Persistent State Pattern

André V. Saúde*
Dept. Computer Science
Federal University of Lavras
Lavras, Brazil
saude@dcc.ufla.br

Ricardo A. S. S. Victório
Mitah Technologies
Lavras, Brazil
ricardo.victorio@mitahtech.com

Gabriel C. A. Coutinho
Mitah Technologies
Lavras, Brazil
gabriel@mitahtech.com

ABSTRACT

Finite State Machines (FSM) provide a powerful way to describe dynamic behavior of systems and components. Implementations of FSM in Object-Oriented (OO) languages have been widely studied since the classical *State* design pattern has been introduced. Various design patterns were derived from the *State* pattern, however, the focus was always on object's behavior. This paper proposes the Persistent State Pattern, an extension to the *State* pattern where persistent data is taken on account. The Persistent State Pattern integrates classical and enterprise design patterns, and it is conceived for enterprise applications. It can be used with OO databases and relational databases, and it can also be incorporated by an object-relation mapping framework. We present the close relationship between the Persistent State Pattern and the concepts of model and event driven design.

1. INTRODUCTION

Finite State Machines (FSM) provide a powerful way to describe dynamic behavior of systems and components. There are several available implementations of FSM in Object-Oriented (OO) languages. The OO implementations of FSM have been widely studied, and several design patterns have been proposed in the literature to deal with states in OO. The basic design pattern for states is the *State* design pattern, popularized by the most cited design pattern reference [10].

The State Pattern is a solution to the problem that an object's behavior is a function of its state, and it must change its behavior at runtime depending on that state. In short, it is a *behavioral* design pattern. The various design patterns derived from the *State* pattern also have the focus on object's behaviour [1, 2, 20, 21]. None of them deals with states in a persistence framework.

Persistent states exist when a database entity (of an Entity-Relationship Model (ERM) [6]), takes part of a business process or a workflow. Processes and workflows may be executed by various actors and must be able to be persistent. A database entity which is part of the process may be persisted in different states. Let us take as example a purchase process, in a company, which could be summarized by the following steps: i) a professional in the company's operation creates a purchase request, which is modeled by an entity called PurchaseRequest; ii) the purchase request is evaluated by the company's financial office before, and it can be approved or rejected; iii) the purchase office contacts suppliers and negociates prices; iv) the financial office pays; v) the product is delivered. During this process, the PurchaseRequest entity changes it's state several times. The PurchaseRequest entity could have, for instance, the states *elaborating*, *approved*, *rejected*, and others. Since there are different actors interacting with the entity, the entity's state must be persistent.

In an Object-Relational Mapping (ORM) framework [4], the PurchaseRequest entity would be mapped to a PurchaseRequest object, and the PurchaseRequest object does not need to change its behavior when changing its state. This means that persistent states are not necessarily associated with specific behavior. This is the main difference between this problem and the problem solved by the behavioral State Pattern and its extensions. Such patterns are not applicable to persistent states. Since we are dealing with persistent data, we must deal with issues related to persistence, such as transaction management policies.

This paper proposes the Persistent State Pattern, an extension to the *State* pattern where persistent data is taken on account.

The Persistent State Pattern integrates classical and enterprise design patterns, for enterprise applications. Enterprise applications are strongly based on OO design patterns, and the basic reference patterns have been proposed as solutions to problems posed by the Enterprise Java^{TM} Beans (EJB) [3, 17] and the Microsoft^{TM} .NET Framework [7] specifications. The proposed pattern is adherent to existing transaction management frameworks [19, 17] and to any type of persistence framework, including ORM frameworks. We present the close relationship between the Persistent State Pattern and the concepts of model and event driven design. We show that the pattern can be used with both design approaches.

*Corresponding author.

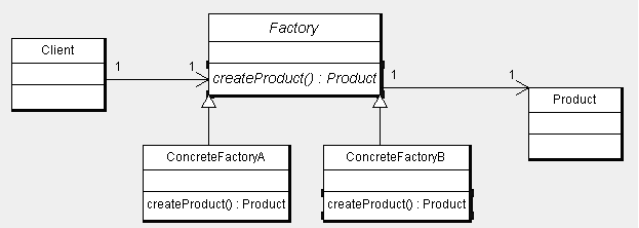

Figure 1: The Abstract Factory design pattern UML class diagram

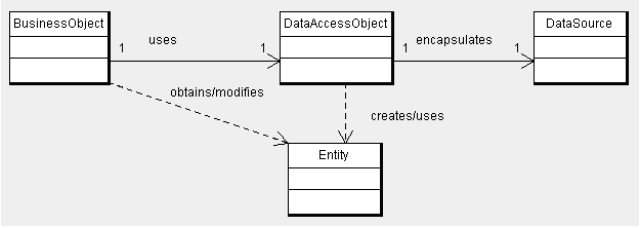

Figure 2: The Data Access Object design pattern UML class diagram.

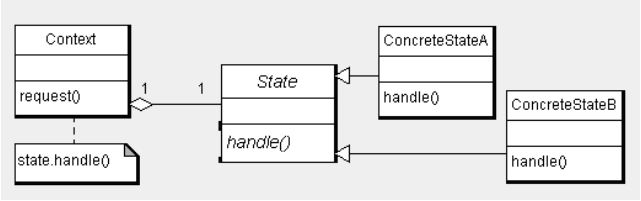

Figure 3: The basic State Pattern UML class diagram

2. CLASSICAL AND ENTERPRISE DESIGN PATTERNS

In this section we recall the background on classical and enterprise design patterns needed for the comprehension of the text.

The book by Gamma *et al.* [10] (known as the Gang of Four or simply GoF) compiles various classical design patterns. In this paper we deal with information systems for the web, with large databases, and with business processes and workflows largely present. Such systems are known as enterprise systems or enterprise applications. GoF patterns are not sufficient for enterprise applications. There are several enterprise design patterns. This paper is based on the Core J2EE enterprise patterns [3]. In the following we present the GoF and Core J2EE needed for this paper. One GoF pattern will be presented separately in Section 3, due to its strongest relationship with the subject of this paper.

2.1 Factory Method and Abstract Factory

The Factory Method and the Abstract Factory are used as the standard way to create objects. The implementation of Factory Method overlaps with that of Abstract Factory in [10]. In Figure 1 we present the most popular implementation.

The goal is to avoid a client to directly instantiate a class. This is especially interesting when the object created must have its lifecycle monitored.

2.2 Command

A Command encapsulates a request as an object. The Command pattern is largely used for events, errors and exceptions. The idea is to have the same interface for several different actions, so the same client can launch different executions by calling the same signature method in different Command objects.

2.3 Data Transfer Object

In enterprise applications, it is usual to implement a database entity as a Data Transfer Object (DTO), an arbitrary serializable Java object. The DTO is usually implemented as a class with only attributes and getter or setter methods. The idea is to avoid network overhead when transfering data in a remote call. So the DTO does not provide fine-grained setter methods for the attributes. The logic for the attributes is implemented by the Business Object. The DTO is sometimes called Value Object [9, 3].

2.4 Business Object

The Business Object is defined by the Core J2EE Patterns as the object that requires access to data. In this paper, the Business Object will always implements the business logic about that data, so the business logic and the persistent data are decoupled in two objects.

2.5 Data Access Object (DAO)

The Data Access Object (DAO) is a Core J2EE pattern. The DAO is an abstraction to the access of a data source. We show its structure in Figure 2.

In Figure 2 the Entity class represents a database entity, and it is implemented as a DTO. The BusinessObject interacts with this DTO by modifying it, based on its business logic.

The DAO abstracts and encapsulates all access to the data source, it manages the connection with the data source to obtain and store data. The data source may be of any kind, a Relation Database Management System (DBMS), an Object-Oriented DBMS, a XML repository, a flat file system, and so forth.

3. STATE PATTERN AND VARIATIONS

The State Pattern is a behavioral software design pattern [10] used to represent the state of an object, a clean way for an object to partially change its type at runtime. It is the basic reference for many other state related patterns. The UML class diagram representing the basic State Pattern is presented in Figure 3.

As Figure 3 shows, the State Pattern is a solution to the problem of creating a state dependent behavior. The *Context* class is the interface with the client. The context is associated with the *State* abstract class, whose method *handle()* represents the state behavior. Concrete classes that extend *State* must give different implementations of the method

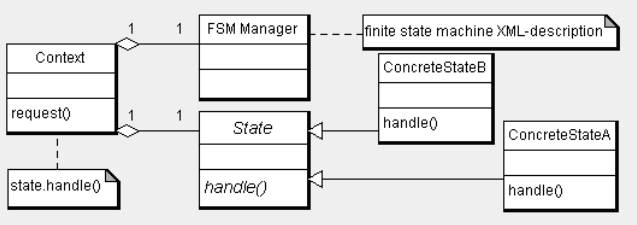

Figure 4: A representation of FSM Pattern

handle(), and each of these classes (e.g. *ConcreteStateA* and *ConcreteStateB*) is a different state.

The State pattern does not specify where the state transition logic is defined. It can be defined in the *Context* object or in individual *ConcreteState* classes. However, by defining the transition logic in concrete states introduces dependencies between subclasses, which is an undesired coupling.

The unclear information about where to define transition logic is only a very simple limitation of the State Pattern. The authors in [1, 2, 20] shows many extensions of the basic State Pattern, all able to solve a specific problem. Some problems solved are related to flexibility of design [12, 16], loose coupling between elements [15, 8], performance [5] ability of reverting states [16].

3.1 The FSM Pattern

In this paper we are especially interested in aspects of loose coupling between elements and the ability to revert states. Loose coupling between elements has been largely studied. The State Pattern is extended by a Finite State Machine (FSM) Pattern [18]. Relationship between statecharts and state machines are treated by the Basic Statechart Pattern and by the Hierarchical Statechart Pattern [20, 21]. Nowadays we have advanced free software that implements FSM and generates code from statecharts [11, 14]. Considering the evolution of statechart tools, we give a representation of a FSM Pattern in Figure 4.

In this representation, the transition logic is described in an XML file, and it is interpreted by a generic and reusable class named FSM Manager. The FSM Manager decouples transition logic from the State Pattern. This representation could be seen as a new pattern, while in fact it is an interpretation of the patterns cited above.

An example of XML file describing the FSM is presented in Figure 5.

In this XML the FSM has two states (0 and 1). The state "1" is assotiated to an entry action, which must be executed when the FSM enters this state. There is only one transition declared, from state "0" to state "1", and this transition is executed if the attribute *value*, of the entity Example is greater than 1000.

An outline of the generic FSM manager is presented in Figure 6.

```
<fsm startState="0">
  <state id="0"/>
  <state id="1">
    <entryAction service="AnotherService"/>
  </state>
  <transition from="0" to="1" id="0">
    <guardCondition type="greaterThan">
      <attribute>value</attribute>
      <constant>1000</constant>
    </guardCondition>
  </transition>
</fsm>
```

Figure 5: ExampleFSM.xml, an XML description of the FSM statechart for an entity called Example

```
public class FSMManager {
  public FSMManager(String entityName) {
    // load <entityName>FSM.xml file
  }
  public static FSMManager getFSMManager(
      String entityName) {
    // optionally consult cache
    return new FSMManager(entityName);
  }
  public void tryStateChange(
      Object e) throws Exception {
    /*
     * state <- e.state, by reflection
     * get transitions from state e.state
     * for each Transition tr
     *    test its guard conditions
     *    if condition is true, call changeState(tr,e)
     */
  }
  private void changeState(
      Transition tr, Object e)
      throws Exception {
    /* e.state <- tr.toState
     * for each entry action of new state
     *    call action.execute(t)
     */
  }
}
```

Figure 6: FSMManager.java, a Java outline for the generic FSM manager

```
public class Action {
  String serviceName;
  public void execute(int param) {
    // call <serviceName>Service.execute(param),
    // by reflection
  }
}
```

Figure 7: Action.java, a Java outline for an Action

The FSMManager class has at least the two methods presented in Figure 6. The method *tryStateChange* analyses the entity attributes and interprets if there is a state change to be performed. If there is a state change, the method *changeState* is called. The *changeState* method may need to execute actions. If the only transition described in the XML of Figure 5 is performed, the FSM will enter the state "1", and the entry action must be executed. The FSMManager class may instantiate Actions in its constructor, based on the information of the XML file. An action can be the simple class presented in Figure 7.

The problem of reverting object states has been much less studied, since the solution proposed in [16] is usually enough. We could not find in the literature a pattern that solves the problem of reverting a state of a persistent object during a database transaction. When dealing with databases, the ability to revert the state of an object requires the ability to revert the state of the database. We need a pattern that links the FSM Manager to a database transaction manager, so to be able to perform rollbacks in the transaction when a state transition cannot not be accomplished. In the next section we describe such pattern.

4. PERSISTENT STATE PATTERN DESCRIPTION

The State Pattern is a behavioral pattern. All its derived patterns are also clearly focused on the object's behaviour. In many applications, FSMs may be applied to objects that simply do not change their behaviour. With regard to runtime objects, it sounds odd, but it is true for persistent objects.

A persistent object is an OO representation of a database entity. An entity may be related to a process or a workflow. As an example, let us consider the purchase process of a company. Many actors are involved with this process, and the complexity of the process varies depending on the company. In our example, we consider that somebody in the company's operation starts the process by creating a purchase request, modeled by the PurchaseRequest entity. A purchase request may be approved or rejected by the company financial office. The PurchaseRequest entity changes its state along the execution of the purchase process. Figure 8 shows the UML statechart for the PurchaseRequest entity.

The statechart for the PurchaseRequest entity is simple, but it is possible to observe that it cannot be executed without persistence, since there are different actors interacting with the entity to cause state transitions.

In the following, we define a new OO design pattern, conceived for persistent states, the Persistent State Design Pattern.

4.1 Context

Several entities in an entity-relationship model may be related to processes or workflows. Such entities may assume various states during a process or workflow execution. An entity in this situation is a FSM. The FSM may launch actions in cascade, and make many changes in the database. The FSM must be managed in a way to allow rollbacks in database transaction if any problem occurs in any level of the cascade.

4.2 Intent

The intent of this pattern is to allow a persistent object to have its behavior and lifecycle managed by a FSM Manager while maintaining the integrity of the database.

4.3 Problem

The State Pattern and its extensions do not present many solutions to the case when reverting object states is important. The available solutions are still focused on runtime behavioral change.

The State Pattern and its extensions also do not present any solution to the case when the objects are persistent. In such case, reverting object states is important, and this may lead to the need of reverting a database state, by controlling the database transactions and executing rollbacks.

4.4 Solution

Use the Persistent State Pattern to integrate an FSM Pattern in a framework with managed transactions.

4.5 Structure

In Figure 9, we show the class diagram representing the relationships for the Persistent State Pattern.

In Figure 10, we show a sequence diagram to illustrate the interaction between the various participants in the Persistent State Pattern.

The participants and the responsibilities of this pattern are described below.

- **Service**
  The Service represents the data client. It is the object that requires access to the data source to obtain and store data. A Service may call other services directly after the sequence of Figure 10. A Service may also call other services indirectly, if the transition logic of the FSM launches actions that call other services. The Service in this pattern represents a primary Service, i.e., a Service which has not been called by another Service. The Service called by another Service has been represented by participant Other Service, in Figure 10.

- **TransactionFactory**
  The TransactionFactory implements the Abstract Factory pattern [GoF]. It is used to create and control a

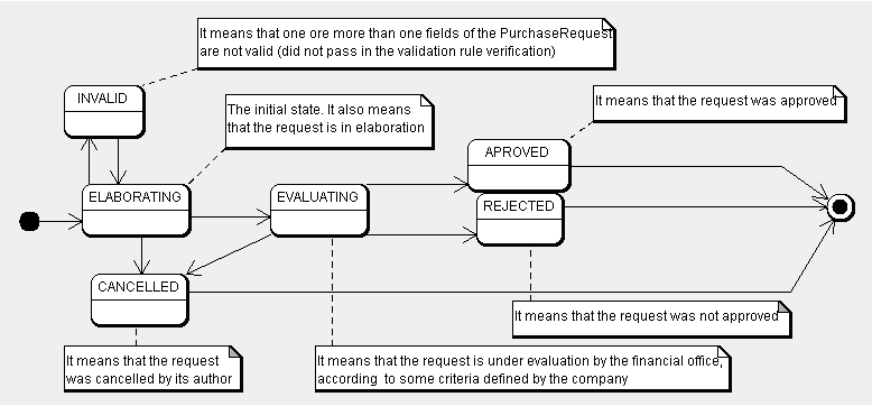

Figure 8: UML statechart for the PurchaseRequest entity

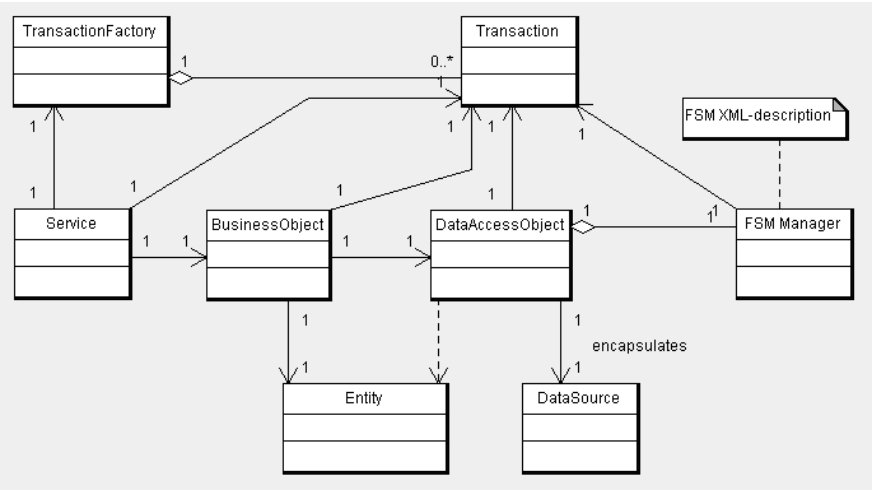

Figure 9: Class diagram representing the relationships for the Persistent State Pattern.

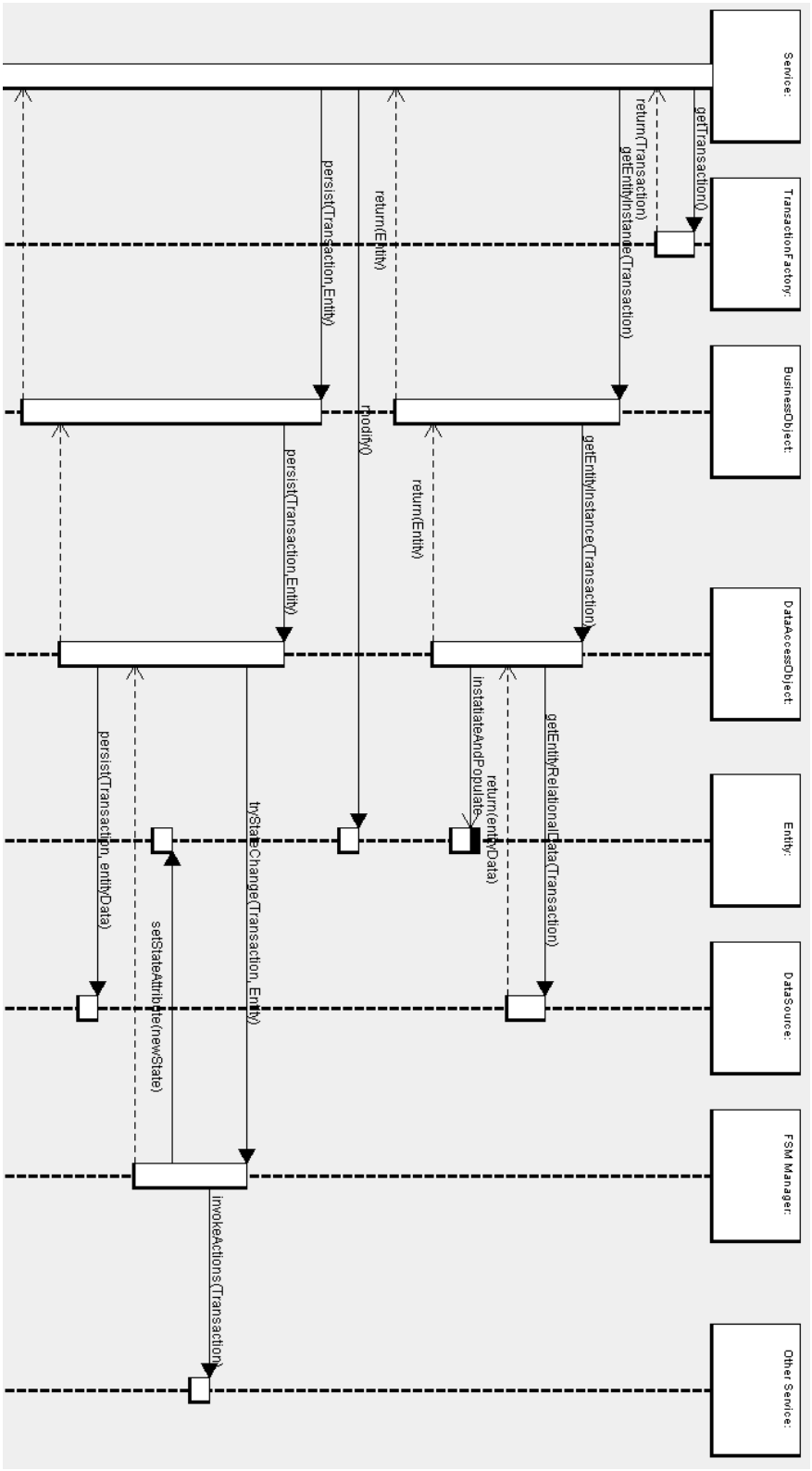

Figure 10: Sequence diagram to illustrate the interaction between the various participants in the Persistent State Pattern.

Transaction lifecycle, so it is possible to perform transaction control in the service level. The idea is to not allow the Service to create an instance of Transaction, so the TransactionFactory can monitor its lifecycle safely. We discuss deeper about the Transaction Factory in Section 6.

- **Transaction**
  The Transaction represents a database open transaction. The Transaction is requested by the Service to the Transaction Factory. The Transaction Factory creates a data transaction and starts monitoring its lifecycle. The Service will pass the Transaction as parameter to every method call that may result in a database access. Every other participant in the sequence does the same. Notice the Transaction being passed as parameter from left to right in Figure 10. When the Service execution finishes, the Transaction is closed. If any error occurs and the Service execution is not finished, the Transaction Factory will perform the rollback of all operations executed by the Transaction.

- **Business Object**
  The Business Object implements the business logic for the persistent entity.

- **Data Access Object (DAO)**
  The DAO is an abstraction layer for the communication with the database.

- **Entity**
  The Entity is the persistent object. An Entity represents an entity of an entity-relationship model. The DAO is responsible for translating this object to the format of the database in use.

- **DataSource**
  This represents a data source implementation. A data source could be a database such as a Relation Database, OO Database, XML repository, flat file system, and so forth.

- **FSM Manager**
  The FSM Manager is responsible to control transition logic of the Finite State Machine (FSM) description. It has the same role as the FSM Manager of the FSM Pattern presented in Figure 4.

- **Other Service**
  The Other Service represents a Service that is called by a primary Service or by another Service in a cascade Service call.

4.6 Integrating Behavior

The diagrams presented above are the core of the Persistent State Pattern. There is no runtime behavioral variation when state changes. Indeed, the Persistent State Pattern itself does not cover the solution given by the basic State Pattern. However, they can be combined.

The participant that implements any behavior is the BusinessObject, since it is responsible for the implementation of the business logic. In the context of enterprise applications, if there is an object that may change its behavior in runtime, this object is the BusinessObject.

The Persistent State Pattern can be combined with the FSM Pattern (Figure 4) to cover this problem. The BusinessObject is made abstract, and it must be extended by concrete BusinessObjects, each one representing the behavior of one state of the FSM. The transition logic of the FSM is maintained in the FSM Manager.

In Figure 11, we show the class diagram representing the relationships for the Persistent State Pattern combined with the FSM Pattern, to include behavioral changes.

The participants and the responsibilities that change in this combination are described below.

- **BusinessObject, BusinessObjectA, and BusinessObjectB**
  The BusinessObject becomes an abstract class, just like the State class of the basic State Pattern presented in Figure 3. The participants BusinessObjectA, and BusinessObjectB are concrete versions of the BusinessObject, implementing specific behavior for each state. BusinessObjectA, and BusinessObjectB have the same role as ConcreteStateA and ConcreteStateB in Figure 3. The abstract BusinessObject has now access to the FSM XML-description. The FSM XML-description must map each state to a concrete BusinessObject implementation. That way, the abstract BusinessObject is able to instantiate each concrete BusinessObject. The *updateState() : BusinessObject* will return the correct concrete instance based on the current state.

- **FSM Manager**
  In the behavioral version of the Persistent State Pattern the FSM Manager shares the FSM XML-description file with the abstract BusinessObject.

All the other participants have the same responsibilities as they had in the simpler version of the pattern.

4.7 Related patterns

All the following patterns are related with this pattern:

- **Abstract Factory [GoF]**
  The Abstract Factory pattern is used to create and control a Transaction lifecycle, so it is possible to perform transaction control in the service level.

- **Business Object [Core J2EE]**
  The Business Object implements the business logic for the persistent entity.

- **Data Access Object (DAO) [Core J2EE]**
  The DAO is an abstraction layer for the communication with the database.

- **Finite State Machine Patterns (FSM)**
  The FSM pattern is used to control transition logic in the simplified version of the Persistent State Pattern. In the behavioral version, it is also responsible to instantiate the concrete Business Objects.

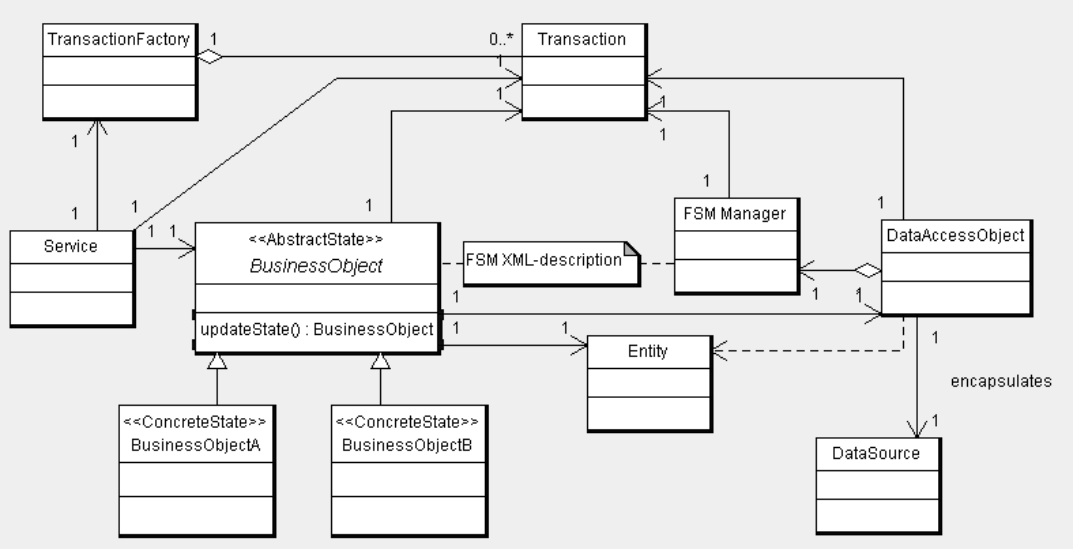

Figure 11: Class diagram representing the relationships for the Persistent State Pattern and the FSM Pattern to include behavioural changes.

4.8 Example

Let us give an example of the use of the Persistent State Pattern. First of all, we update some lines of the FSMManager and the Action classes, such as they handle Transactions. It is shown in Figure 12.

A Transaction is a class that represents a database open transaction. The *rollback* method is responsible for the rollback of the entire transaction.

For our example, we show in Figure 13 the ExampleEntity, the ExampleDAO and the ExampleBO classes, representing an entity, a Data Access Object and a Business Object, respectively.

Note that the ExampleEntity has only two attributes: "state" and "value". The attribute "value" is important to the Example FSM, described by the ExampleXML.xml file in Figure 5. It is part of a guard condition.

The ExampleDAO has only two methods. The *getEntityInstance* method queries the database (represented by a class named MyDataSource) and creates an ExampleEntity instance from relational data. The *persist* method does the opposite, it populates a relational database from data found in an ExampleEntity object. However, the *persist* method is updating an entity, and it may cause a state transition. Thus, the FSMManager generic class is instantiated, and it loads the ExampleFSM.xml file. The ExampleDAO delegates the state transition to the *tryStateChange* method of the FSMManager. If no Exception is thrown by the *tryStateChange*, it means that the ExampleEntity has successfully changed its state, and it can finally be persisted. Otherwise, the Exception is throw to whom has called the *persist* method.

In Figure 14 we show the code for the Business Object.

```
public class Transaction {
  public void rollback() {
    // stub
  }
}

public class FSMManager {
  //...
  public void tryStateChange(Transaction t,
      Object e) throws Exception {
  //...
     *    if condition is true, call changeState(t,tr,e)
  //...
  private void changeState(Transaction t,
      Transition tr, Object e)
      throws Exception {
  //...
     *    call action.execute(t)
  //...
}

public class Action {
  String serviceName;
  public void execute(Transaction t, int param) {
    // call <serviceName>Service.execute(t, param),
    // by reflection
  }
}
```

Figure 12: FSMManager, Action, and the introduction of the Transaction

```
public class ExampleEntity {
  public int state = 0;
  public int value = 0;
}

public class ExampleDAO {
  ExampleEntity getEntityInstance(Transaction t,
      int param) {
    ExampleEntity e = null;
    // get relational data
    MyDataSource ds = new MyDataSource();
    Object o = ds.getEntityRelationalData(t, param);
    //... populate 'e' by reading 'o'
    return e;
  }
  public void persist(Transaction t, ExampleEntity e)
      throws Exception {
    Object o = null;
    // change state if needed
    FSMManager fsm = FSMManager.
        getFSMManager("Example");
    fsm.tryStateChange(t, e);
    //... populate 'o' by reading 'e'
    // persist
    MyDataSource ds = new MyDataSource();
    ds.persist(t, o);
  }
}
```

Figure 13: The ExampleEntity and ExampleDAO classes.

```
public class ExampleBO {
  public ExampleEntity getEntityInstance(
      Transaction t,int param) {
    ExampleDAO dao = new ExampleDAO();
    return dao.getEntityInstance(t, param);
  }
  public void persist(Transaction t, ExampleEntity e)
      throws Exception {
    ExampleDAO dao = new ExampleDAO();
    dao.persist(t,e);
  }
  public void setValue(ExampleEntity e, int value) {
    e.value = value;
  }
}
```

Figure 14: The ExampleBO class

```
public class ExampleService {
  public void execute(int param) {
    // get a transaction
    Transaction t = TransactionFactory.getTransaction();
    try {
      execute(t, param);
    } catch (Exception ex) {
      t.rollback();
    }
  }
  public void execute(Transaction t, int param)
      throws Exception {
    // get entity
    ExampleBO bo = new ExampleBO();
    ExampleEntity e = bo.getEntityInstance(t, param);
    // modify entity
    bo.setValue(e, 3000);
    // persist
    bo.persist(t, e);
  }
}
```

Figure 15: The ExampleService (see text)

Regarding persistence, the BusinessObject always delegates the work to the DAO. The only specific method in the ExampleBO class is *setValue*, which is the setter method for the ExampleEntity.value attribute. In this example, there is no logic implemented, but it could have.

Finally, we present the code for the ExampleService in Figure 15.

Note that the implementation of the service has two methods. The method *execute(int)* and the method *execute(Transaction,int)*. When the service is called from a remote request, a new transaction must be created, and the *execute(int)* must be used. The *execute(int)* creates a Transaction by calling the TransactionFactory (the TransactionFactory is specific to the database or the persistence API), and then calls *execute(Transaction,int)* inside a *try* scope. The *execute(int)* method is the method that starts the sequence presented in Figure 10. From that point on, no other try statement need to be added, because all the subsequent operations are performed in the same Transaction. Note that the transaction aware Action class presented in Figure 12 will always call another service by forwarding the Transactions.

With the Persistent State Pattern, the implementation of the service becomes simpler, since the transaction management is performed by the other participants of the pattern, which have standardized code.

5. PERSISTENT STATE PATTERN FOR MODEL AND EVENT DRIVEN DESIGN

The Persistent State Pattern is directly applicable in contexts of Model Driven Design (MDD) and Event Driven Design (EDD).

The participants presented in Figure 11 are organized. Each participant has its role, and some of them are fix or can be generated from models. The Transaction Factory is a generic component, it is not dependent of the application. The same

is true for Transaction, DataSource, and the FSM Manager. The others are discussed below.

- **Entity**
  The Entity is just a representation of an entity of the database. It has no logic implemented. It is a Data Transfer Object (DTO). All the information necessary to implement such DTO is available in an entity-relationship model (ERM). An ERM may be modeled in a UML class diagram. In this case, a class represents an entity, but no operation is added to the class. There are several tools for automated processing of UML diagrams, since the UML model can be exported to a XML Metadata Interchange (XMI) file. We are able to automatically generate the entities from UML models.
- **Data Access Object (DAO)**
  For each DataSource type, there are rules to map the entity data to the DataSource. It means that, once the DataSource has been defined, it is possible to automatically generate DAOs from the entities definition in UML.
- **Service**
  A Service may implement basic operations or complex ones. Of course, automatic code generation for complex Services from models would certainly depend on other model types, such as Business Process or Business Rules models. However, there are plenty of simple Services that can be automatically generetad from UML class diagram: the Data Services. Data Services are services for data access. A single data Service serves only to create, read, update or delete (CRUD) a single entity of the database. These Services can also be automatically generated.
- **Business Object**
  The Business Object is the most complex participant in the pattern. It implements logic about the entity. One Business Object exists for one Entity. For the version of the Persistent State Pattern that includes behavioural changes, it is difficult to generate fully functional concrete state classes. Only stubs can be generated. However, it is rare the need for behavioural changes in a Business Object. Most of the logic present in a Business Object is related to state transtitions or field validation rules. Both can be described in the UML model, and such definitions can be used for automatic code generation.

In Figures 9 and 11, there is also the FSM XML model, which is essential to the FSM Manager. The FSM XML model is a direct mapping of the UML State Diagram exported to XML. There are available tools that automatically generates such code too [11, 14].

It is clear how the Persistent State Pattern is useful in an MDD context. The EDD concept is also useful, specially for the implementation of the FSM Manager.

In statecharts, transitions and states may have actions associated. An action may be launch when entering (entry action) or exiting (exit action) a state. Transitions may also launch actions. In the MDD approach, such actions will be modeled in the UML State Diagram, and so they will be available in the FSM XML model. The idea is to associate to each action, a *Command* class that implements it.

The EDD approach is a little different. All the actions of a state chart are handled by a single class, and such class implements also an Event Listener. In that approach, the FSM Manager implementation is much simpler, since it will only throw out events without carrying on executing and managing the actions.

6. PERSISTENT STATE PATTERN AND TRANSACTION MANAGEMENT

The most important difference between the Persistent State Pattern and other patterns derived from State is the presence of the TransactionFactory and Transaction participants.

Managed transactions are available for the two most important enterprise development platforms. Java^{TM} Enterprise Edition defines container and bean-managed transactions, based on the Java^{TM} Transactions API [13, 17], the Microsoft^{TM} .NET Framework defines transaction scopes [7]. For the Java^{TM} platform, there is also the Spring Framework transaction management [19].

It is not the scope of this paper to compare these frameworks. Transaction propagation is supported by all of them. As an example, with the Spring Framework, the transaction propagation needed for the Persistent State Pattern can be achieved by setting the propagation to *required*.

When the propagation is set to *required*, a logical transaction scope is created for each method to which the setting is applied. Each such logical transaction scope can determine rollback-only status individually, with an outer transaction scope being logically independent from the inner transaction scope. This means that the Persistent State Pattern can be implemented without loss o generality. By using the pattern, software designers remain flexible for the implementation.

7. DISCUSSION

In this paper we have presented the Persistent State Pattern, a new design pattern for OO programing with persistent states. We have recalled the classical State design pattern definition and we have discussed several extensions of such pattern. We concluded in the research that the State pattern and its extensions are focused on the objects' behaviour, while the problem of persistent state is related to the transaction management. There was not a design pattern available for persistent states.

The Persistent State Pattern has been presented in two forms. In the simple form, where the pattern is sufficient for managing the state transition logic of a persistent object in a managed transaction framework. The simple form is sufficient for the most part of the information systems, where states are related to business processes or workflows. The state of an entity that is part of a process or workflows usually reflects simply the state of the process or workflow

themselves. If the persistent object needs to behave differently for each different state, the behavioral form of the Persistent State Pattern must be used. In this case, the basic of the State pattern is added to the simple Persistent State Pattern, and the result is a pattern that is able to manage different behaviors for persistent objects in an environment with managed database transactions.

We showed that the Persistent State Pattern can be used by software designers without loss of flexibility or generality in programming. The pattern is perfectly adherent to model and event driven design (MDD and EDD). The participants of the pattern can be automatically generated from UML models, which is useful for MDD tools. The state machine management can be implemented in a synchronous approach, where actions are mapped to methods directly, and the execution of an action means calling a method. But it can also be implemented in an event driven approach, where event listeners implement actions, the listeners are registered in the finite state machine (FSM) Manager, and the FSM Manager simply rises an event to inform the listener that an action must be executed.

Finally, we conclude that the Persistent State Pattern fills a lack in the literature on State design patterns and extensions, by providing an extension to the State Pattern to be used in a persistence viewpoint.

Acknowledgements

This work is a result of the brazilian public support to the research in Mitah Technologies private company as well as regular public research funding. The authors would like to thank the brazilian government public funding agencies Fapemig, CNPq and Finep for the financial support.

8. REFERENCES

[1] Adamczyk, P.: The anthology of the finite state machine design patterns. In: Proceedings of the Pattern Languages of Programs Conference (PLoP) (2003)

[2] Adamczyk, P.: Selected patterns for implementing finite state machines. In: Proceedings of the Pattern Languages of Programs Conference (PLoP) (2004)

[3] Alur, D., Malks, D., Crupi, J.: Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall / Sun Microsystems Press, 2nd edn. (May 2003)

[4] Ambler, S.W.: Agile Database Techniques: Effective Strategies for the Agile Software Developer. John Wiley & Sons (2003)

[5] Douglas, B.P.: Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns. Addison Wesley (1998)

[6] Elmasri, R., Navathe, S.: Fundamentals of Database Systems. Addison Wesley, 6th edn. (2010)

[7] Esposito, D., Saltarello, A.: Microsoft .NET: Architecting Applications for the Enterprise (PRO-Developer). Microsoft Press, 1st edn. (Oct 2008)

[8] Ferreira, L., Rubira, C.M.F.: The reflective state pattern. In: Proceedings of Pattern Languages of Programs Conference (PLoP) (1998)

[9] Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 1st edn. (Nov 2002)

[10] Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (Nov 1994)

[11] Gurov, V., Mazin, M.: UniMod project website, http://unimod.sourceforge.net

[12] van Gurp, J., Bosch, J.: On the implementation of finite state machines. In: Proceedings of the IASTED International Conference (1999)

[13] Jendrock, E., Ball, J., Carson, D., Evans, I., Fordin, S., Haase, K.: Java(TM) EE 5 Tutorial. Prentice Hall, 3rd edn. (Nov 2006)

[14] Korotkov, M.: Automatic layout of state diagrams. White Paper - UniMod website, http://unimod.sourceforge.net/articles.html

[15] Martin, R.: Three Level FSM. In: Proceedings of Pattern Languages of Program Design (PLoPD) (1995)

[16] Odrowski, J., Sogaard, P.: Pattern integration - variations of state. In: Proceedings of Pattern Languages of Programs (PLoP) (1996)

[17] Panda, D., Rahman, R., Lane, D.: EJB 3 in Action. Manning Publications, 1st edn. (Apr 2007)

[18] Shalyto, A., Shamgunov, N., Korneev, G.: State machine design pattern. In: 4th Intl. Conf. on .NET Thechnologies. pp. 51–57 (Jun 2006)

[19] Walls, C., Breidenbach, R.: Spring in Action. Manning Publications, 2nd edn. (Aug 2007)

[20] Yacoub, S., Ammar, H.: Finite state machine patterns. In: Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP) (1998)

[21] Yacoub, S., Ammar, H.: A pattern language of statecharts. In: Proceedings of Pattern Languages of Programming Conference (PLoP) (1998)