources are limited and differ from card to card, programmers have to code to a least common dominator and cannot rely on the flexible use of transient data. Not only will this force programmers to use entirely persistent objects, the larger RAM capacities of upcoming smart card hardware will ironically remain unused. The different handling of transient and persistent data also causes different transient and persistent object layout due to the indirect access to transient data through a persistent header.

6 Transient Environment

Monk: Why would anyone hurry to create gardens and buildings and monuments?
Lord Buddha: Everything is transient and nothing endures.

The problem of persistent and transient objects in Java Card can be stated as follows: *How can the persistent object model of Java Card be augmented to enable flexible use of transient objects?* The solution surprisingly lies in introducing a mechanism that is symmetrical to the persistent environment. The persistent environment consists of a persistent root, the applet instance and a set of objects reachable therefrom allocated in EEPROM. Analogously, the transient environment is formed by a tree of objects with the difference that all objects reside in RAM.

Both environments are separated in as far as references to transient objects are **forbidden** to be held in the persistent set. Storing references to transient objects in the persistent store is prevented by the virtual machine (VM) which throws an exception when such assignments are attempted. The Java (Card) bytecode instruction set uses different instructions to store primitive and reference types; only the latter must be checked by the VM, making the overhead negligible compared to the necessary EEPROM write operation in case of a store. Assignments of persistent references in transient stores need not be checked as it does not affect the integrity of the persistent store. Dangling pointers in case of power losses are avoided as transient references are never stored in the persistent set.

```
public class DummyApplet extends Applet {
   public boolean select() {
      JCSystem.beginTransience();
       Object o = new Object();
      JCSystem.endTransience();
      JCSystem.setTransientEnvironment(o);
      return true;
   }

   public void short process(APDU apdu) {
      Object o;
     o = JCSystem.getTransientEnvironment();
      // use o
   }
}
```

Fig. 2: Using Transient Environments

The lifetime of an object is controlled with the help of an API mechanism that demarcates transient allocation defaulting to persistent allocation when not called. As shown in Figure 2 the API is used to toggle the allocation mode. Allocations between *beginTransience* and *endTransience* are in RAM, defaulting to EEPROM.

RAM overflow is signaled by an exception similar to how EEPROM overflow is signalled. This mechanism resembles a typical transaction API wherein behavior of state manipulation is toggled between a *beginTransaction* and *commitTransaction* code section.

Allocated transient objects may be interconnected to form a transient environment whose root can be registered with the Java Card runtime by invoking the *setTransientEnvironment* method. Where a particular persistent environment is implicitly made available to the applet during invocation of the process method, the applet can request its transient environment by explicitly invoking the *getTransientEnvironment* method. This allows accessing transient objects even when assignments of transient object references to persistent fields are prohibited.

The transient environment naturally fits in the current execution model of Java Card applets. It may typically be built in the beginning of a session in the select method. Transient data which must survive the session is copied to the persistent store at the end of the session. After the session, the transient environment is reset and the designated RAM space is available for the next applet session. The ease of creating transient objects of any type leads to a convenient programming style, results in better performance and more compact code. The similarities between the transient and persistent environment also simplifies virtual machine implementations where the same object layout may be used for both the transient and persistent objects.

7 Implications

Apart from enabling a convenient programming mechanism the transient environment scales into the future when more smart card resources become available. The only limitation introduced by the transient environment is the restriction on assignment. Future systems may choose to remove this restriction and still provide binary compatibility. The restricted assignment also enhances security in that an applet cannot store a reference to a transient object which it received from the system or from a different applet in its persistent set. Since the object is only accessible during a session it cannot be accessed in situations not foreseen by the service provider. For instance, it allows the deletion of a server applet without the danger of dangling pointers in the client applet. The transient data approach is not upgradable in large part due to the fixed lifetimes in the API. The transient space is statically split for the individual applets which make temporary allocations and deallocation in the future practically impossible. Lifetimes contradict the ease-of-use of standard Java.

The transient environment also strengthens other aspects of the Java Card runtime, especially memory reclamation and the sharing mechanism.

7.1 Memory Reclamation

While not addressed by the Java Card 2.1 Specification memory reclamation can be very effective for the smart card environment. Clearly manual memory reclamation cannot be allowed due to the security implications and must be avoided in favor of garbage collection. Surprisingly, EEPROM write performance and not the size of a general garbage collector is a hindering factor with regards to a memory reclamation scheme. For instance, a simple mark and sweep garbage collector first annotates all reached objects and frees all unreferenced objects in a second pass [12]. It is most likely due to the limited RAM size that the annotation information, i.e. the mark bits, must be written into EEPROM. The resulting performance penalties make it impossible to interrupt the applet execution for garbage collection as soon as memory is scarce but only at fixed points in time. However, cleaning up EEPROM is not as critical as that of RAM.

7.1.1 Transient Environment Garbage Collection

Typical applets allocate their persistent set at installation time and limit the changes therein to data update. New instantiations or complete replacement may be considered rare. The RAM space is limited and can be consumed quickly as soon as an applet allocates transient objects which are not reused in the transient environment but allocated only for the duration of an applet invocation. However, in case of transient environments garbage collection is not only feasible but indeed practical. The transient environment can be garbage collected completely separately from the persistent environment. Persistent objects need not be scanned as their fields never reference transient objects. Our implementation achieves satisfying results when invoking the garbage collector upon return from the applet's select, deselect, or process methods. The Java stack is empty and the root set for the garbage collection consists only of the transient environment.

The same garbage collector may be used for cleaning up the EEPROM. Due to the limited performance this should only be carried out after an explicit request by an applet or by a special command from an external application.

7.1.2 Limitations of Transient Data Memory Reclamation

EEPROM garbage collection is also permitted by the transient data approach. However, a different mechanism must be used for reclaiming RAM space. The runtime can attempt to reuse the space for globally allocated ‘‘clear on reset’’ arrays after their applets have been deleted. It may also allow the increase of ‘‘clear on reset’’ space after the applet with the most ‘‘clear on deselect’’ usage has been deleted. The added complexity stands in contrast to the low amount of memory which can be reclaimed generally in this static environment.

7.2 Sharing

The flexibility of the transient environment is not only afforded by the selected applet but also by the services it uses. The Java Card environment distinguishes and supports three different sharing scenarios:

1. *An applet is linked against a separate package.*
   The package contains shared code used by different applets to create instances of classes and invoke methods in this package.
2. *An applet has a reference to a shared object provided by a another applet*.
   The client applet requests the reference from the runtime which forwards the request to the serving applet and returns the received reference to the client applet. The reference must be an interface type and the virtual machine will refuse any other attempts to access the methods specified by the interface.
3. *An external application collaborates with two or more applets on the card.*
   The applets know about the collaboration and want to keep access to their transient state as long as the collaboration lasts.

The transient environment can provide a flexible use of transient data in all three scenarios. The degree of flexibility depends on the runtime providing support for only one transient environment, multiple transient environments and/or garbage collection.

7.2.1 Single Transient Environment

The transient environment plays well in the first scenario wherein a shared package can either be given access to the transient environment of its client applet or can build its own transient environment. In the first case, the applet and package transient environment must obey Java type rules which involves the applet subclassing its environment root class to the package environment root class. In the second case, where the transient environment of the applet and the package differ, the applet can use a simple mechanism to adopt its transient environment to contain the package’s transient environment. The applet has to reserve one node in its environment for the root of the package environment. The first time it calls into the shared package it saves its current environment in a local variable and resets its environment at the runtime. Within the call the shared package can create an environment for itself, register it and fulfill the requested service. Upon return the applet can save the package environment in its designated node and register its original environment. From now on the applet must always save its environment on the stack and register the package environment before it invokes the target package. If the runtime provides a garbage collector, both the applet and the shared package can allocate local objects which are not part of the environment and are garbage collected after the current invocation of the applet. As the applet is in control of the shared package environments, it can reset their roots any time during the session and thus subject them to garbage collection. This allows the optimization of memory usage for instance when an applet uses multiple packages alternately during a session. The same mechanisms apply in the second scenario.

The third scenario demands extending the transient environment lifetime over an applet session. This is achieved by either requiring the applet to not reset its transient environment at the end of the session or using a system method to retrieve it. The longer living environment can then be shared by the collaborating applets. They manipulate it alternately until the last deselected applet during the collaboration finally resets it. The runtime may then free the transient space for the next session.

7.2.2 Multiple Transient Environments

Collaborating applets may have different transient environments causing the runtime to support multiple transient environments at a time. Each collaborating applet creates and registers its own transient environment with the runtime system and extends its lifetime to last longer than its current session. The runtime switches between the transient environments whenever an applet is deselected and the next applet during the collaboration is selected. When the last deselected applet resets its transient environment the runtime releases the transient space. If a garbage collector exists, parts of the transient space can be reclaimed when any applet resets its transient environment.

The support of multiple environments can be useful in the second scenario to simplify programming and to encourage the full use of all available RAM. When a client applet requests a reference from a server applet, the request is forwarded by the runtime to the serving applet. The server applet creates and initializes its own transient environment, registers it with the runtime and returns a reference to the requested shared reference. The runtime marks the serving applet as being part of the current session and forwards the reference to the client applet. Whenever the client invokes a method on the server reference the runtime switches to the appropriate transient environment. The server object can access its transient environment and use it to execute the requested service. When the client applet is finally deselected, all transient environments of the participating server applets are also reset.

As opposed to the single transient environment the management of the individual environments is up to the Java Card runtime. This simplifies the programming model but limits the control of the executing applet over memory utilization. The Java Card runtime is not able to release any of the participating environments prior to the end of the session without the support of the shared services or the request of the currently selected applet.

7.2.3 Limitations of Transient Data Sharing

The static memory model of the transient data approach fails to provide a flexible use of transient data in any of the three scenarios.

For the first scenario, a shared package can only allocate transient data if it is called during the installation of the applet. Moreover, it must connect its transient data to the persistent set for later use.

For the second and third scenarios, both the shared object and the collaborating applets have to keep their transient information in globally reserved ‘‘clear on reset’’ arrays. A shared object cannot use the ‘‘clear on deselect’’ arrays of its implementing applet as the ‘‘clear on deselect’’ space is reserved for the currently selected applet, the client applet. This forces either the global reservation of RAM by allocating ‘‘clear on reset’’ arrays or the renunciation of transient data. In the first case shared applets can hog RAM causing denial of service problems by preventing installation of any client applet. In the second case the performance of EEPROM will prevent the sharing of any complex services and force each client to implement parts of or even the whole service, defeating code reuse.

8 Conclusions and Future Work

The contributions of this paper are:

- We suggest a terminology and framework with which to describe the issues underlying memory models in Java Card.
- We identify the restriction that can be placed on orthogonal persistent systems while retaining all the benefits of persistent systems for the Java Card programmer.
- We have presented other solutions attempting to solve the problems of transient data and shown their weaknesses. In particular we have shown that the static memory model described in the Java Card 2.1 Specification results in an unusual programming model and restricts the possibilities for memory reclamation and object sharing to an unsatisfying degree.
- We have shown that, by making support for transience explicit, a Java Card can provide a scalable API that can allow the manipulation of transient data similar to the standard Java environment. The resulting dynamic memory model naturally fits Java Card’s execution model, allowing the simple and effective deployment of a garbage collector and enhances object sharing.

The proposed environment has been implemented and tested on a number of applets. In the future we hope to provide concrete benchmarks supporting its simplicity and performance to enable comparison of the design choices. We hope the transient environment will further demystify smart card programming and permit Java Card programmers to truly enjoy the benefits of persistence.

9 Acknowledgments

We are deeply indebted to Peter Buhler and Michael Baentsch for their invaluable input on implementation and many productive discussions. We are grateful to Stephan Hild for carefully reviewing the paper.

10 References

[1] Arnold, K. and Gosling, J., *The Java Programming Language*, Addison-Wesley, 1996.

[2] Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification*, Addison-Wesley, 1996.

[3] Sun Microsystems Inc., *Java Card API 2.1 Specification*, //java.sun.com/products/javacard/JavaCard21API.pdf

[4] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R., *An approach to Persistent Programming*, Computer Journal, 26(4), 360-365, Nov. 1983.

[5] Hosking, A. L. & Moss, J.E.B., *Approaches to Adding Persistence to Java*, Proceedings of the First International Workshop on Persistence and Java, Drymen, Scotland, Sept. 1996.

[6] Atkinson, M.P. and Morrison, R., *Orthogonally Persistent Object Systems*, VLDB Journal, 4(3), 1995.

[7] Atkinson, M.P., Daynès, L., Jordan, M.J., Printezis, T. and Spence, S., *An Orthogonally Persistent Java*, ACM SIGMOD Record, Dec. 1996.

[8] Hosking, Anthony L. and Moss, J. Elliot B., *Compiler Support for Persistence*, COINS Technical Report 91-25, March 1991.

[9] Schuh, Dan, Cory, Michael and Dewitt, David, *Persistence in E revisited - Implementation Experiences*, Proceedings of the Persistent Object Systems Workshop, Martha's Vineyard, MA, September 1990.

[10] Cooper, Tim and Wise, Michael, *Critique of Orthogonal Persistence*, International Workshop on Object Orientation in Operating Systems, October 1996.

[11] Guthery, Scott. B., *Java Card: Internet Computing On A Smart Card*, IEEE Internet Computing, pp. 57-59, Jan/Feb 1997.

[12] Jones. R. and Lins. R., *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*, Wiley, 1996.