Persistent State Pattern

ANDRÉ V. SAÚDE, Federal University of Lavras, Department of Computer Science
RICARDO A. S. S. VICTÓRIO, Mitah Technologies Inc.
GABRIEL C. A. COUTINHO, Mitah Technologies Inc.

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 Pattern has been introduced. Various design patterns were derived from the State Pattern, but all of the focus on object's behavior. This paper describes the *Persistent State Pattern*, an extension to the State Pattern for persistent data. The *Persistent State Pattern* integrates transaction management with classical and enterprise design patterns. It can be used with OO databases and relational databases, and it can also be incorporated by an object-relation mapping framework. We show how the *Persistent State Pattern* may be useful for model or event driven development.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.2.10 [**Design**] Methodologies; H.2.8 [**Database Management**]: Database Administration—*Security, integrity, and protection*

General Terms: Design

Additional Key Words and Phrases: Persistent states, state machine, design patterns

ACM Reference Format:

Saúde, A. V., Victório, R. A. S. S. and Coutinho, G. C. A. 2010. Persistent State Pattern. Proc. 17th Conference on Pattern Languages of Programs (PLoP), (October 2010), 16 pages.

1. INTRODUCTION

Finite State Machines (FSM) provide a powerful tool to describe dynamic behavior of systems and components. There are several implementations of FSM in Object-Oriented (OO) languages. OO implementations of FSM have been widely studied, and several existent design patterns deal with states in OO. The basic design pattern for states is the State Pattern, popularized by the most cited design pattern reference [Gamma et al. 1994].

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 behavior [Adamczyk 2003; 2004; Yacoub and Ammar 1998a; 1998b]. None of them deals with states in a persistence framework.

Persistent states exist when a database entity (of an Entity-Relationship Model (ERM) [Elmasri and Navathe 2010]), takes part in a 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.

Enterprise applications are usually OO, and they usually rely on Object-Relational Mapping (ORM) frameworks [Ambler 2003]. In ORM, a database entity is mapped to a class. An instance of such class is an object that

This work is supported by Mitah Technologies Inc., Fapemig, CNPq and Finep.
Author's address: A. V. Saúde, Departamento de Ciência da Computação, Universidade Federal de Lavras, CP 3037, CEP 37200-000, Lavras/MG, Brazil; email: saude@dcc.ufla.br. R. A. S. S. Victório, G. C. A. Coutinho, Mitah Technologies Inc., Av. Álvaro A. Leite, 370, CEP 37200-000, Lavras/MG, Brazil; email: {ricardo.victorio,gabriel}@mitahtech.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 17th Conference on Pattern Languages of Programs (PLoP). PLoP'10, October 16-18, Reno, Nevada, USA. Copyright 2010 is held by the author(s). ACM 978-1-4503-0107-7

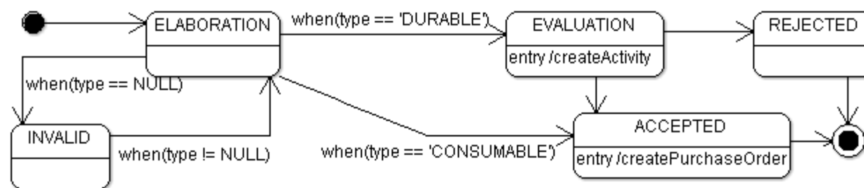

Fig. 1. UML statechart for the PurchaseRequest entity. See text.

may not change its behavior when changing its state. This means that persistent states are mostly not 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) [Alur et al. 2003; Panda et al. 2007] and the Microsoft^{TM} .NET Framework [Esposito and Saltarello 2008] specifications. The proposed pattern is adherent to existing transaction management frameworks [Panda et al. 2007; Walls and Breidenbach 2007] 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.

2. RUNNING EXAMPLE

We start our explanation by a running example, so the problem can be clearly exposed. Let us consider the purchase process of a company. Many actors are enrolled with this process, and the complexity of the process varies depending on the company. So, we simplify such purchase process by splitting it in two steps: i) the purchase request, which is launched by an operator and must be approved by the financial office, and ii) the purchase order, which is an approved purchase, and must be performed by the company's purchase division.

While modeling the purchase process, we create two entities: PurchaseRequest and PurchaseOrder. An instance of PurchaseRequest is created when the company's operator starts the purchase process by requesting to purchase something. An instance of PurchaseOrder is created only if the purchase request is approved.

A PurchaseRequest may pass by several states, from the elaboration to the approval. One possible state machine model for the PurchaseRequest entity is represented in Figure 1 by a UML statechart. We consider a constraint where the PurchaseRequest must be of type 'DURABLE' or 'CONSUMABLE'. Any purchase of durable goods must be approved by a superior.

The state machine in Figure 1 has five states: ELABORATION, INVALID, EVALUATION, ACCEPTED or REJECTED. This state machine starts when an operator creates a purchase request, i.e., an instance of the PurchaseRequest entity. By creating a PurchaseRequest, the state machine starts and goes to the ELABORATION state. While the operator has not yet finished to fill out the form for the PurchaseRequest, the entity stays in the ELABORATION state. When the operator submits the form, there are three triggers based on the 'type' attribute. If the type is not set, the PurchaseRequest goes to the INVALID state. If it is a purchase request for consumable goods, it goes directly to the ACCEPTED state, and if the request is for durable goods, it goes to the EVALUATION state. In the EVALUATION state, an entry action is set: 'createActivity'. So, when this state is achieved, an activity must be created for who will evaluate the request. If the request is accepted, it goes to the ACCEPTED state. Otherwise, it goes to the REJECTED state. The ACCEPTED state also has an entry action: 'createPurchaseOrder'.

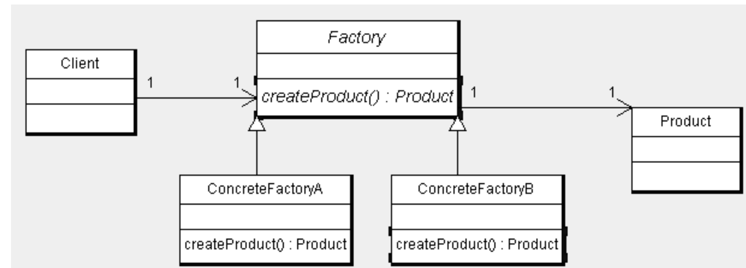

Fig. 2. The Abstract Factory Pattern UML class diagram

So, when this state is achieved, a PurchaseOrder instance is created, and the second step of the purchase process starts.

The example statechart is very simple, but it is possible to observe that it cannot be executed without persistence, since there are different actors (operator and its superior) interacting with the entity to cause state transitions. Furthermore, in an Enterprise Resource Planning (ERP) system, it may be a requirement to keep the history of all the requested purchases, accepted or rejected, thus the PurchaseRequest must be persistent.

Suppose that the operator's superior accepts the request. The PurchaseRequest is persisted and the 'createPurchaseOrder' action is launched. If the 'createPurchaseOrder' action fails, for any reason, the persistent data will become inconsistent, since there will be an accepted purchase request but no purchase order has started. In this case, the PurchaseRequest state should be reverted to EVALUATION, so the operator's superior could try again his activity of evaluating and accepting the request.

This problem is always present when a state machine is associated to a persistent entity and actions are executed as consequence of state transitions. Depending on the complexity of the action, there may be several reasons for the action to fail, and the integrity of the system's database must be guaranteed.

3. PATTERNS BACKGROUND

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.* [Gamma et al. 1994] (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 [Alur et al. 2003]. In the following we present the GoF and Core J2EE needed for this paper. The background on the State Pattern is presented separately in Section 4, due to its strongest relationship with the subject of this paper.

3.1 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 [Gamma et al. 1994]. In Figure 2 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.

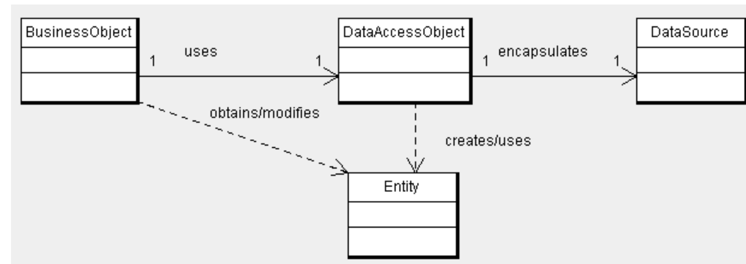

Fig. 3. The Data Access Object Pattern UML class diagram.

3.2 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.

3.3 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 transferring 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 [Alur et al. 2003; Fowler 2002].

3.4 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 implement the business logic about those data, so the business logic and the persistent data are decoupled in two objects.

3.5 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 3.

In Figure 3 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.

4. STATE PATTERN AND VARIATIONS

The State Pattern is a behavioral software design pattern [Gamma et al. 1994] 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 4.

As Figure 4 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 the State Pattern must give different implementations of the method *handle()*, and each of these classes (e.g. *ConcreteStateA* and *ConcreteStateB*) is a different state.

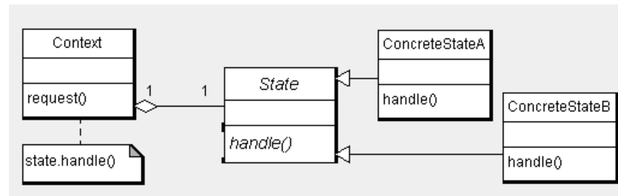

Fig. 4. The basic State Pattern UML class diagram

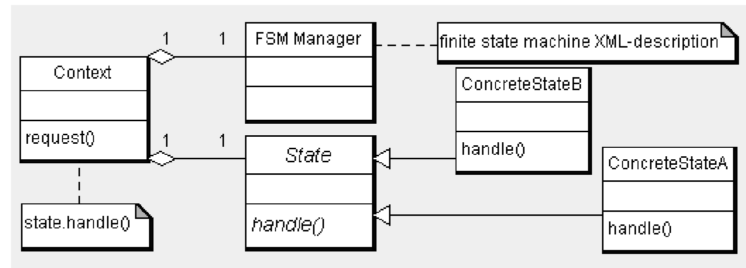

Fig. 5. A representation of FSM Pattern

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, 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 [Adamczyk 2003; 2004; Yacoub and Ammar 1998a] shows many extensions of the basic State Pattern, all able to solve a specific problem. Some problems solved are related to flexibility of design [van Gurp and Bosch 1999; Odrowski and Sogaard 1996], loose coupling between elements [Ferreira and Rubira 1998; Martin 1995], performance [Douglas 1998] ability of reverting states [Odrowski and Sogaard 1996].

4.1 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 [Shalyto et al. 2006]. Relationship between statecharts and state machines are treated by the Basic Statechart Pattern and by the Hierarchical Statechart Pattern [Yacoub and Ammar 1998a; 1998b]. Nowadays we have advanced free software that implements FSM and generates code from statecharts [Gurov and Mazin 2010; Korotkov 2010]. Considering the evolution of statechart tools, we give a representation of a FSM Pattern in Figure 5.

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 XML file describing the FSM is presented in Figure 6. It describes two states and one transition of Figure 1. An outline Java code of the generic FSM manager is presented in Figure 7.

The FSMManager class has at least the two methods presented in Figure 7. 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 6 is performed, the FSM will enter the state “1”, and the entry action must be

```xml
<fsm startState="ELABORATION">
  ...
  <state name="ELABORATION" id="0"/>
  <state name="EVALUATION" id="1">
    <entryAction service="createActivity"/>
  </state>
  <transition from="0" to="1" id="0">
    <trigger>
      <expression>type=='DURABLE'</expression>
    </trigger>
  </transition>
  ...
</fsm>
```

Fig. 6. PurchaseRequestFSM.xml, part of the XML description of the FSM statechart for the PurchaseRequest entity

```java
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 trigger's expressions
     *    if expression evaluates to 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(param)
     */
  }
}
```

Fig. 7. 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
    */
  }
}
```

Fig. 8. Action.java, a Java outline for an Action

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 8.

While loose coupling has been largely studied, the problem of reverting object states has not, because the solution proposed in [Odrowski and Sogaard 1996] is usually enough. We could not find in the literature a solution to 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 to link the FSM Manager to a database transaction manager, so to be able to perform rollbacks in the transaction when a state transition cannot be accomplished. The *Persistent State Pattern* solves this problem.

5. PATTERN DESCRIPTION

The State Pattern and its derived patterns are clearly focused on object's behavior. None of them deal with the need of persisting states. The *Persistent State Pattern* fills the lack of patterns for applications with persistent states.

5.1 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 invoke actions, actions may invoke services in cascade, and many changes may be persisted in cascade. The FSM must be transaction aware, and its transaction must be managed in a way to allow database rollbacks if any problem occurs in any level of the cascade.

5.2 Intent

The intent of this pattern is to provide a safe design for the use of finite state machines associated to persistent objects that participate to business processes or workflows.

5.3 Problem

When a state machine is associated to a persistent entity, and actions are executed as consequence of state transitions, a new constraint is imposed: database integrity must be guaranteed. Several reasons may cause an action to fail. If an action fails for any reason, the system must be able to rollback all the database changes since the first state change occurred, to guarantee the data consistency.

5.4 Solution

Use the *Persistent State Pattern* to integrate transaction management to finite state machines.

5.5 Structure

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

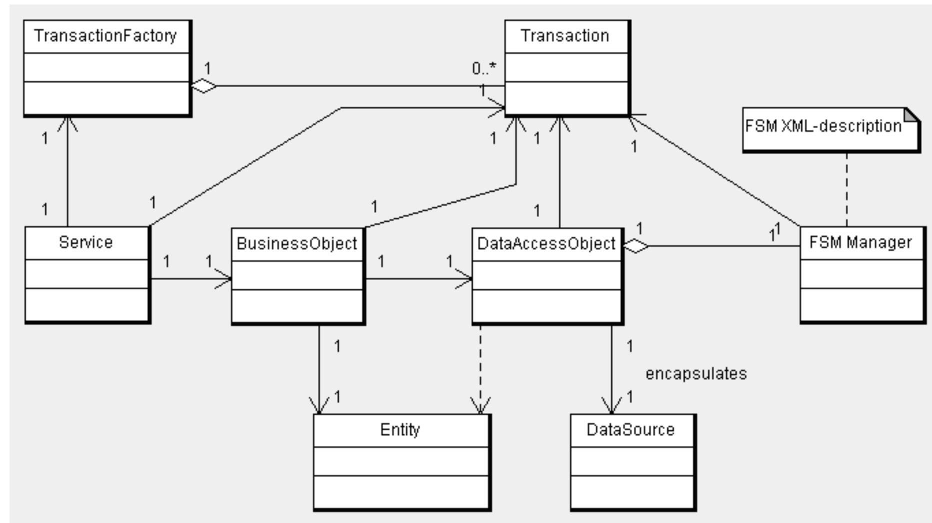

Fig. 9. Class diagram representing the relationships for the *Persistent State Pattern*.

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 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 TransactionFactory in Section 7.

—**Transaction**
The Transaction represents a database open transaction. The Transaction is requested by the Service to the TransactionFactory. The TransactionFactory 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 TransactionFactory 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.

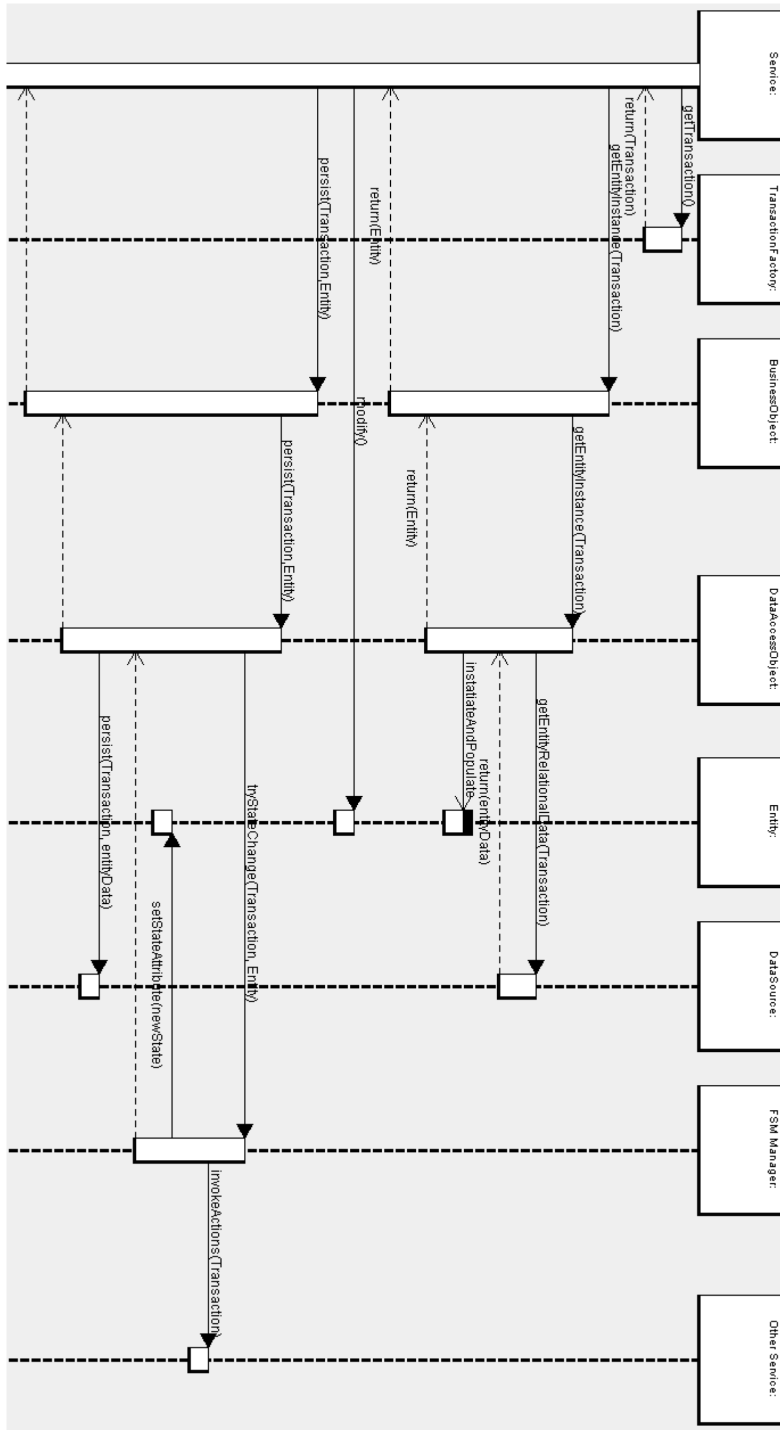

Fig. 10. Sequence diagram to illustrate the interaction between the various participants in the *Persistent State Pattern*.

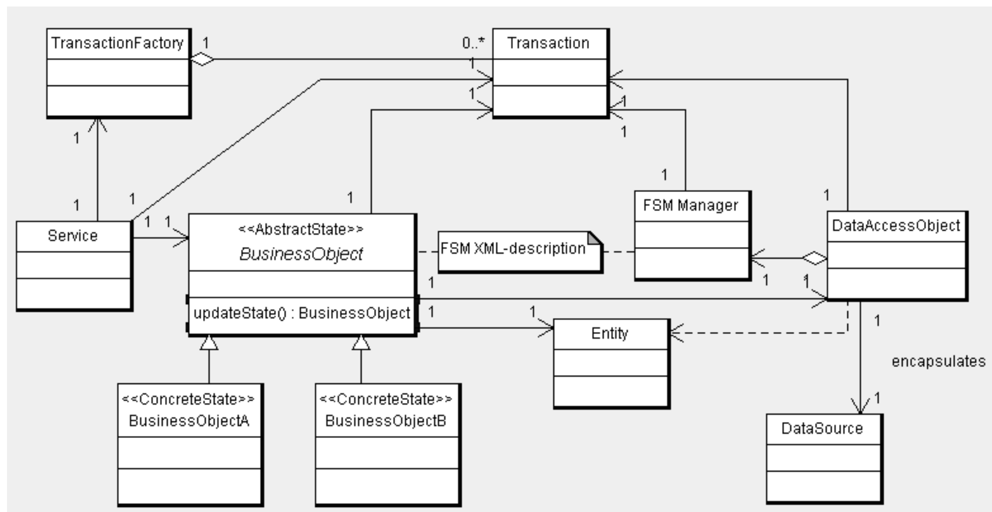

Fig. 11. Class diagram representing the relationships for the *Persistent State Pattern* and the FSM Pattern to include behavioral changes.

—**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 5.

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

5.6 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 5) 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 4. 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 4. 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.

5.7 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.

5.8 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 code for the PurchaseRequest, the PurchaseRequestDAO, and the GoodType classes. PurchaseRequest represents the entity, PurchaseRequestDAO represents its correspondent Data Access Object. GoodType is a simple enumeration for the type of request.

Note that we have shown only two attributes for the PurchaseRequest entity: 'state' and 'type'. The 'type' attribute is important to the PurchaseRequest FSM, described by the PurchaseRequestFSM.xml file in Figure 6. It is part of the trigger expression.

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

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

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

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

Fig. 12. FSMManager, Action, and the introduction of the Transaction

In Figure 14 we show the code for the PurchaseRequestBO class, which is the Business Object for the PurchaseRequest entity. Regarding persistence, the Business Object will always delegate the work to the DAO. The only specific method in the PurchaseRequestBO class is *setType*, which is the setter method for the PurchaseRequest.type attribute. The code that implements logic has been omitted.

Finally, we present in Figure 15 the code for the ExampleService, a service that manipulates data on a PurchaseRequest. 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 used by the application), 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 Action class presented in Figure 12 is Transacion aware, and it will always forward the Transaction when calling another service.

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.

```
public class PurchaseRequest {
  public int state = 0;
  public GoodType type;
}

public class PurchaseRequestDAO {
  public PurchaseRequest getEntityInstance(Transaction t,
      int param) {
    PurchaseRequest 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, PurchaseRequest e)
      throws Exception {
    Object o = null;
    // change state if needed
    FSMManager fsm = FSMManager.
        getFSMManager("PurchaseRequest");
    fsm.tryStateChange(t, e);
    //... populate 'o' by reading 'e'
    // persist
    MyDataSource ds = new MyDataSource();
    ds.persist(t, o);
  }
}

public enum GoodType {
  DURABLE, CONSUMABLE
}
```

Fig. 13. The PurchaseRequest and PurchaseRequestDAO classes.

6. THE PERSISTENT STATE PATTERN FOR MODEL AND EVENT DRIVEN DEVELOPMENT

The *Persistent State Pattern* is directly applicable to the contexts of Model Driven Development (MDD) and Event Driven Development (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 TransactionFactory is a generic component, it is not dependent of the application. The same is true for Transaction, DataSource, and FSMManager. 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 described by 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. We are able to automatically generate the entities from UML models.

```
public class PurchaseRequestBO {
  public PurchaseRequest getEntityInstance(
      Transaction t,int param) {
    PurchaseRequestDAO dao = new PurchaseRequestDAO();
    return dao.getEntityInstance(t, param);
  }
  public void persist(Transaction t, PurchaseRequest e)
      throws Exception {
    PurchaseRequestDAO dao = new PurchaseRequestDAO();
    dao.persist(t,e);
  }
  public void setType(PurchaseRequest e, GoodType type) {
    // ... some logic implementation
    e.type = type;
  }
}
```

Fig. 14. The PurchaseRequestBO 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
    PurchaseRequestBO bo = new PurchaseRequestBO();
    PurchaseRequest e = bo.getEntityInstance(t, param);
    // modify entity
    bo.setType(e, GoodType.DURABLE);
    // persist
    bo.persist(t, e);
  }
}
```

Fig. 15. The ExampleService (see text)

—**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 generated 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 behavioral changes, it is difficult to generate fully functional concrete state classes. Only stubs can be generated. However, it is rare the need for behavioral changes in a Business Object. Most of the logic present in a Business Object is related to state transitions 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 [Gurov and Mazin 2010; Korotkov 2010].

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.

7. THE PERSISTENT STATE PATTERN AND TRANSACTION MANAGEMENT

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

Managed transactions are available for enterprise development platforms. Java^{TM} Enterprise Edition defines container and bean-managed transactions, based on the Java^{TM} Transactions API [Jendrock et al. 2006; Panda et al. 2007], the Microsoft^{TM} .NET Framework defines transaction scopes [Esposito and Saltarello 2008]. For the Java^{TM} platform, there is also the Spring Framework transaction management [Walls and Breidenbach 2007].

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.

8. DISCUSSION

We have presented the *Persistent State Pattern*, a design pattern for OO programing with persistent states. We have recalled the classical State Pattern definition and the extensions of such pattern. We have shown that the State Pattern and its extensions are focused on the objects' behavior, while the problem of persistent states is related to the transaction management. There was no published design pattern 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 development. The participants of the pattern can be automatically generated from UML models, which is useful for MDD. 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 patterns for state management, by providing an extension to the State Pattern to be used in a persistence viewpoint.

Acknowledgments

This work is a result of the Brazilian public support to the research in Mitah Technologies Inc. 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.

REFERENCES

ADAMCZYK, P. 2003. The anthology of the finite state machine design patterns. In *Proceedings of Pattern Languages of Programs (PLoP)*.

ADAMCZYK, P. 2004. Selected patterns for implementing finite state machines. In *Proceedings of Pattern Languages of Programs (PLoP)*.

ALUR, D., MALKS, D., AND CRUPI, J. 2003. *Core J2EE Patterns: Best Practices and Design Strategies* 2nd Ed. Prentice Hall / Sun Microsystems Press.

AMBLER, S. W. 2003. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. John Wiley & Sons.

DOUGLAS, B. P. 1998. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Addison Wesley.

ELMASRI, R. AND NAVATHE, S. 2010. *Fundamentals of Database Systems* 6th Ed. Addison Wesley.

ESPOSITO, D. AND SALTARELLO, A. 2008. *Microsoft .NET: Architecting Applications for the Enterprise (PRO-Developer)*. Microsoft Press.

FERREIRA, L. AND RUBIRA, C. M. F. 1998. The reflective state pattern. In *Proceedings of Pattern Languages of Programs (PLoP)*.

FOWLER, M. 2002. *Patterns of Enterprise Application Architecture* 1st Ed. Addison-Wesley Professional.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. M. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

GUROV, V. AND MAZIN, M. 2010. UniMod project website. Access in September – http://unimod.sourceforge.net.

JENDROCK, E., BALL, J., CARSON, D., EVANS, I., FORDIN, S., AND HAASE, K. 2006. *Java(TM) EE 5 Tutorial* 3rd Ed. Prentice Hall.

KOROTKOV, M. 2010. Automatic layout of state diagrams. White Paper - UniMod website. Access in September – http://unimod.sourceforge.net/articles.html.

MARTIN, R. 1995. Three Level FSM. In *Proceedings of Pattern Languages of Program Design (PLoPD)*.

ODROWSKI, J. AND SOGAARD, P. 1996. Pattern integration - variations of state. In *Proceedings of Pattern Languages of Programs (PLoP)*.

PANDA, D., RAHMAN, R., AND LANE, D. 2007. *EJB 3 in Action* 1st Ed. Manning Publications.

SHALYTO, A., SHAMGUNOV, N., AND KORNEEV, G. 2006. State machine design pattern. In *4th Intl. Conf. on .NET Thechnologies*. 51–57.

VAN GURP, J. AND BOSCH, J. 1999. On the implementation of finite state machines. In *Proceedings of the IASTED International Conference*.

WALLS, C. AND BREIDENBACH, R. 2007. *Spring in Action* 2nd Ed. Manning Publications.

YACOUB, S. AND AMMAR, H. 1998a. Finite state machine patterns. In *Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP)*.

YACOUB, S. AND AMMAR, H. 1998b. A pattern language of statecharts. In *Proceedings of Pattern Languages of Programs (PLoP)*.

PLoP'10, October 16-18, Reno, Nevada, USA. Copyright 2010 is held by the author(s). ACM 978-1-4503-0107-7