PDC: Persistent Data Collections pattern

Tiago Massoni Vander Alves Sérgio Soares Paulo Borba*
Centro de Informática
Universidade Federal de Pernambuco

Introduction

The object–oriented applications layer architecture [2, 3] allows the distribution of classes into well–defined layers, according to each crosscutting concern of an application (business, communication, data access, etc.) to obtain separation of concerns. Elements from different layers communicate only through interfaces. However, we have to refine these layers by filling them with specific classes. The complete set of these classes, related to business and data access concerns, was transformed into a design pattern, called PDC (Persistent Data Collections), which is presented in this paper.

Brief

Provide a set of classes and interfaces in order to separate data access code from business and user–interface code, promoting modularity.

Context

When developing persistent object–oriented information systems applications using specific Application Programming Interfaces (APIs) that lead to interwoven code making maintenance and reuse difficult.

Problem

Obtain better maintenance and reuse levels when using persistence mechanisms to develop an object–oriented application.

Forces

- Developers should be able to address the business aspects of an application independently from persistence operations.

*Supported in part by CNPq, grant 521994/96–9. Electronic mail: {tmasson,vralves}@us.ibm.com, {scbs,phmb}@cin.ufpe.br. Av. Professor Luis Freire s/n Cidade Universitária 50740–540 Recife PE Brazil.

- *Ad hoc* implementations directly using specific Application Programming Interfaces (APIs) usually lead to interwoven code that is hard to maintain. For example, a Java [8] program can use the JDBC API (Java Database Connectivity API [14]) for manipulating persistent data within business code.
- The type of persistent storage or vendor may change over the life of an application.
- Business classes may be reused by other applications.
- It may be non-trivial to deal with some aspects from persistent systems, such as enabling connections to database platforms and managing transactions efficiently.
- The system performance should not be affected.

Solution

The basic idea of PDC is to avoid mixing data access code with business code from domain-related objects, leading to extensibility and reusability. For this purpose, we propose the separation of design classes in two types:

- classes describing business logic objects.
- classes for data manipulation and storage, with specific persistence code.

The communication between these two types of classes is carried out through interfaces, which guarantee independence between the business layer and the data access layer. Business code will be the same, regardless of how data access operations are implemented.

PDC suggests the use of persistent data collections, which contain code for manipulating a group of persistent objects of an application. These collections represent a clear distinction between the "data" and the "data set", being the core of our solution. Our solution is complemented by ideas taken from other well-defined design patterns, as Facade, Abstract Factory and Bridge [7]. The goal is to reduce the impact caused by modifications in the system functional and non–functional requirements.

As in the example in Figure 3, for each important domain object which will be persistent in the application (like `Account`), we create two other classes: the business collection (`AccountRecord`) and the data collection (`AccountRepositoryJDBC`) classes, representing business and persistent collections of domain objects, respectively. Furthermore, each persistent domain class must inherit from the `PersistentObject` class, indicating that its objects will be stored persistently.

The `Bank` class encapsulates all services offered by the application (applying the Facade pattern [7]). The object from this class calls methods on all business collection objects of the application (as `AccountRecord`), in order to implement the services. The business collection in turn uses persistence-related services from its corresponding persistent data collection (as the `insert` and `search` methods).

The `PersistenceMechanismJDBC` class is used by `Bank` and persistent data collections (as `AccountRepositoryJDBC`) for performing database platform services, such as connection and transaction management. These issues are addressed by specific methods in the persistence mechanism.

In order to request services from the data access layer, the business objects send messages to data access objects only through interfaces, which provides extensibility for the design of the application. In the example, the `IAccountRepository` interface separates business collections from persistent data collections, and the `IPersistenceMechanism` interface isolates specific persistence mechanism services from its business clients, such as the `Bank` class.

As in the example above, we can use PDC to structure the application using a set of specific classes, separating business and user–interface concerns from persistence concerns. Such application is easier to maintain and to extend, since its core functionality is decoupled from data access code. In addition, classes from the application can also be reused by other applications.

Structure

Figure 1 details the structure of PDC, using an UML class diagram [4]. The class names denote the element of the pattern itself, including classes with the "Interface" stereotype, which denote interfaces containing only method signatures to be implemented by the indicated classes.

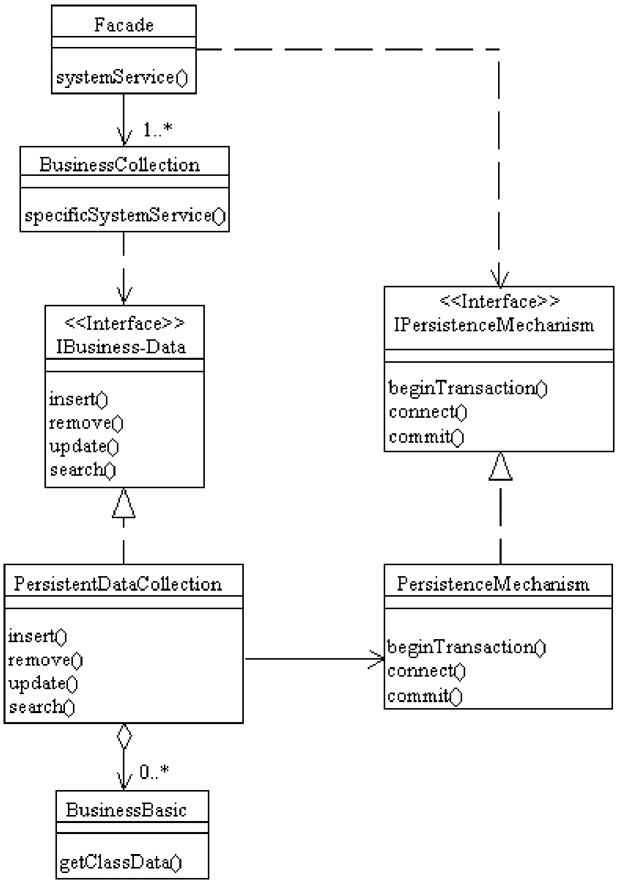

Figure 1: Class diagram of PDC.

The participants of the pattern are presented as follows, along with their matching elements of the example presented in Figure 3:

- **Facade.** This class provides a simple interface to all services of a complex system [7]. A facade offers a simple default view of the system that is useful for most clients. It keeps references to the several `BusinessCollection` objects of the application, and delegates calls to them. Additionally, it implements the Singleton pattern, thus exactly one instance of this class will be active during execution. This element is represented by the `Bank` class in the example.

- **BusinessBasic.** This class represents a business basic concept, reflecting clearly the problem domain (for instance, account, client, investment). If we choose this class to inherit from an abstract class containing abstract data access methods (see Implementation Section), the `BusinessBasic` class has to implement those methods. Using this approach, although some data access code is placed within a business class, the business code of the class does not depend on the data access code. Such code on a business basic class can be easily removed or replaced, with no impact on business code. In the example, this class is represented by the `Account` class.

- **BusinessCollection.** This class represents a grouping of objects from a significant business basic class, on the business' perspective. It contains methods for inserting, querying, updating, and deleting business objects, with verification and tests of preconditions related to the object manipulation. Furthermore, the `BusinessCollection` class also contains methods directly related to the application domain. This element is represented by the `AccountRecord` class in the example.

- **PersistentDataCollection.** This class contains methods for manipulating persistent objects of a specific business basic class. The code for these methods depends on a specific API for accessing some persistence platform, thus any changes to this platform will cause direct impact on this class, but absolutely no impact on business code (since the `IBusiness–Data` interface isolates these changes). The `PersistentDataCollection` class implements methods from a `IBusiness–Data` interface and depends on services from the `PersistenceMechanism` class in order to perform database operations, more specifically for finer granular transactions and database connections. In the example, the role of this class is played by the `AccountRepositoryJDBC` class.

- **IBusiness–Data.** This interface establishes a communication protocol between `BusinessCollection` objects and `PersistentDataCollection` objects. A business collection class depends on this interface for storing and retrieving objects from the database. This approach promotes modularity, since changes to the data access code do not have impact on business code. In the example, this interface is represented by `IAccountRepository`.

- **PersistenceMechanism.** This class contains methods that implement specific services related to a database platform, such as connecting to and disconnecting from the database, and transaction management. Methods related to connection management open and maintain a database connection for a service from the application, making this connection available to one or more `PersistentDataCollection` objects involved in the accomplishment of the service. Methods related to transaction management open, confirm or abort transactions, in order to provide consistency among all operations used to accomplish an application service. The code of these methods depends on a specific persistence API. This class is represented by the `PersistenceMechanismJDBC` class in the example.

- **IPersistenceMechanism.** This interface is defined in order to provide independence between the business classes and the `PersistenceMechanism` class (which implements this interface). Therefore, if we change the database platform, we have to replace the old `PersistenceMechanism` object by a new object, but this modification does not have impact on business classes. The `Facade` class depends on this interface for invoking transaction methods. The example presents an interface with the same name.

Dynamics

Figure 2 shows a sequence diagram [4] of a typical scenario for the use of PDC, using the approach of data access methods encapsulated into a business basic class (see Implementation Section). The `Facade` object creates a `PersistenceMechanism` object, whose services will be requested during execution. Next, a service on the `Facade` object is called, which in turn begins a transaction (invoking a method on the `PersistenceMechanism` object) and delegates the call to a `BusinessCollection` object in order to perform this service (a querying operation that retrieves data from the database). The `BusinessCollection` object performs all validation and tests on the input data, then invokes an operation to manipulate persistent data on the corresponding `PersistentDataCollection` object (through the corresponding business–data interface). The latter creates an empty `BusinessBasic` instance and fills it with database information (calling `deepAccess`, which in turn executes queries through services offered by the `PersistenceMechanism` object, as the `executeQuery` method), returning the resulting object to the `Facade` object. In the end of the operation, the `Facade` object confirms the end of a database transaction, invoking `commitTransaction` on the `PersistenceMechanism` object.

Consequences

The use of PDC offers the following benefits:

- *Support for independent implementation.* PDC's layer architecture allows to address the business aspects independently from persistence operations. This abstraction is promoted by interfaces between the business layer and the data access layer.

- *Maintainability.* The pattern's structure increases the system maintainability by separating business code from data access code. Therefore, changes in the data access classes should not interfere in the business classes.

- *Extensibility.* The pattern makes it easier to seamlessly change the database technology or vendor, minimizing or even eliminating impact on business code. Interfaces between the business layer and the data access layer promote the desired extensibility for the application.

- *Use of several persistence platforms.* The resulting code is able to support storing objects into several persistence platforms, such as files, relational and object–oriented databases, by creating a number of implementations for the persistence

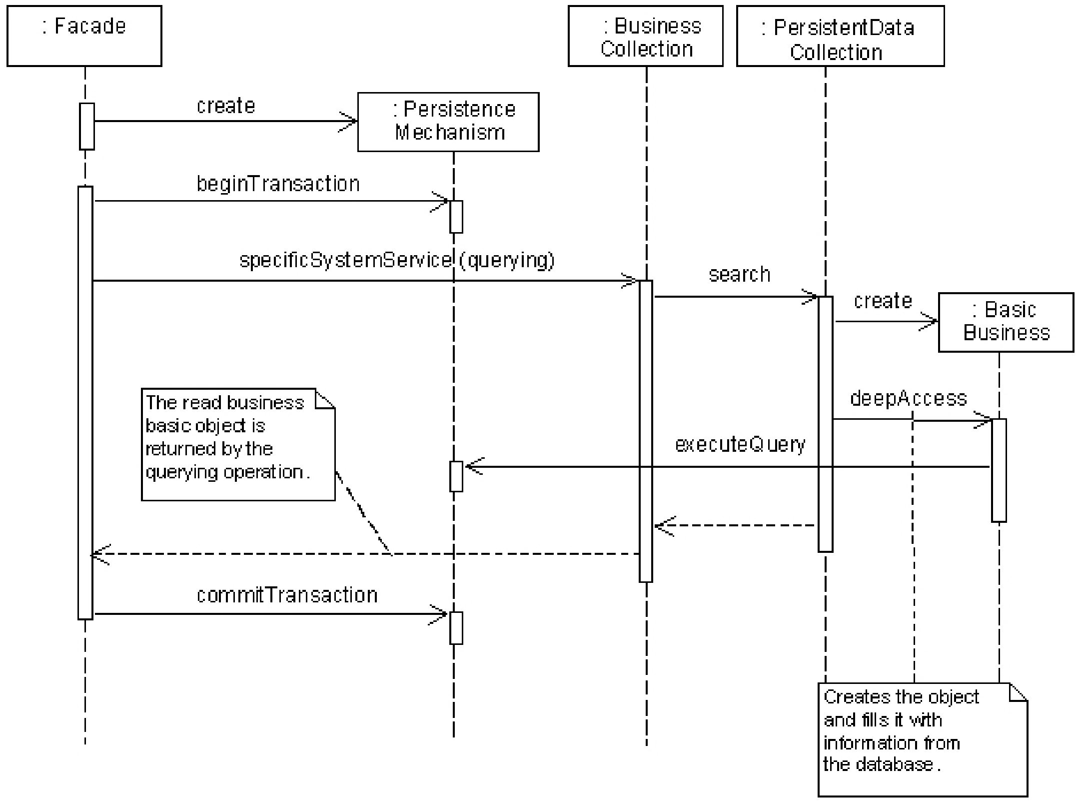

Figure 2: Dynamics of PDC.

mechanism class and for each persistent data collection class; all of these classes must implement the corresponding interfaces.

- *Reuse.* Due to the structure provided by the pattern, business classes can be easily reused by another application based on other database technologies. In addition, changes to data access issues are simpler, since they are restricted to data access code.

- *Abstraction.* As the pattern abstracts the persistence problem by using interfaces, persistence implementation may use complex algorithms or APIs to deal with some non-trivial aspects from persistent systems, such as enabling connections to database platforms and managing transactions efficiently.

- *Support for progressive implementation.* During early phases of the application development, functionally complete prototypes are constructed, where business collection classes depend on business–data interfaces, but the latter are implemented by volatile data collections (storing objects in memory only). Later, data access code can be added seamlessly, replacing volatile data collections by specific persistent data collection objects, then adding a persistence mechanism object. Such approach enables addressing the business problems independently from persistence operations, simpler validation of user requirements, and simplification of tests [9].

The liabilities of the pattern are:

- *Increased number of classes.* For each significant business basic class, we have to create up to three additional classes and one interface. However, their structure is simple and their generation can be simply automated by tools.

- *Increased indirection.* In order to introduce the layer architecture we must use different kinds of classes that delegate some calls to others, which may decrease system performance. In fact, this lost of efficiency is minimal, since these indirections are locally executed, and the additional execution time is irrelevant when compared to the overhead of the IO operations that read from and write to the persistence mechanism.

Implementation

Here we consider how to implement PDC using JDBC as the data access API for using relational database services. Consider the following implementation issues:

- *Java platform.* The pattern elements must be implemented in the Java programming language, since JDBC is part of the Java platform.

- *Inheritance in the business basic class.* Most code for manipulating objects using JDBC can be contained in business basic classes, within methods inherited from an abstract class (`PersistentObject` in our banking example). It can be considered a miscellaneous of business and data access code, even though those inherited data access methods are not invoked by business code (as mentioned earlier). One alternative for such situation is to transfer all code for manipulating persistent business basic objects to the persistent data collection classes. The disadvantage of such approach is that changes in a business basic class will also reflect in the corresponding persistent data collection class; it is necessary to implement a new persistent data collection for each new platform. On the other hand, in this approach changes in the persistent platform will not affect the business basic classes.

- *Transactions.* Using JDBC, we can easily implement transactions using database services. We must use the `setAutoCommit`, `commit` and `rollback` methods on the `Connection` class in order to implement a transaction when implementing a sequence of operations, which must be executed as a single one.

- *Business basic subclasses.* A business basic class can be specialized in business basic subclasses, depending on the business rules. In the case of business collection and persistent data collection classes (including business–data interfaces), we can choose from two design alternatives: one is to create a class for each business basic subclass; another is to use only one class, in order to avoid duplicate code. A detailed discussion about this topic is presented in a related work [15].

- *Concurrency control.* One concurrency problem arises when using a connection pool to manage the connections with the persistence mechanism. Each execution flow (thread) must obtain a connection from the connection pool before communicating with the persistence mechanism. Usually there is a single connection pool containing all the connections of the system, and thus this poll is accessed concurrently. Moreover, we need to apply some concurrency control to the system. Examples of others situations in which concurrency control should be addressed are interference by business rules (system policies), unsafe data types, and other race conditions [12].

- *Volatile data collections.* We can use this type of class for storing objects in a non–persistent manner, in order to support progressive implementation. Using this approach, we can abstract from persistence or any other non–functional requirement, when implementing functional prototypes for the application. These prototypes can be useful for validating user requirements and simplifying tests. This class also implements its corresponding business–data interface, but its methods use in–memory data structures like arrays or lists to manipulate business objects.

- *Abstract factories.* Variations of PDC can include classes which represent abstract factories [7], in order to increase extensibility and reusability of business classes. An abstract persistence factory class can be introduced, containing a method for creating a persistence mechanism object, and such method can be implemented by a subclass of the abstract factory, the concrete factory. The facade object can call this method to instantiate the persistence mechanism, without making a explicit call to its constructor method. The same idea can be used for creating persistent data collections, isolating the business classes (facade and business collection classes) from the instantiation code. In both cases, the information needed by the concrete factories to instantiate the objects is placed in simple text or XML configuration files.

Sample Code

We now provide a brief sketch of the implementation of the main elements of PDC using Java and the JDBC API, in the banking application example introduced in Figure 3. First, we present a business basic class, `Account`, which reflects directly the problem domain. The `public` modifier in classes and methods is omitted by brevity.

```
class Account extends PersistentObject {
    private Number number;
    private double balance;
    void credit(double value) {  balance = balance + value;  }
    ...
    /* Data access operations */
    void insert() throws StoringException {
        try {
            String sql = "insert into account values (";
            sql += "ID = "+super.getId();  // get the object id
            sql += "NUMBER = "+this.getNumber();
            sql += "BALANCE = "+this.getBalance();
            super.pm.executeUpdate(sql);
        } catch (SQLException e) { throw new StoringException(); }
    }
```

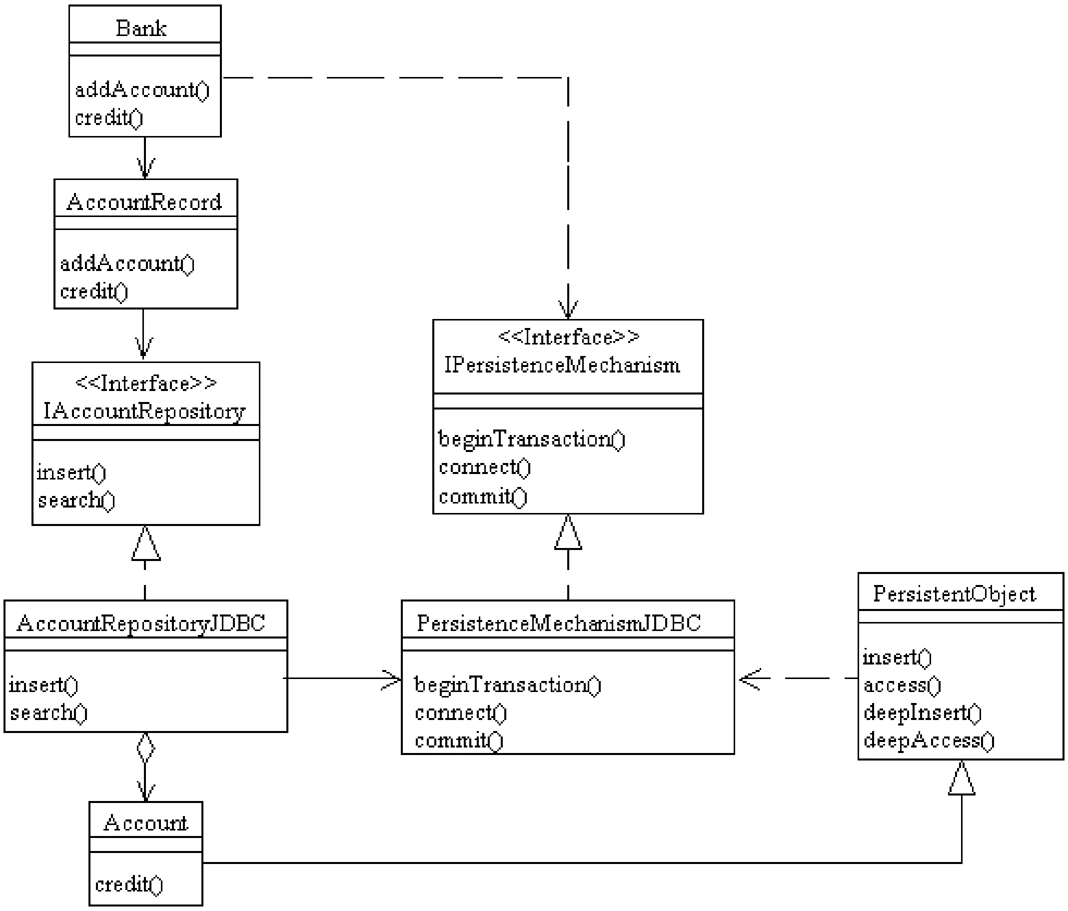

Figure 3: Example of PDC applied to a banking application.

Two of the attributes and one business operation, `credit` (containing only business code and not invoking any data access method), are presented above. In another portion of the class, there are data access methods inherited from the `PersistentObject` class, containing specific code for database operations in this class (as the `insert` method). Any exception related to the data access API (`SQLException`) is replaced by a general database exception (`StoringException`).

In addition, this class contains methods with the `deep` prefix, which are special operations for manipulating attributes which are references to other objects or collection of objects (as the `number` attribute). The `deepInsert` method in the `Account` class has an `IPersistenceMechanism` interface parameter receiving a reference to a persistence mechanism object in order to perform the corresponding database operation:

```
    void deepInsert (IPersistenceMechanism pm)
            throws StoringException {
        super.pm = pm;
        this.number.deepInsert(pm);
        this.insert();
    }
    ...
}
```

Notice that deepInsert is called first for the attribute, before the insert for the Account object. This order is followed in operations to write data to the database, due to a restriction of relational databases, which forces the code to insert rows in auxiliary tables first (number attribute), then insert a row in the main table (Account object). In this way, the relationships can be established with no errors. This order does not need to be followed in operations querying the database. Operations deleting data from the database depend on the referential integrity defined for the tables involved.

Although there is business code along with data access code in the same class, the business methods do not depend on the data access methods, since the former do not invoke the latter. Therefore, we can insert and remove data access methods with no impact on business code (a process easily automated by tools). The PersistentObject class is presented below:

```
abstract class PersistentObject {
    protected long id;
    protected IPersistenceMechanism pm;
    abstract void insert() throws StoringException;
    abstract void deepInsert(IPersistenceMechanism pm)
        throws StoringException;
    abstract void access() throws StoringException;
    abstract void deepAccess(IPersistenceMechanism pm)
        throws  StoringException;
    ...
}
```

where the id and the pm attributes denote the object identity of a persistent object and a persistence mechanism object to perform database operations, respectively. The abstract data access methods in this class must be implemented by all business basic classes, which will be made persistent. The StoringException exception is raised when a problem occurs in any database operation.

In order to represent a set of business basic objects on the business' vision, we use a business collection class. We present the class AccountRecord, which represents a set of bank accounts:

```
class AccountRecord {
   private IAccountRepository accountsRep;
   AccountRecord(IAccountRepository accountsRep) {
       this.accountsRep = accountsRep;
   }
```

where the constructor of AccountRecord receives as argument an object which implements a business–data interface, and two of the business operations for this class, addAccount and credit, are also presented. The first method inserts an Account object into the database, raising an exception if an account with the same number already exists.

```
    void addAccount(Account account)
            throws StoringException, DuplicateAccountException {
        if (this.accountsRep.exists(account.getAccountNumber()))
            throw new DuplicateAccountException();
        else  this.accountsRep.insert(account);
    }
```

The second method queries the database for a given account. If the query is successful, a value is added to the account's balance and the account is updated in the database. However, if the account does not exist in the database, an exception is raised.

```
    void credit(Number accountNumber, double value)
            throws StoringException, UnknownAccountException {
        if (accountsRep.exists(accountNumber)) {
            Account account = accountsRep.search(accountNumber);
            account.credit(value);
            this.accountsRep.update(account);
        }
        else  throw new UnknownAccountException();
    }
    ...
}
```

The database is represented by the attribute `accountsRep`, a business–data interface with data access operations. This interface is as follows:

```
interface IAccountRepository {
    void insert(Account account) throws StoringException;
    Account search(Number accountNumber) throws StoringException;
    void update(Account account) throws StoringException;
    boolean exists(Number accountNumber) throws StoringException;
    ...
}
```

where the `update` method is important to maintain consistency between in–memory (volatile) and persistent objects. Other methods on this interface could be complex queries (for instance, returning a set of objects) and methods for sequential querying.

A class implementing a business–data interface is a persistent data collection class. In our example, this class implements its methods invoking data access methods defined in the business basic classes. In our example, the `AccountRepositoryJDBC` class is presented as follows:

```
class AccountRepositoryJDBC implements IAccountRepository {
    private PersistenceMechanismJDBC pm;
    void insert(Account account) throws StoringException {
        account.deepInsert(this.pm);
    }
```

Note that the `pm` attribute stores a persistence mechanism object, which is passed as an argument for the database operations on `Account` objects, as in the `search` method.

```
    Account search(Number accountNumber) throws StoringException {
        Account ac = new Account(accountNumber);
        ac.deepAccess(this.pm);
        return ac;
    }
    ...
}
```

On the other hand, if it is desired to develop a functional prototype first, we can implement a business–data interface using a volatile data collection. In the banking application, we can create a class which stores and retrieves `Account` objects from an array. The objects will be maintained in the array only during the current execution.

The facade class of the pattern is represented by the `Bank` class in this application:

```
class Bank {
    private IPersistenceMechanism pm;
    private AccountRecord accounts;
    Bank() throws PersistenceMechanismException {
        PersistentFactory factory = PersistentFactory.getFactory();
        this.pm = factory.createPersistenceMechanism();
        this.accounts = new AccountRecord(
             AccountDataFactory.getFactory().createDataCollection(pm));
    }
    void addAccount(Account account)
            throws StoringException, AccountAlreadyExistsException {
        this.pm.beginTransaction();
        try {   this.accounts.add(account);  }
        catch (Exception e) {
            this.pm.cancelTransaction();
            throw e;
        }
        this.pm.commitTransaction();
    }
    void credit(String accountNumber, double value)
            throws StoringException, UnknownAccountException {
        this.pm.beginTransaction();
        try {  this.accounts.credit(accountNumber,value);  }
        ...
    }
    ...
}
```

This persistence mechanism object is instantiated in the `Bank`'s constructor, in order to initialize the system, being stored in an `IPersistenceMechanism` interface attribute. All the initialization process is performed using a `PersistenceFactory` class, which reads a configuration file and creates the right specific persistence factory object for the application. This object will then create the specific persistence mechanism object for the `Bank` class, promoting extensibility of the business code (the facade class does not instantiate the persistence mechanism object directly). See the Implementation section.

Bank uses services from its AccountRecord attribute, delegating calls to the latter in its methods. This attribute is initialized by passing as argument a new persistent data collection object, which implements a business–data interface and receives a persistence mechanism object. In order to maintain separation between business and data access code, this persistent data collection object is instantiated by a specific data factory for JDBC, which in turn was first instantiated by a static method (getFactory) in an abstract AccountDataFactory class (see Implementation section). In the addAccount and credit methods, the Facade class invokes methods on the persistence mechanism object for beginning and confirming a transaction, or canceling it if some exception occurs.

The IPersistenceMechanism interface, which is used by Bank, is presented as follows:

```
interface IPersistenceMechanism {
    void beginTransaction() throws PersistenceMechanismException;
    void commitTransaction() throws PersistenceMechanismException;
    void cancelTransaction() throws PersistenceMechanismException;
    void connect() throws PersistenceMechanismException;
    void disconnect() throws PersistenceMechanismException;
    ...
}
```

where PersistenceMechanismException is the exception raised when some error occurs in one of those operations. A persistence mechanism class implements this interface using specific database API operations, as in the following example:

```
class PersistenceMechanismJDBC implements IPersistenceMechanism {
    void beginTransaction() throws PersistenceMechanismException {
        try {
            // requests a connection from a connection pool
            Connection conn = this.requestConnection();
            conn.setAutoCommit(false);
        }
        catch (SQLException e) {
            throw new PersistenceMechanismException();
        }
    }
    ...
}
```

This class implements the beginTransaction method using services from the JDBC API. First, a connection to the database is requested from a connection pool (allowed by JDBC). If there is not any opened connection, a new one is created. Then a transaction is initialized in the context of the connection. Any SQLException raised is replaced by a general exception, in order to guarantee isolation between business and data access code.

Known Uses

Several organizations have been using PDC as a design pattern for many real software projects. Most of these projects have aimed at developing from simple to complex applications, and satisfactory results have been collected in such situations. Some of these systems are presented as follows:

- A system to manage clients of a telecommunication company. The system is able to register mobile telephones and manage client information and telephone services configuration. The system can be used over the Internet.
- A system for performing online exams. This system has been used to offer different kinds of exams, such as simulations based on previous university entry exams, helping students to evaluate their knowledge before the real exams.
- A complex supermarket system. A system that is responsible for the control of sales in a supermarket. This system will be used in several supermarkets and is already been used in other kinds of stores.
- A system for registering health system complaints. The system allows citizens to complaint about health problems and to retrieve information about the public health system, such as the location or the specialties of a health unit.
- This pattern is also used in undergraduate and graduate courses on object–oriented programming at the Center of Computer Science of the Federal University of Pernambuco. Several kinds of systems (such as games, academic control systems, and sales systems) have been developed in these courses.

In addition, the pattern is one of the basic patterns of the Progressive Implementation Method (Pim) [5]. Pim is a method for the systematic implementation of complex object–oriented applications in Java. In particular, this method supports a progressive approach for object–oriented implementation, where persistence, distribution and concurrency control are not initially considered in the implementation activities, but are gradually introduced, preserving the application's functional requirements [1, 9, 11, 15]. Pim relies on the use of specific architectural and design patterns for structuring object–oriented applications, in order to promote modularity and separation of concerns [10]. PDC is the design pattern applied for dealing with persistence.

Related Patterns

- *Crossing Chasms [6].* In their set of patterns for object–relational integration, Brown and Whitenack deal with the definition of database schemas for relational databases, supporting the object model. These patterns can be useful in PDC (for setting up the database tables), since they have distinct objectives (PDC aims at structuring the application in layers for a seamless introduction of persistence).
- *Persistent Layer and other patterns [16].* Yoder's patterns and PDC have very similar objectives in obtaining separation of concerns between business and data access code. Many of the ideas presented in the Yoder's patterns can be combined into elements of PDC in a practical way (for instance, Transaction Manager and Connection Manager can be instantiated as the PDC's persistence mechanism class). However, Yoder's patterns do not separate definitions of "data" and "data set", as defined in our persistent data collections, and assuming to be applied specifically

to relational databases. We believe that PDC can be applied almost directly to a number of persistence platforms, including object databases and files.

- *Abstract Factory [7].* This pattern is applied in PDC to implement a persistence factory class for creating persistence mechanism objects, which is used by a facade class. Factories also can be used for creating persistent data collection objects transparently for the business collection classes (see Implementation section).
- *Facade [7].* The facade class of PDC is a direct implementation of the Facade pattern.
- *Singleton [7].* Usually only one facade object is required in an application. Thus facade objects are often implemented as Singletons.
- *Bridge [7].* This pattern is used in PDC as the business–data and persistence mechanism interfaces, which play the role of a bridge between the business and the data access layers.
- *Concurrency Manager [13].* This pattern can be used in PDC to control concurrent situations, such as interferences by business rules (system policies), unsafe data types, and other race conditions.

Acknowledgements

We would like to give special thanks to our shepherd in this paper, Rosana Teresinha Vaccare Braga, from ICMCSC-USP, for making important suggestions for improving this pattern. We also thanks Jorge L. Ortega Arjona and Gunter Mussbacher for the suggestions made at the conference.

References

[1] Vander Alves. Progressive Development of Distributed Object-Oriented Programs. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, February 2001.

[2] Scott Ambler. *Building Object Applications that Work.* Cambridge University Press and Sigs Books, 1998.

[3] Scott Ambler. *The Object Primer.* Cambridge University Press, 2001.

[4] Grady Booch et al. *The Unified Modeling Language User Guide.* Object Technology. Addison-Wesley, 1999.

[5] Paulo Borba, Saulo Araújo, Hednilson Bezerra, Marconi Lima, and Sérgio Soares. Progressive implementation of distributed Java applications. In *Engineering Distributed Objects Workshop, ACM International Conference on Software Engineering*, pages 40–47, Los Angeles, USA, 17th–18th May 1999.

[6] K. Brown and B. Whitenack. Crossing Chasms: A Pattern Language for Object-RDBMS Integration. In J. Vlissides et. al. (eds.), Pattern Languages of Program Design 2. Addison-Wesley, 1996.

[7] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[8] James Gosgling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[9] Tiago Massoni. A Software Process with Support to Progressive Implementation (in portuguese). Master's thesis, CIn – Federal University of Pernambuco, February 2001.

[10] David L. Parnas et al. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of ACM*, 15(12):1053–1058, December 1972.

[11] Sérgio Soares. Progressive Development of Concurrent Object-Oriented Programs (in portuguese). Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, February 2001.

[12] Sérgio Soares and Paulo Borba. Concurrency Control with Java and Relational Databases (in portuguese). In *V Brazilian Symposium of Programming Languages*, 23th–25th May 2001.

[13] Sérgio Soares and Paulo Borba. Concurrency Manager. Technical report, State University of Rio de Janeiro—UERJ, Rio de Janeiro, Brazil, 3th–5th October 2001. To appear.

[14] Sun Microsystems. Java Database Conectivity Specification, 2000. Available at ftp://ftp.javasoft.com/pub/jdbc.

[15] Euricélia Viana. Integrating Java with Relational Databases (in portuguese). Master's thesis, Centro de Informática, UFPE, 2000.

[16] J.W. Yoder, R.E. Johnson, and Q.D. Wilson. Connecting Business Objects to Relational Databases. In *Proceedings of the 5th Conference on the Pattern Languages of Programs*, Monticello-IL-EUA, August 1998.