

Extensible Transaction Management in PJava

Laurent Daynès
Department of Computing Science,
University of Glasgow, Scotland
laurent@dcs.gla.ac.uk

Abstract

Persistent programming languages offer an attractive alternative to the increasing number of applications whose needs cannot be satisfied with traditional database support. The requirement of these so called non-traditional applications have prompted the development of numerous transaction models whose semantics vary from the traditional transaction model as well as from each other. In order to minimize the investment to realize new transaction models, application builders must be offered a simple framework they can use to quickly define the transaction behavior they want and incorporate it into the persistent programming system.

This paper reports on our effort to augment PJava, an alternative platform for the Java language, with such extensible transaction management features. The aim is to allow on demand integration of user-defined transaction model to PJava while maintaining transaction independence. This means that the body of transactions of an arbitrary transaction model can be programmed using any available Java classes without any alteration to the original source and compiled form of these classes.

1 Introduction

Persistent programming languages offer an attractive alternative to the increasing number of applications whose needs cannot be satisfied with traditional database support. The requirement of these so called *non-traditional* applications have prompted the development of numerous transaction models whose semantics vary from the traditional transaction model as well as from each other [11, 5]. The ever growing proliferation of transaction models, all unable to satisfy all needs, has definitively buried the hope of finding an universal model in the short term, if at all. In the absence of a proper transaction model, most persis-

tent application builders end up investing a significant amount of time developing in-house transaction models and find ways to get around the proposed transaction support in order to better accommodate the needs of their application.

In order to minimize the investment to realize new transaction models, application builders must be offered a simple framework they can use to quickly define the transaction behavior they want and incorporate it into the persistent programming system. Ideally, these extensions should not require their programmers to have a strong knowledge of the system's implementation of transaction processing abilities. Furthermore, each addition of a new transaction model should not make it necessary to rebuild the system. Instead, the system should be able to *adjust itself at runtime* to take these extensions into account. Lastly, the user's extensions should be tightly integrated with the system in order to minimize the impact on the system's global performance.

This paper reports on our effort to augment PJava, an alternative platform for the Java language [3], with such extensible transaction management features. The main goal of the PJava project is to leverage Java to support faster development and better maintenance of persistent and transactional applications via provision of *orthogonal* properties. Providing properties such as persistence and transaction semantics orthogonally has two benefits:

1. Application programming is not polluted with considerations unrelated to the application logic itself, such as persistence or enforcement of some transactional properties. In particular, programmers do not have to discriminate the data that may become persistent or may be used in a transactional way, or the code that may be used to manipulate persistent data or within transactions. The addition of the desired property (e.g, persistence, persistence + transaction), which depends on the operational context, is done by simple composition of the application code with some

context-aware code that encapsulates the particularities of the operational context (e.g., management of roots of persistence, monitoring of transaction execution).

2. Any Java classes can be used for building applications in a specific operational context (non-persistent Java, persistent Java, persistent and transactional Java) without any change to either the sources or the compiled form of these classes; no extra rewriting/pre-processing or code generation steps are necessary to execute standard Java classes in PJava and obtain persistence or transaction semantics. Conversely, the source and compiled form of any classes programmed with PJava can be re-used for any standard Java development environment and executed by any standard virtual machine, except for a minority of classes that encapsulate the use of built-in classes specific to PJava (the “context-aware” classes).

Hence, programming in PJava is not different from programming in Java while adding value such as persistence and transactions.

The current PJava prototype realizes orthogonal persistence: it adds persistence to the Java language with no perturbation to its initial semantics and definition, and enables the re-use of any normal Java classes for building persistent applications without any alteration to either the source or the compiled form of these classes. The reader is referred to [4] for an extensive definition of orthogonal persistence and to [1] for its application to the language Java. From the programmer’s point of view, persistence is simply obtained by composing normal Java classes with a few other persistence-aware classes (in most cases one) that interact with the class `PJavaStore` to identify the root of persistence objects, bind these root objects to the relevant application’s variables, and trigger the stabilization of updates¹ onto the persistent store when required by the application.

Our design to add extensible transaction management in Java follows a similar philosophy. Transactions are introduced in Java without changing the language definition and such that programmers don’t have to discriminate the data manipulated within transactions or the codes used in transactions, irrespective of the transaction model used. The aim is to allow the usage of any available Java classes to program the body of transactions of an arbitrary transaction model without any alteration to the original

¹Atomic propagation of updates onto the persistent stable store in PJava’s parlance.

source and compiled form of these classes.

In order to achieve extensibility, we augment the PJava virtual machine with a *Customizable Transaction Processing Engine* (or CTPE) whose behavior may be altered *at runtime*. The intention is to give knowledgeable Java programmers the ability to define new transaction models by programming customization of the CTPE in Java using predefined *building blocks*. Building blocks are objects that abstract the key mechanisms of individual CTPE’s components such as the lock or the recovery manager. They give expert programmers access to the low level mechanisms of the CTPE components without requiring any knowledge of the implementation of these components. Building blocks allow the programming of new transaction behaviors in a manner we believe is simple and safe. Ordinary Java programmers can then use the available transaction models conveniently in their applications.

The rest of this paper is organized as follows. Section 2 gives an overview of our design. Section 3 details the programming model offered to ordinary programmers. Section 4 describes the framework offered to define new transaction models and how arbitrary Java code may be composed freely with transactions of any model. The building blocks are discussed in section 5. We conclude with a summary of the status of our design and implementation plans.

2 Approach

Our design choices for augmenting PJava with extensible transaction management capabilities are led by three strong requirements:

- Safety. User-defined transaction models should not compromise the safety of the Java language.
- No change to the language definition.
- Transaction independence. Data and code used within a transaction must not differ from data and code used in a non-transactional context.

The following sections outline the three main principles of our design.

2.1 Transactions as Java objects

A transaction defines a unit of work for which some properties must be enforced. The basic interface common to all transaction models is made of functions for

demarcating the boundaries of transactions, such as the classic `begin/end/abort` bracketing.

Advanced transaction models extend this common interface with new operations (e.g., operations for restructuring the scope of transactions such as `split` and `join` [16], or for declaring a transaction as a member of a cooperative group [12]). Furthermore, the semantics of the same operation may vary from one transaction model to another. For instance, the operation `end` that indicates the successful termination of a transaction has different semantics whether it applies to a flat transaction, a sub-transaction in a nested transaction model, or a member transaction in a group transaction model [12]. In the classic flat transaction model, a successful termination requires the updates made by the transaction to be atomically and durably propagated on the persistent data, and made publicly visible; in a nested transaction model, the successful termination of a sub-transaction requires the updates to be atomically delegated to its parent transaction, and to be made visible to the descendant of its parent transaction only; in a group transaction model, the updates may be required to be atomically and durably propagated to the store and made visible to the transactions which are members of the same group only. This shows the need for a transaction management interface that is both extensible (introduction of new operations) and polymorphic (overriding of operation semantics).

Defining transactions as first-class objects allows the introduction of the transaction concept in Java without changing the specification of the Java language and provides a convenient framework for defining an extensible and polymorphic interface to transaction management. Transaction models are implemented as classes and their instances execute transactions according to the semantics their class defines.

Having transaction models implemented as Java classes also allows extension of the persistent system with new transaction models without requiring a rebuild of the system. Furthermore, transaction management inherits the dynamic and incremental properties of Java: new transaction classes can be incrementally introduced and used by existing applications without requiring these applications to be recompiled, as long as the functional interface of the new transaction model supports the operation required by the applications.

2.2 Two level interface

Our design provides Java programmers with two APIs corresponding to two levels of understanding

of transaction management. It presumes two categories of programmers: expert Java programmers with skills in transaction model specification who implement new transaction classes, and ordinary Java programmers who program transactional applications using the available transaction classes.

The intention is to lay out transactional applications in four distinct layers of increasing re-usability and independence with respect to transaction management issues. Figure 1 summarizes this layered approach (the size of each layer is not representative in terms of volume of classes).

The *external* API provides to ordinary application programmers a functional view of transaction management. It is intended for Java programmers who understand how to use the functional interface of the transaction model(s) best suited for their applications. They are responsible for the implementation of the transaction-aware portion of the application, which should account for a small portion of the application code. Transaction-aware classes typically encapsulate the creation of transaction objects, the definition of the boundaries of transactions, and the invocations of the methods specific to the transaction objects used (e.g., `split/join` for open-ended transaction models).

The transaction-aware classes isolate the rest of the application code from classes that depend on classes specific to the external API. Hence, above the logical software layer made of transaction-aware classes, there is no discernible difference from ordinary Java programming, except that methods execute transactionally when invoked from within a transaction. The classes implemented on top of the transaction-aware layer can be exported “as is” for execution on virtual machines supporting standard Java classes. An example of application programming using this layering is given in the appendix at the end of this document.

The programming model offered by the external API requires each transaction body to be organized into one or several `Runnable` objects, i.e., objects that implement the `Runnable` interface². `Runnable` objects are the basis for composing arbitrary Java code with arbitrary transaction objects. Composition via `Runnable` objects is similar to the approach taken for thread in Java and is a substitute to the absence of support for methods as first-class objects. The *Core Reflection* API promised with JDK 1.1 [15] will help

²An interface in Java specifies a collection of methods without implementing their bodies. When a class implements an interface, it must implement the bodies of all the methods described in that interface. Interfaces provide encapsulation of method protocols without restricting the implementation to one inheritance tree.

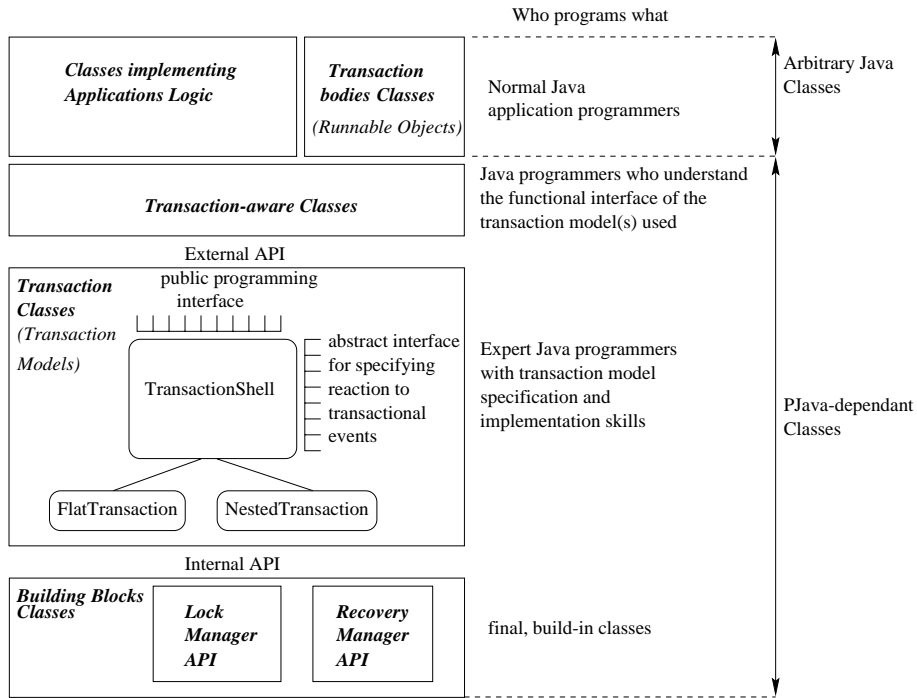


Figure 1: Extensible transaction management in PJava.

to limit the proliferation of `Runnable` classes. Figure 7 in the appendix shows how a single class can wrap arbitrary method invocation in a `Runnable` object. This is beneficial for both multi-threaded and transactional applications.

The external API itself consists of a hierarchy of transaction classes, each class implementing a given transaction model. The root of the hierarchy is the abstract class `TransactionShell`. It provides two sets of methods that correspond to the two levels of understanding of transaction management mentioned above. A first set of **public** concrete methods provides with an external interface for defining the boundary of a transaction irrespective of the model that transaction implements (see section 3). The methods of this interface are **final** and therefore cannot be overloaded. The methods of the second set are all **abstract** and **protected**. They define the reaction of the transaction model with respect to transaction management events that may occur during the execution of transactions (see section 4). These methods are part of the mandatory methods that a transaction model implementer must define for safety and completeness reasons. Ordinary application programmers are not exposed to these methods.

The class `TransactionShell` does not implement any transaction model; only its specializations do.

Specializations of the class `TransactionShell` may also augment the basic functional interface of transactions with new transaction management primitives specific to the model they implement.

The *internal* API provides an implementation view of transaction management. This API concerns the expert programmer who wishes to augment the set of available transaction models in order to satisfy new needs. The internal API consists of *building blocks* which may be used to implement a specialization of a `TransactionShell`. Building blocks are Java classes and interfaces that expose the visible functions of individual components of a *Customizable Transaction Processing Engine* or CTPPE. All the classes that compose the internal API are final for safety. The CTPPE currently exposes an interface to only two components: the lock and the recovery manager.

2.3 Automated enforcement of transaction semantics

None of the APIs proposed to the programmers contains functions related to the enforcement of transaction behavior. Examples of such functions include the acquisition of locks, the tracking of data dependencies between transactions, and the generation of recovery information.

Our design advocates the automation of these functions for the following reasons:

- it relieves programmers of onerous and error prone tasks such as setting locks and noting updates manually, which improves both safety and development-time.
- it promotes transaction independence by making both the sources and the compiled form of the code that implement the logic of applications free of any operations specific to the enforcement of transaction semantics. This limits the set of transaction-aware classes to the classes that directly use the methods of transaction classes.

Pre-processing techniques are inadequate for automating the enforcement of transaction semantics for two reasons: they may save the programmer from writing the code required for enforcing transaction semantics but does not fully provide with transaction independence since the pre-processor injects extra Java code that is ultimately part of the compiled code of the methods. Hence, though the programmer's work is simplified and transaction semantics are safely enforced, the code generated still depends on the operational context and cannot be re-used in a different context. The second reason for not choosing pre-processing technique is performance. Enforcement of transaction semantics, may be required at virtually every access to an object (especially in Java which does not enforce strict encapsulation). Planting one or several Java method calls in the user code, for acquiring locks or noting some updates, may be very inefficient.

Runtime compilation (or post-processing) techniques may solve the problem since both the sources and the generated codes remain in their original form and may be re-used in any context. They requires the augmentation of the instruction set of the virtual machine if code interpretation is used. Alternatively, if *just-in-time* technology is available, the runtime compiler may be modified for planting additional instructions for each interaction with the CTPE.

Another solution, which we have adopted in the short term, is to let the virtual machine identify at runtime when transaction semantics need to be enforced, and to interact directly with the relevant CTPE's component. The virtual machine keeps track of the transactional context assigned to each Java thread by the `TransactionShell` they are running for and uses it for interacting with the CTPE. This transactional context specifies to the CTPE the (possibly customized) semantics that must be enforced.

Because of safety, everything must run within the scope of a transaction in PJava. Furthermore, all data manipulations from within a transaction are subject to that transaction's behavior. Lastly, the enforcement of the transaction behavior is done automatically (as just explained). Hence, all data types are treated equally with respect to transaction management. This eliminates the need for introducing a discrimination between the objects that enforce transaction's properties from those that don't, and limits the pollution of application code with transaction management operations. If mechanisms have to be provided for enabling the escape of some objects from the control imposed by transactions, they should be provided in a way that is both orthogonal to the data type description and independent of the way the application code is written. We plan to incorporate such mechanisms in our design in the future.

In summary, our approach makes the programming of transactional applications independent of transaction management except for a marginal number of transaction-aware classes. Furthermore, the work of the application programmers in charge of the transaction-aware portion of the application just consists of choosing the right transaction model for their application, writing the code delimiting the boundaries of the application's transactions, and dealing with the transaction management issues specific to the transaction model chosen using the method of the corresponding transaction class.

3 Programming model

The choice of an interface for defining the boundaries of transactions raises two issues. First, the interface must be flexible enough to encompass the largest range of programming styles. As an example, consider a simple GUI application with a single frame and several buttons to control the execution of a transaction (e.g., start a new transaction, end it, kill it or execute the operations selected via the buttons on its behalf). The simple bracketing of an arbitrary block of code with markers such as "begin" and "end" is not sufficient to describe the boundary of the transaction in that case, since the body of the transaction may be composed of several action spread in the various event handling methods of the GUI application. Similarly, consider a back-end server that dispatches incoming requests to threads available in a pool. A given transaction may send more than one request, each being potentially dispatched to a different thread of the pool

each time. Here again, the requirement of the application cannot be satisfied with a simple “begin” / “end” bracketing.

The second issue is related to the confinement of errors within the boundaries of the transaction that made them. More specifically, any exceptions left uncaught in the body of a transaction must remain confined within that transaction and must be propagated to the failure handling mechanism defined for that transaction. Since the body of a transaction is made of arbitrary Java methods, a transaction body can spawn an arbitrary number of threads. This makes the detection and confinement of failure even more complex.

The class `TransactionShell` offers an uniform framework for defining the body of a transaction. This framework enables both procedural and event-driven programming styles and deals with arbitrary multi-threaded transactions. The example in Figure 2 illustrates these two styles³.

In both cases, transactions are defined by creating an instance of a transaction class. An instance of a transaction class is really just a shell in which to execute a transaction according to the model defined by that transaction’s class. A transaction is effectively created when the shell is invoked using its `start` method. If there is no ongoing invocation, a transaction object is nothing but an empty shell. When the invocation completes, the transaction object can be invoked again, starting another transaction.

In the procedural programming style, a transaction instance is directly associated with an object that satisfies the `Runnable` interface. The body of the transaction consists of the `run` method of the associated `Runnable` object. The `start` method of the transaction object triggers the execution of the transaction. The transaction terminates when the execution of its `Runnable` object ends (either normally or because of a failure). The result of the transaction may be obtained using the `claim` method of the transaction object. The `start` method is provided with both synchronous and asynchronous variant, and the `claim` method is provided with both blocking and non-blocking variants (see [2] for details).

In the event-based programming style, the transaction object is not directly associated with a `Runnable` object. Instead, the body of the transaction is made of all the `Runnable` objects that *enter* in the transaction between the boundaries explicitly defined by the programmer. When a thread calls the `enter` method

of a transaction object *t* with a `Runnable` object *o*, it executes *o* on behalf of *t*. When the `enter` method returns, the thread is said to *leave* the transaction, i.e., it enters back in the transaction it was before. No limitation is imposed on the number of threads that may enter the transaction concurrently.

Also, a thread implicitly enters in a transaction when it calls synchronously the method `start` of that transaction with a non-null `Runnable` object. Threads launched from within the body of a transaction (i.e., any of the `Runnable` object running on behalf of the transaction) are called *inner threads*.

A multi-threaded transaction terminates when the `end` method of its shell is invoked and all its inner threads as well as all the threads that entered the transaction prior to the call to `end` are completed. Entering a transaction in a terminal state kills that transaction and raises an exception to the thread that attempted to enter the transaction.

With the model just described, programmers are forced to specify the body of their transactions, or part of them, as `Runnable` objects, and cannot just bracket an arbitrary block of Java code with “begin” and “end” transaction marks. The rationale for this approach is to confine exceptions that are uncaught by transaction bodies to the limit of the transaction. By forcing the encapsulation of every piece of code that participates in the body of a transaction, a `TransactionShell` can catch all exceptions left uncaught by these transaction bodies simply by invoking the `Runnable` object within a `try / catch` Java block, and route the transaction execution to the code that deals with failures.

Achieving the same confinement of exceptions with an approach based on block delimitation make it necessary either to force the programmer to explicitly catch exceptions and trigger manually the appropriate action (e.g., abort the faulting transaction), or to change the language definition to incorporate transaction bracketing as suggested in [13]. Both options are incompatible with our requirements.

4 Transaction Shell

New transaction models are introduced by defining specializations of the abstract class `TransactionShell`. The `TransactionShell` is intended to make the definition of transaction classes simple and safe by:

- enforcing programmed transaction classes to conform to the uniform programming model defined

³The present example does not illustrate well the isolation of transaction-aware classes for the sake of conciseness. A better example is given in the appendix.

```

public class GUIApp extends Frame {
    TransactionShell ts;
    // ...
    // Procedural style
    public boolean Debit(BankAccount ba, int debit){
        // Define a flat transaction running a debit operation
        Runnable body = new DebitOp(ba,debit);
        ts = new FlatTransaction(body);
        ts.start(); // Start the transaction
        // return the transaction's status
        return (ts.claim() ==
TransactionShell.COMMITTED);
    }

    // Event-based style
    public boolean handleEvent(Event evt) {
        if( evt.id == Event.ACTION_EVENT &&
            evt.target instanceof Button) {
            String label = (String) evt.arg;
            if (label.equals(buttonBegin))
                ts.start(); // Initiate a new transaction in ts
            else if (label.equals(buttonEnd))
                ts.end(); // Complete ts's ongoing transaction
            else if (label.equals(buttonAbort))
                ts.kill(); // Abort ts's ongoing transaction
            else if (label.equals(buttonDeposit)) {
                Runnable job = new Job();
                // perform a job on behalf of
                ts.enter(job); // ts's ongoing transaction
            } else
                return super.handleEvent(evt);
            return true;
        }
        return super.handleEvent(evt);
    }
    // ...
}

```

Figure 2: Programming style examples.

by the public interface of the class **TransactionShell**. Any transaction, irrespectively of the model it implements, can then be composed with arbitrary **Runnable** objects.

- automating all the monitoring of transaction executions. The class **TransactionShell** relieves programmers from implementing the monitoring of all events that may occur during the execution of a transaction and impact its behavior.
- enforcing the definition of complete transaction behavior by requiring the programmer to (1) fill in mandatory methods that will react to transaction execution events that may happen during the execution of a transaction.
- using default system defined concurrency or recovery behaviors if not specified,
- using a default system defined recovery procedure if the user-defined one failed (i.e., is either incomplete or erroneous).

The class **TransactionShell** provides two sets of methods that correspond to the two levels of interface mentioned in section 1. The external interface is made of concrete public methods that implement the programming model described in the previous section. The internal interface is made of abstract protected methods. Each method corresponds to a reaction to a transaction execution related event. Declaring these methods as abstract forces the programmer to specify a reaction to these events and thus guarantees the completeness of the transaction class implementation. The class **TransactionShell** transparently detects the occurrence of these events and triggers the execution of the methods that implement the corresponding reaction.

Hence, the task of a transaction class programmer just consists of defining a concrete implementation for each of the **TransactionShell**'s abstract methods, and implementing the transaction management functions specific to the corresponding transaction model.

Table 1 lists the principal events sent to transaction objects⁴. There are two categories of events sent to transaction objects: *per transaction* events and *per participating thread* events.

Per transaction events concern the whole transaction. Such events include transaction initiation, normal termination, termination due to failure (e.g.,

⁴Events related to building blocks, such as conflict notification events, are sent to the building blocks rather than to the transaction objects they are assigned to.

deadlock failure, user initiated failure, etc..) and execution of inner transactions. Upon transaction initiation, the transaction class must react by assigning default building blocks for concurrency control and recovery management to the notified transaction object. If a default block is missing after transaction initiation, the notified transaction object is automatically provided with building blocks set to a default behavior (e.g., strict isolation for concurrency control).

Events related to the execution of inner transactions include invocation, successful termination or failure of inner transactions. Upon reception of a inner transaction event, a transaction object may react by inhibiting the transaction semantics of the inner transaction prior to executing its body. In this case, the inner transaction just executes as a normal method call. This may be useful for preventing the composition of transactions of different classes (namely if the interaction between the transaction model of the invoker and those of the invokee is unknown) or for enforcing the “flatness” of a transaction. Inhibition of inner invocations is controlled via a protected method of the class `TransactionShell`.

Per participating thread events concern individual thread that executes on behalf of a transaction. A thread participates in a transaction either because it has explicitly entered the scope of that transaction (via either the `enter` or `start` method of the public interface of a `TransactionShell` objects), or because it has been created within a transaction (see section 3). Events notifying the participation of threads and the successful or abnormal end of their participation are generated for each kind of thread.

Before a thread participates to a transaction, it must be assigned some transactional attributes. Updates book-keeper and locking capability (see section 5) are examples of such transactional attributes. These attributes are building block objects that define how the thread enforces the concurrency control and recovery behavior of the transaction they participate in. Assignment of transactional attributes must be done when the transaction object is notified of the participation of a thread. If no attributes are specified, the `TransactionShell` of the notified transaction object assigns default attributes to the thread. These default attributes are defined at transaction initiation time.

When a thread leaves the scope of a transaction, the class `TransactionShell` arranges for the automatic reinstallation of its previous transactional attributes.

Events related to transition of a transaction’s state	<code>notifyBegin</code> <code>notifyEnd</code> <code>notifyAbort</code>
Events related to inner invocations of transactions	<code>notifyInvokee</code> <code>notifyEndInvokee</code> <code>notifyFailedInvokee</code>
Events related to participant threads	<code>notifyThreadEnter</code> <code>notifyThreadLeave</code> <code>notifyFailedEnteredThread</code>
Event related to inner threads	<code>notifyStartThread</code> <code>notifyEndThread</code> <code>notifyFailedInnerThread</code>

Table 1: List of the principal events notified to a `TransactionShell`.

5 Building blocks

Building blocks are Java objects that provide interfaces to individual components of the CTPE; they are intended to ease the implementation of transaction classes. They give programmers access to the low level mechanisms of the CTPE components without requiring any knowledge of the implementation of these components. The aim is to give a great deal of flexibility for defining the behavior of a given transaction model while retaining safety and simplicity of usage.

Currently, building blocks are defined for two components only: the lock manager and the recovery manager. This section briefly reviews the building blocks currently defined for these two components.

5.1 The Recovery Manager’s building blocks

The Java interface to the recovery manager components is composed of the two classes `UpdateBookKeeper` and `Snapshot`. These classes allow programmers to cleanly and safely implement the recovery semantics of a transaction model.

A `UpdateBookKeeper` is an object that supports the control of the updates performed by a set of threads. All threads are required to be bound to exactly one `UpdateBookKeeper` at any time. Several threads can be bound to the same `UpdateBookKeeper` providing they run on behalf of the same `TransactionShell`. Each `UpdateBookKeeper` transparently maintains a history of all the updates performed by the threads they are bound to.

`UpdateBookKeepers` offer methods to make persistent or durable the updates they are responsible for,

transfer the responsibility of these, or a subset of these, updates (i.e., delegate updates) to other `UpdateBookKeepers`, and rollback updates using `Snapshot` objects.

5.1.1 Stabilization and Snapshots

A `Snapshot` is an object that defines a location in the history of updates maintained by an `UpdateBookKeeper`. A `Snapshot` may be used to rollback the updates recorded by a `UpdateBookKeeper` to the point where the `Snapshot` was created in the history. The effect of this rollback is to restore objects to the state they were in at the time of the `Snapshot`'s creation. Hence, from the point of view of a `UpdateBookKeeper` a `Snapshot` logically captures the state of all objects at the time of its creation.

A `Snapshot` provides two methods: one for restoring the state captured by the `Snapshot`, and one for disabling permanently the restoration of the `Snapshot`. An attempt to restore a disabled `Snapshot` raises a `DisabledSnapshotException`. `UpdateBookKeepers` may also be queried about all their available `Snapshots`, as well as their order in the history they maintain.

The restoration of a `Snapshot` consists of rolling back the undo log of the `UpdateBookKeeper` that generated this `Snapshot` to the record that stands for the `Snapshot` in the log.

By capturing several successive states of one `UpdateBookKeeper` using `Snapshots`, programmers can implement undo facilities. `UpdateBookKeepers` and `Snapshots` can be used to implement the recovery semantics of a transaction model, but can also serve more general purposes, such as undo mechanisms for editor-based applications, debuggers, versioning tools, etc.

`UpdateBookKeepers` provide a `stabilize` method which, when invoked, ensures that the updates they are responsible for are made persistent. They may not be durable though since the stabilized `UpdateBookKeeper` may still have some `Snapshots` available for rollback. The `stabilize` method may also take as a parameter a set of `Snapshots` to disable. Stabilizing the updates tracked by a `UpdateBookKeeper` while disabling all the `Snapshots` of that `UpdateBookKeeper` provides durability in the ACID sense.

5.1.2 Delegation of updates

The history of updates maintained by an `UpdateBookKeeper` may be re-structured using *delegation* of

updates. Delegation of updates allows one `UpdateBookKeeper` to *atomically* transfer the responsibility for an object's updates to another `UpdateBookKeeper`. Transferring the responsibility for some updates means transferring the ability to control the fate of these updates, i.e., whether they will be propagated onto the store or rolled back, etc. This effectively transforms the history recorded by the delegating `UpdateBookKeeper`, and allows updates to escape the control of an `UpdateBookKeeper` in order to fall under the control of another `UpdateBookKeeper`.

We speak of global delegation when one `UpdateBookKeeper` transfers at once all the updates it is currently responsible for, and partial delegation when one `UpdateBookKeeper` transfers only its updates on a selected set of objects. A global delegation merges all the delegator's information to those of the delegatee. Partial delegation just transfers the information corresponding to the objects that are delegated.

Global delegation of updates just needs the specifications of two `UpdateBookKeepers`: the delegator and the delegatee. The responsibility for all the updates of the former is given to the latter. Partial delegation requires a third parameter which enumerates the objects whose updates are delegated.

The nested transaction model [19, 14] is a well-known example of use of global delegation: upon commit, a sub-transaction does a global delegation of its updates to its parent transaction. The split-join transaction model [16, 20] is a straightforward example of use of partial delegation: a transaction selects a set of objects and delegates its updates on these objects to another transaction. The decision on which objects should be delegated is driven by the concurrency control.

5.2 The Lock Manager's building blocks

The Java interface to the lock manager components of the CTPE currently⁵ consists of the class `LockingCapability` and the `ConflictHandler` interface.

A locking capability, or capability for short, is a book-keeper of locks with a customizable conflict detection mechanism. Every thread must be bound to a capability, and several threads can be bound to the same capability (typically, all threads running on behalf of the same transaction are bound to the same capability). A thread can also change the capability it is bound to during its execution (typically when it enters a different transaction).

⁵We plan to open the management of deadlocks and provide programmers with a `DeadlockHandler` interface.

When a thread runs, the capability it is bound to automatically acquires the locks protecting the objects the thread operates on. Locks are acquired with respect to the conflict detection mechanism encoded in the capability. Transaction model implementors customize the conflict detection mechanism of each capability by specifying *ignore-conflict* relationships. Locking capabilities also provide methods to control the ownership of locks. The locks acquired by a capability can be either released all at once or delegated to another capability. These concepts are briefly reviewed below. A more complete discussion is given in [9].

5.2.1 Ignoring Conflicts

Transactions access and manipulate objects of the persistent store by invoking operations on them. Two operations are said to be *compatible* when they do not *conflict*. Two operations conflict if their effects on the state of an object or their return values (if any) are not independent of their execution order. When an invoked operation op_i conflicts with an operation op_j in progress, a dependency⁶ is formed if op_i is allowed to execute. Such dependencies reveal possible inconsistent states which may induce either an abortion of the dependent transaction or a specific commit ordering [7]. The traditional ACID transaction model usually prevents such dependencies from happening, while “advanced” transaction models allow these dependencies to happen temporarily.

A transaction management system must keep track of the ongoing operations and of dependencies that have been induced by the conflict. PJava uses a *customizable lock manager* for this purpose.

A lock manager detects conflict as follows. Objects are associated with locks. To perform an operation op_i on an object O , the lock protecting O must be acquired in a *locking mode* corresponding to op_i . The compatibility of locking modes (and thus of operations) is defined by a two dimensional compatibility table: one dimension corresponds to the mode of lock, the other corresponds to the mode requested. The entry of the compatibility table corresponding to the current state of the lock and the mode of the lock request determines whether there is a conflict. If the request does not conflict, the requester is added to the set of *owners* of the lock.

PJava’s customizable lock manager allows a lock request to specify, in addition to the locking mode requested, a set of *ignore-conflict* relationships. An

⁶These dependencies are categorized as *dependencies due to behavior* in [7].

ignore-conflict relationship is a way to specify that one lock request can ignore an incompatible owner of the lock when diagnosing a conflict with the requested lock. For instance, a lock request issued from a transaction T_1 , specifying a ignore-relationship with T_2 (we say that T_1 is *non-conflicting* with T_2) will ignore any conflict with T_2 when deciding whether the lock can be granted.

In PJava, ignore-conflict relationships are specified using a *labeled directed graph* where vertices are locking capabilities and edges are *ignore-conflict* relationships. Edges are directed and labeled as either *transitive* or not. We note $C_i \succ^t C_j$ a transitive edge directed from C_i to C_j and $C_i \succ^{\neg t} C_j$ a non transitive edge from C_i to C_j . By default, edges are *transitive*.

This labeling of edges restricts the set of predecessors a locking capability can ignore conflict with. We call this set, $Pred(C)$ for a capability C , the set of *effective predecessors* of C . Thus, given a graph of locking capabilities, a locking capability ignores conflicts with all its effective predecessors in that graph. For instance, given the graph of locking capabilities illustrated on figure 3, we have:

$$\begin{aligned} C_p \succ^t C_q \succ^{\neg t} C_r &\Rightarrow Pred(C_r) = \{C_q\} \\ C_p \succ^t C_q \succ^t C_s &\Rightarrow Pred(C_s) = \{C_p, C_q\} \end{aligned}$$

Hence, C_s can ignore conflicts with both C_p and C_q , while C_r can ignore conflicts only with C_q . More formally, the set of predecessors of a capability C is defined as:

$$Pred(C) = Pred_{\neg t}(C) \cup \left[\bigcup_{\forall C_i \in Pred_t(C)} (\{C_i\} \cup Pred(C_i)) \right]$$

where

$$\begin{aligned} Pred_{\neg t}(C) &= \{ C_i \mid \exists C_i \succ^{\neg t} C \} \\ Pred_t(C) &= \{ C_i \mid \exists C_i \succ^t C \} \end{aligned}$$

$Pred_{\neg t}(C)$ (respectively, $Pred_t(C)$) denotes the set of immediate predecessors that forbid (allow) transitivity;

We also define $Owner(l, M)$ as the set of locking capabilities which own lock l in mode M , and $I_{owner}(l, M)$ as the set of owners of lock l in a mode *Incompatible* with mode M . For instance, in the case of read/write locking mode⁷, we have:

⁷PJava considers only read/write locking modes because they are easy to detect transparently. However nothing precludes the use of arbitrary locking modes defined according to some semantic criteria with the mechanism just described.

$$I_{owner}(l, Read) = Owner(l, Write)$$

$$I_{owner}(l, Write) = Owner(l, Write) \cup Owner(l, Write)$$

Lastly, we define $NCW(C)$ as the set of capabilities which are *Non-Conflicting With C*:

$$NCW(C) = \{C\} \cup Pred(C)$$

With these definitions, a request for a lock l in mode M is granted to a locking capability C if:

$$I_{owner}(l, M) \subseteq NCW(C)$$

As already mentioned, ignored conflicts create dependencies. More specifically, a dependency is created for each capability C_i such that $C_i \in (I_{owner}(l, M) \cap Pred(C))$. PJava keeps track of these dependencies and leaves to the user the interpretation and the elimination of these dependencies. Locking capabilities can be queried about their dependencies at any time. An exception is raised when one tries to release the locks of a locking capability that depends on at least one other capability, because it may lead to an inconsistent behavior. The intention is to force programmers to explicitly eliminate the dependencies using one of the available means.

There are three ways to eliminate dependencies: abort the transaction that owns the dependent capability, wait for a specific commit order before releasing the lock, or transfer the responsibility of the locks, and thereby, the visibility of the state of the objects these locks protect, to one of the transactions the dependency comes from. The latter is called *delegation* of locks.

5.2.2 Delegation

Delegation of locks allows one locking capability to *atomically* transfer the responsibility for its locks to another capability. Transferring lock responsibility means changing the ownership of the delegated locks, and thus transferring the control over the visibility of the objects the delegated locks protect. It also means transferring the dependencies that have been created for acquiring these locks. For instance, if a locking capability C_1 delegates its exclusive lock on an object O to a capability C_2 , C_1 is no longer able to access O after the delegation, until C_2 releases O 's lock or delegates it back to C_1 . Moreover, if C_1 acquired the lock on O by ignoring a conflict with a capability C_3 , C_1 's dependency on C_3 for O is transferred as well such that, after delegation, C_2 depends on C_3 .

We speak of *global delegation* when a capability transfers the responsibility for all its locks at once,

and *partial delegation* when it transfers the responsibility for only a subset of its locks. The class **LockingCapability** provides both forms of delegation. The method for global delegation takes just one parameter: the delegatee **LockingCapability**. A partial delegation takes an additional parameter to enumerate the objects whose locks must be delegated.

Global delegation is suitable for transaction models with well-defined development, that is, where the set of objects whose visibility will be delegated at the end of the transaction is known in advance. This is the case for the nested transaction model [19] and the coloured action model [22]. Partial delegation is required for supporting dynamic restructuring of transactions [16], a facility necessary in open-ended activity where developments are unpredictable and the set of objects that must be delegated is known only at the time when the need for restructuring the transaction occurs.

5.2.3 Notifications

A transaction model programmer can specify the sending of notifications when its customized conflict detection mechanism diagnoses a conflict. Every locking capability can specify one *conflict notification handler*, that will be invoked when a conflict is detected on a lock owned by the capability. Any object that implements the **ConflictHandler** interface can be used as a handler. This interface specifies a method that takes five parameters: the capability the handler is bound to, the mode in which it holds the lock, the object the lock protects, the capability that requested a conflicting lock, and the mode of the requested lock.

A **ConflictHandler** is typically used to mediate with end-users as part of the conflict resolution algorithm. Used together with the **LockingCapability**'s methods for restructuring the visibility of transactions, it allows the support of powerful multi-user collaborative environments.

Locking capabilities that are supplied with non-null **ConflictHandler** are given a thread to handle notifications. The enclosing **TransactionShell** of the thread is the owner of the notified capability. The thread is bound by default to the notified capability, but it may be bound to any other capability owned by its enclosing **TransactionShell**.

6 Summary

A design for adding extensible transaction management features to PJava has been presented. It aug-

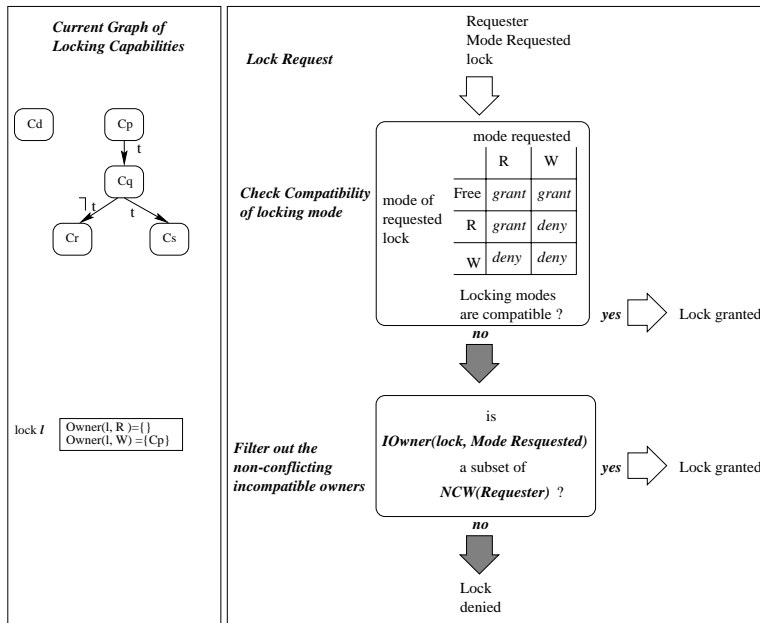


Figure 3: Customization of the lock manager's behavior using a graph of locking capabilities.

ments PJava with an extensible pool of transaction models and gives expert programmers the ability to extend this pool to accommodate the needs of new applications. Ordinary application programmers can then select at development time the transaction model best suited to their needs.

The support for extensible transaction management does not change the definition of the language Java, and does not require any discrimination of the data types and methods that can be used to build transactions. This allows programmers to write transactional persistent applications using any available Java classes, and without changing either the source or the compiled form of these classes.

The design presented in this paper is still a preliminary step toward a more comprehensive set of extensible transaction management features. We are currently investigating an alternative design that will rely more extensively on the usage of Java **interfaces** for defining the behavior of transaction classes, rather than by specialization of the abstract class **TransactionShell**. We will also work on extensions related to distribution and inter-operability.

The design also lacks of flexibility with respect to granularity issues. At the moment, it is assumed that the transaction properties are enforced at the object granularity and for all data manipulations. However,

the granularity of objects in Java is too small to realize transaction properties efficiently. This is especially true for concurrency control. The problem is to find a way to enable programmers to express larger granularities without changing the language definition. That is, the definition of object granules should remain orthogonal to the description of data types. Furthermore, the definition of these granularities must be logical, dynamic and persistent. For instance, one may want a list of objects to be considered as a single granule from the concurrency control point of view, no matter how many objects belong to the list or the number of insertions or deletions that occur on the list. The definition of such object granules would be useful for expressing locking granularity as well as for defining the objects that can escape from transaction control.

Our immediate concern is the implementation of our design. The addition of extensible transaction management as described in this paper requires further modifications to the current PJava virtual machine. The coupling with the CTPE's components requires the following changes:

- The machine-dependent implementation of Java threads must be changed to accept transactional attributes (e.g., enclosing **TransactionShell**, bound **UpdateBookKeeper** and **Locking-**

Capability) and to notify `TransactionShells` of the creation and termination of threads.

- The interpreter and the native methods must be augmented with mechanisms to trigger the automatic acquisition of locks and per update book-keeper generation of recovery information.
- The object cache as well as the garbage collected heap need to be extended with the ability to undo updates.

The implementation of the CTPE's components themselves will rely as much as possible on existing technologies. Implementation techniques for the lock manager components and the related building blocks have been discussed in [9] and will rely on previous experience in a similar context [10, 8]. The technology for implementing the recovery manager components will rely on recent extensions to the ARIES recovery method to support advanced features such as delegation of updates [18, 17]. Detail-led specifications are expected by the end of 1996. An initial prototype with support for flat transactions only will be implemented first based on these specifications. Support for extensible transaction management will be added gradually to this initial prototype during 1997.

References

- [1] M.P. Atkinson, L. Daynès, M.J. Jordan, T. Printezis, and S. Spence. "An Orthogonally Persistent JavaTM". *SIGMOD RECORD*, 25(4), December 1996.
- [2] M.P. Atkinson, L. Daynès, and S. Spence. "PJava Design 1.2". Working Document available via <http://www.dcs.gla.ac.uk/susan/pjava>, March 1996.
- [3] M.P. Atkinson, M.J. Jordan, L. Daynès, and S. Spence. "Design issues for persistent Java: A type-safe, object-oriented, orthogonally persistent system", May 1996. In the pre-proceedings of the 7th International Workshop on Persistent Object System (POS 7).
- [4] M.P. Atkinson and R. Morrison. "Orthogonal Persistent Object Systems". *VLDB Journal*, 4(3), 1995.
- [5] N.S. Barghouti and G.E. Kaiser. "Concurrency Control in Advanced Database Applications". *ACM Computing Surveys*, 23(3):269–317, September 1991.
- [6] E. Bertino, S. Jajodia, and L. Kerschberg, editors. *International Workshop on Advanced Transaction Models and Architectures (ATMA)*, Goa, India, September 1996. In conjunction with VLDB '96.
- [7] P.K. Chrysanthis and K. Ramamritham. "Synthesis of Extended Transaction Model using ACTA". *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.
- [8] L. Daynès. "Conception et réalisation de mécanismes flexibles de verrouillage adaptés aux SGBDO client-serveur". PhD thesis, Université Pierre et Marie Curie (Paris VI – Jussieu), 1995.
- [9] L. Daynès, M.P. Atkinson, and P. Valduriez. "Efficient Support for Customizing Concurrency Control in Persistent Java". In Bertino et al. [6], pages 216–233. In conjunction with VLDB '96.
- [10] L. Daynès, O. Gruber, and P. Valduriez. "Locking in OODBMS clients supporting Nested Transactions". In *Proc. of the 11th Int. Conf. on Data Engineering*, pages 316–323, Taipei, Taiwan, March 1995.
- [11] A.K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Data Management Systems. Morgan-Kaufman, San Mateo, CA, 1992.
- [12] M.F. Fernandez and S. Zdonik. "Transaction Groups: A Model for Controlling Cooperative Transactions". In Rosenberg and Koch [21], pages 341–350.
- [13] A. Garthwaite and S. Nettles. "Transaction for Java". Technical Report MS-CIS-96-17, School of Engineering and Applied Science, Computer and Information Science Department, University of Pennsylvania, Philadelphia, June 1996.
- [14] T. Härder and K. Rothermel. "Concurrency Control Issues in Nested Transactions". *VLDB Journal*, 2(1):39–74, 1993.
- [15] JavaSoft, a Sun Microsystems, Inc. Business. "JavaTM Core Reflection – API and Specification", October 1996.
- [16] G.E. Kaiser and C. Pu. "Dynamic Restructuring of Transactions". In Elmagarmid [11], chapter 8, pages 266–295.
- [17] C. P. Martin and K. Ramamritham. "ARIES/RH: Robust Support for Delegation by Rewriting History". Technical Report 95-51, University of Massachusetts, Amherst, Massachusetts, June 1995.
- [18] C. Mohan, D. Haderle, B. Lindsay, H. Pirashesh, and P. Schwarz. "ARIES : A Transaction Recovery Method supporting Fine-granularity Locking and Partial Rollbacks using Write-Ahead Logging". *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.
- [19] J.E.B. Moss. "Nested Transactions : An Approach to Reliable Distributed Computing". PhD thesis, Massachusetts Institute of Technology, April 1981.
- [20] C. Pu, G.E. Kaiser, and N. Hutchinson. "Split-Transactions for Open-Ended Activities". In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 26–37, Los Angeles, California, August 1988.

- [21] J. Rosenberg and D. Koch, editors. *Persistent Object Stores (Proc. of the Third Int. Workshop on Persistent Object Systems)*, Workshops in Computing, Newcastle, New South Wales, Australia, January 1989. Springer-Verlag in collaboration with the British Computer Society.
- [22] S.K. Shrivastava and S.M. Wheeler. "Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-coloured Actions". In *Proc. of the Int. Conf. on Distributed Computing Systems*, pages 203–210, Paris, France, May 1990.

Appendix: An Example of application programming with transactions

This appendix describes a small example of an application program with transaction semantics. It illustrates the various concepts underlying our design for introducing extensible transaction management into PJava.

The example is made of several Java classes. The classes in Figure 4 and Figure 5 are ordinary Java classes that implement the logic of the application.

```
/**
 * @class AutoTellerRequest
 * Formatted ATM request.
 * Exportable to non PJava virtual machine.
 * Assume a BankAccount class that defines
 * methods such as "deposit" and "withdraw",
 * which accept a single long for parameter.
 */
public class AutoTellerRequest {
    BankAccount ba;
    String op;
    long amount;
}
```

Figure 4: An ATM request.

The Figure 6 shows how the core of the application is isolated from the PJava external API. All transaction operations are mediated via an interface that encapsulates the transaction management primitives needed by the application. It shows also how using normal Java's features, the application can select at runtime the transaction model of its transactions.

The class `MethodInvocation` (Figure 7) shows how one can reduce the proliferation of `Runnable` classes by defining a single generic class that wraps arbitrary method invocation in a `Runnable` object. This

```
/**
 * @class AutoTellerBackend
 * A backend server for an ATM.
 * Exportable to non PJava virtual machine.
 */
public class AutoTellerBackend {
    private TransactionProcessor _tp;

    // Configuring the Backend server with a tp service
    public AutoTellerBackend(TransactionProcessor tp) {
        _tp = tp;
    }

    private AutoTellerRequest nextRequest(){
        // whatever...
    }

    public void serverLoop(){
        AutoTellerRequest rq;
        Object [] rq_args = new Object[1];

        while ( (request = nextRequest()) != null ) {
            rq_args[0] = new Long(rq.amount);
            _tp.runTransaction(new
                MethodInvocation(ba,rq.op,rq_args));
        }
    }
}
```

Figure 5: An ATM backend server.

`MethodInvocation` class requires the reflexive functionalities of JDK 1.1, described in JavaSoft's draft of the Core Reflection API [15].

The compiled form of all the classes and interfaces shown in this example, except for the class `GenericTransactionProcessor` which uses directly the external API for its implementation, may be exported to and executed "as is" by a standard Java virtual machine. In order to be able to execute the class `AutoTellerBackend` on a non PJava virtual machine, one just needs to supply it with an implementation of the `TransactionProcessor` interface.

```

/**
 * @interface TransactionProcessor
 * Interface of objects with the ability to
 * execute transaction.
 * Exportable to non PJava virtual machine.
 * This interface isolates the application code from
 * the transaction-aware layer.
 */
interface TransactionProcessor {
    boolean runTransaction(Runnable body);
    void begin();
    void end();
    void abort();
    void participate(Runnable participantBody);
}

```

Figure 6: How the core of the application is isolated from the transaction-aware part.

```

/**
 * @class MethodInvocation
 * Requires the Java Core Reflection API from JDK 1.1
 * Exportable to non PJava virtual machine.
 * Simple implementation of method invocations as
 * first-class objects. Basis for composing easily
 * arbitrary method invocations with TransactionShell
 * or Thread objects.
 * Example of composition:
 * <pre>
 * class A {
 *     public ma(B b, int i){...}
 *     ...
 * }
 * A a = new A();
 * Object [ ] invocationArgument = new Object[2];
 * invocationArgs[0] = new B();
 * invocationArgs[1] = new Integer(19);
 * MethodInvocation mo =
 *     new MethodInvocation(a, "ma", invocationArgs)
 * new Thread(mo).start();
 * new FlatTransaction().start(mo);
 * </pre>
 * Primitive types must be wrapped by an equivalent class
 * object (e.g., int must be wrapped in a java.lang.Integer).
 *
 * @see java.lang.Class
 * @see java.lang.reflect.Method
 * @see java.lang.Runnable
 */
public class MethodInvocation implements Runnable
{
    private Object _target;
    private java.lang.reflect.Method _method;
    private Object [ ] _args;

    public MethodInvocation(String method_name,
                           Object target,
                           Object [ ] args )
        throws NoSuchMethodException,
               SecurityException
    {
        Class [ ] paramsType = null;
        if (args != null) {
            paramsType = new Class[args.length];
            for ( int i = 0; i < args.length; i++)
                paramsType[i] = args.getClass();
        }
        Class c = target.getClass();
        _method = c.getMethod(method_name,
                              paramsType);

        _target = target;
        _args = args;
    }

    public void run(){
        Object result = _method.invoke(_target, _args);
    }
}

```

Figure 7: Generic method invocation objects.

```

/**
 * @Class GenericTransactionProcessor
 * The only class using directly the external API.
 * Cannot be exported to non PJava virtual machine
 */
import
pjava.transactions.building_blocks.TransactionShell;

class GenericTransactionProcessor implements
TransactionProcessor {
    Class transactionModel;

    public GenericTransactionProcessor( String model )
        throws ClassNotFoundException
    {
        transactionModel = Class.forName(model);
    }
    boolean runTransaction(Runnable body) {
        TransactionShell t =
transactionModel.newInstance();
        t.setBody(body);
        t.start();
        return ( t.claim() == TransactionShell.SUCCESS );
    }
    void begin() {
        t.start();
    }
    void end() {
        t.end();
    }
    void abort(){
        t.kill();
    }
    void participate(Runnable participantBody){
        t.enter(participantBody);
    }
}

```

Figure 8: The only transaction-aware class of the application.