The Reflective State Pattern[1]

Luciane Lamour Ferreira Cecília M. F. Rubira
Institute of Computing - State University of Campinas
P.O. Box 6176, Campinas, SP 13083-970
e-mail: {972311, cmrubira}@dcc.unicamp.br

1 Abstract

This paper presents the Reflective State pattern that is a refinement of the State design pattern [GHJV95] based on the Reflection architectural pattern [BMRS+96]. This pattern proposes a solution for some design decisions that have to be taken in order to implement the State pattern, such as the creation and the control of State objects and also the execution of state transitions. When the object has a complex dynamic behavior, its implementation can also become very complex. The Reflective State pattern implements the control aspects in the meta level, separating them from the functional aspects that are implemented by the Context object and the State objects, located at the base level. This pattern provides a solution that is easier to understand, extend and reuse than the State pattern.

2 Introduction

The State design pattern[GHJV95] is a well known pattern that has been used in various applications[JZ91] [Rub94]. Its purpose is to allow an object to change its behavior when its internal state changes. This is achieved by the Context object through delegation of its state-dependent services to State objects. The implementation guidelines of this pattern discuss some design decisions that should be taken in order to implement the states and to control the transitions. Various patterns have discussed the implementation aspects of the State pattern, and have proposed refinements [DA96], variations[OS96] and extensions[Pal97]. When a class has a complex behavior, the implementation of the control aspects of the State pattern can also become complex.

The Reflection architectural pattern[BMRS+96] provides a mechanism for changing dynamically structure and behavior of software. This pattern divides an application in two levels: a base level and a meta level. The base level defines the application's logic where objects implement the functionalities as defined on its functional requirements. The meta level consists of metaobjects that encapsulate and represent information about the base-level objects. The metaobjects can perform management actions that dynamically interfere with the current computations of the corresponding base-level objects. The relationship among the base-level objects and the meta-level objects is specified by means of a metaobject protocol (MOP). Generally speaking, a metaobject protocol establishes the following interactions [Lis98]: (1) attachment of base-level and meta-level objects, that can be static or dynamic, and in one-to-one or many metaobjects to one base-level object basis; (2) reification, which means materialization of information otherwise hidden from the program, such as incoming and outcoming messages, arguments, data and other structural information; (3) execution, which consists of meta-level computation that interferes in the base-level behavior transparently through the interception and reification mechanisms; (4) modification, which is the capability of the metaobjects of changing behavior and structure base-level aspects. The Reflection architectural pattern has been previously used aiming the separation of concerns, separating the non-functional requirements, such as fault tolerance[BRL97] [Lis98] and distribution [OB98][Buz94], from the functional requirements.

[1] Copyright 1998, Luciane Lamour Ferreira.
Permission is granted to copy for the PLoP-98 conference.

In this work we present the Reflective State pattern which uses the Reflection architectural pattern to separate the State pattern in two levels, the meta level and the base level, implementing the control of states and transitions by means of metaobjects. The Reflective State is a generic pattern that intends to solve the same problems of the State pattern in the same context. However, this pattern also implements complex dynamic behavior of an object transparently, separating its control aspects from the functional aspects.

3 The Reflective State Pattern

Intent

To separate the control aspects related to states and their transitions from the functional aspects of the State pattern. These control aspects are implemented in the meta level by means of the metaobjects which represent the state machine's elements.

Motivation

Consider the same motivation example of the State pattern[GHJV95]: a class TCPConnection that represents a network connection. For simplicity, we restrict the TCPConnection to having two basic states: established and closed. Depending on its current state, it can respond differently to the client's requests. For example, the implementation of an *open()* request depends on whether a connection is in its closed state or established state. Moreover, the TCPConnection object can change its current state when an event occurs or when a condition is satisfied. In this example, the same request *open()* also represents an event that causes a transition to the established state. There is also a guard-condition associated with this transition, meaning that the transition will only be triggered if the TCPConnection object has received an acknowledge. A transition can also have an action associated with it, that is performed in the moment the transition is triggered. In our example, when a TCPConnection changes to the established or closed state, it can display a message. So, a request can represent a service that has a state-specific implementation and an event that causes a transition to the next state. This dynamic behavior is specified by the state diagram of Figure 1.

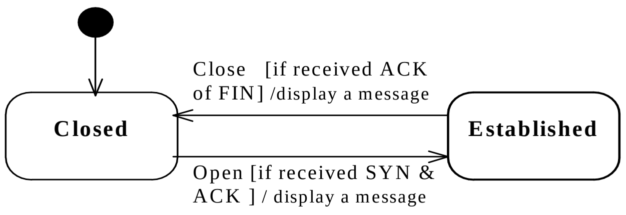
Close [if received ACK of FIN] /display a message
Closed
Established
Open [if received SYN & ACK] / display a message

Figure 1: State diagram of the class TCPConnection using the UML notation.

Context

The behavior of an object depends on its internal state, so the implementation of its services can be different for each possible state. Furthermore, an object can have a complex dynamic behavior specified by a state diagram or a statechart[Har87]. The state diagram is composed by a triple (states, events/guard-conditions, state transitions). The state transition function depends on the current state and the input event and/or a guard-condition. The statechart extends the state diagram with the notion of hierarchy, concurrence and communication.

There are several contexts where the pattern can be applied, for instance, in reactive systems that receive events (outside stimuli) and respond to them by changing their state and, consequently their behavior. Other examples of use can be in the context of distributed systems, control systems, graphical user's interface systems, fault-tolerant systems, etc. These systems can have classes with a complex dynamic behavior specified by a complex and large statechart. Ideally, the design and implementation of the corresponding state machine using object-oriented approach should be made in a structured manner, representing the states and their transitions as much explicit as possible, to keep the complexity of the system under control.

Problem

We can use the State pattern to localizes state-specific behavior in State subclasses, which implement the state-dependent services. The Context object delegates the execution of its services to the current State object. However, the implementation of the State pattern deals with design decisions related to the control aspects of the statechart. These decisions are summarized in the following questions:
(1) Where should the definition and initialization of the possible State objects be located?
(2) How and where should the input events and guard-conditions be verified?
(3) How and where should the execution of state transitions be implemented?

When the object has a complex behavior, the implementation of these issues can also become very complex. According to the implementation guidelines of the State pattern, the control aspects can be located either in the Context object or in the State objects. In the first approach, the Context object is responsible for creating and initializing the State objects, maintaining the reference to the current State object, and performing the transition to the next state. Moreover, it is also responsible for delegating the state-dependent service execution to the current State object. In the second approach, the State objects have the knowledge of their possible next states, and have the responsibilities for handling the events or conditions that causes transitions. Both approaches have some disadvantages:

- Centralizing the control aspects in the Context object can make its implementation very complex since its functional aspects are implemented together with the control aspects. Moreover, it makes the Context class high coupled with the State classes, which makes it difficult to reuse and extend them.
- Decentralizing the responsibilities of the transition logic, by allowing State subclasses themselves specify their successor states, can make them high coupled, introducing implementation dependencies between State subclasses. It also prevents the State class hierarchy from being extended easily.

The following forces are related with these implementation problems:

- Ideally, the implementation of the control aspects of the State pattern should be separated from the functional aspects implemented by the Context object and the State objects. These control aspects should be implemented transparently, ideally in a non-intrusive manner, so that they do not complicate the design.
- The Context and State class hierarchies should be loosely coupled, to facilitate their reutilization and extension.
- The State subclasses should also be independent, with no implementation dependencies between them, so that the addition of a new subclass does not affect other subclasses.

Solution

To solve the problems related with the implementation aspects of the State pattern we propose the use of the Reflection architectural pattern [BMRS+96]. The meta-level classes are responsible for implementing the control aspects of the state machine, separating them from the functional aspects that are implemented by the Context class and the State classes at the base level. In the meta level, the elements of the state diagram (states and transitions) are represented by the MetaState and the MetaTransition class hierarchies. The state machine's controller is represented by the MetaController class. The interception and materialization mechanisms provided by the metaobjects protocol make the execution of the control aspects transparent, oblivious to the base-level objects. Figure 2 shows the class diagram of the Reflective State pattern.

Structure

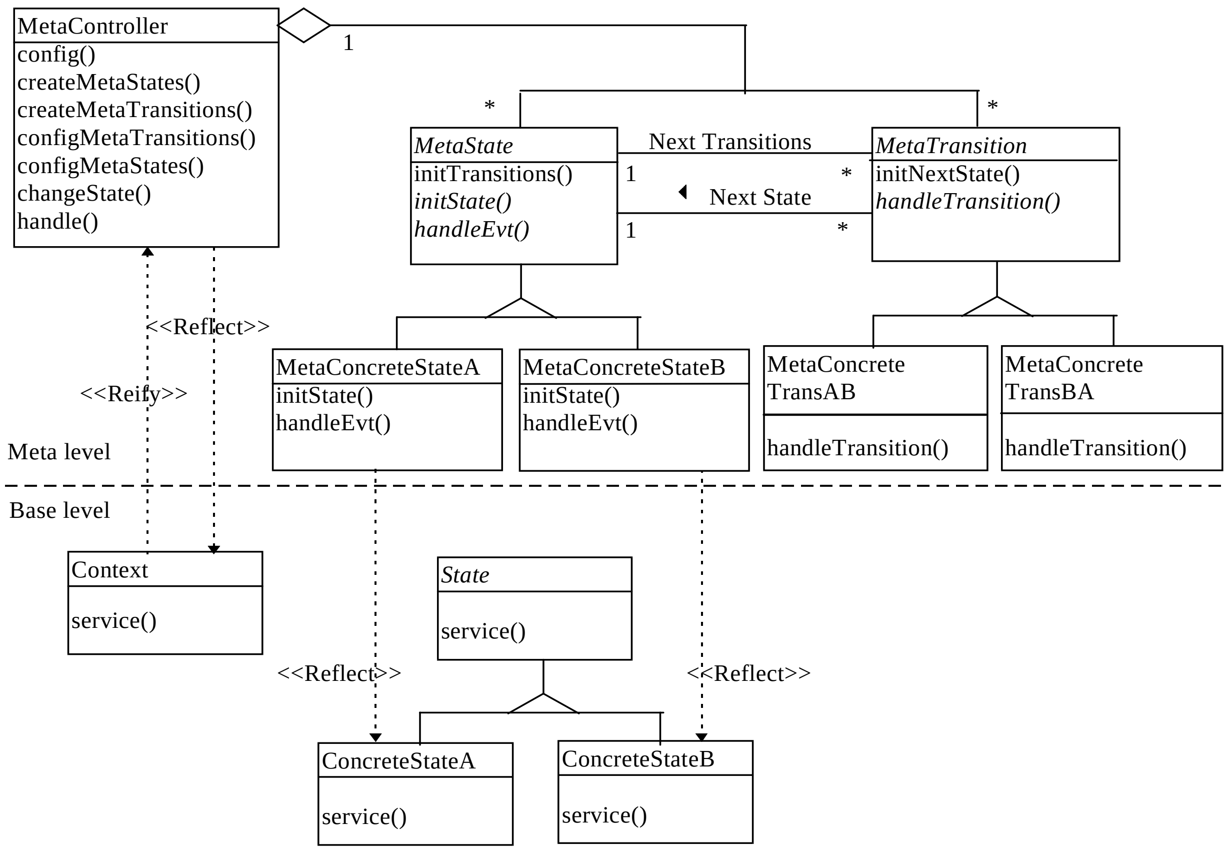

Figure 2: Class diagram of the Reflective State pattern using the UML notation.

To illustrate our solution, we can design the TCPConnection example using the Reflective State pattern structure. Figure 3 shows the object diagram for an instance of the TCPConnection class, with its respective State objects and metaobjects which implement the TCPConnection's statechart of Figure 1.

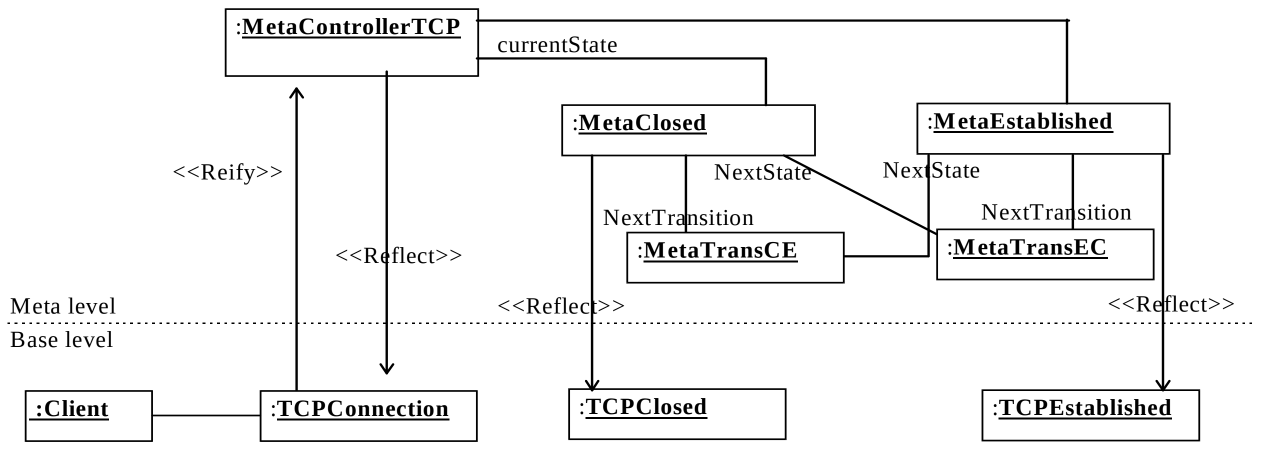

Figure 3: An object diagram for a TCPConnection instance, applying the Reflective State pattern.

The states of the TCPConnection are represented by the TCPEstablished object and the TCPClosed object at the base level, which implement the state-dependent services. The MetaEstablished and MetaClosed metaobjects are responsible for initializing their corresponding State objects, and controlling the execution of the state-dependent service. The MetaTransition metaobjects represent the transitions of the statechart and they are: MetaTransEC (established-to-closed transition) and MetaTransCE (closed-to-established transition). Each MetaTransition has information about the transition function (the event, the guard-conditions and the exit action) that should be verified unless a transition is triggered. The MetaControllerTCP metaobject maintains a reference to the current MetaState metaobject, and changes it when a MetaTransition signals that a transition has occurred. The MetaControllerTCP is responsible for intercepting and materializing all client's service requests targeted to the TCPConnection object.

Participants

The responsibilities of the pattern classes are presented using the CRC Cards notation.

- MetaController

Class MetaController	**Collaborators**
• Configures the meta level, instantiating and initializing the concrete MetaState and MetaTransition subclasses, according to the state diagram specification. • Intercepts all messages sent to the Context object. • Maintains the reference to the MetaState metaobject that represents the current state, and delegates to it the handling of the intercepted messages. • Performs the state transition, changing the reference to a new current MetaState metaobject, that is passed by a MetaTransition object.	In the metalevel: • MetaState • MetaTransition In the base level: • Context

- MetaState

Class MetaState	**Collaborators**
• Defines an interface for handling an event that represents a state-dependent service. • Defines an interface for initializing the State object at the base level. • Defines a method that initializes itself with a list of MetaTransition references that represent the transitions that can exit from this state.	In the meta level • MetaTransition

- MetaConcreteState subclasses

Class MetaConcreteState	**Collaborators**
• Handles all events delegated to it by the MetaController. • Creates and initializes the corresponding State object at the base level, and delegates to it the execution of the state-dependent services. • Broadcasts each event to the MetaTransition metaobjects so that they can verify if the event causes a transition. • Receives the result of the service execution from the state object at the base level, and can also handle the result, if necessary. • Returns the result of the service execution to the MetaController.	In the meta level • MetaConcreteTrans subclasses In the base level • ConcreteState subclasses

- MetaTransition

Class MetaTransition	**Collaborators**
• Defines an interface to handle transitions. • Defines a method that initializes itself with a reference to a MetaState metaobject that represents the next state to be activated when the transition is triggered.	In the meta level • MetaState

- MetaConcreteTrans subclasses

Class MetaConcreteTrans	**Collaborators**
• Has all information that defines a transition function, i.e., the current state, the event/guard-condition and the next state. • Verifies if an event causes a transition and/or if a guard-condition is satisfied. • If the transition is triggered, it requests the Metacontroller metaobject to change its current state, passing to it the reference to the next MetaState.	In the meta level • MetaConcreteState subclasses • MetaController

- Context class

Class Context	**Collaborators**
• Defines the service interface of interest to clients, as defined in its functional requirements.	In the meta level • MetaController

- State class

Class State	**Collaborators**
• Defines an interface for encapsulating the behavior associated with a particular state of Context (as defined in the State pattern)	

- ConcreteState subclasses

Class ConcreteState	**Collaborators**
• Each subclass implements a behavior associated with a state of the Context(as defined in the State pattern).	In the meta level • MetaConcreteState subclasses

Collaborations

The metaobjects represent a direct mapping of the state diagram elements. The configuration of the meta level consists of: instantiation of each concrete subclass of the MetaState class and MetaTransition class; initialization of the MetaState metaobjects with their corresponding MetaTransitions metaobjects; initialization of each MetaTransition metaobject with its corresponding next MetaState metaobject. The MetaController metaobject is responsible for implementing all these configurations according to the state diagram's specification of a Context class.

The metaobjects are responsible for controlling the execution of state-dependent services and the state transitions (Figure 4). The interactions between the metaobjects and the base-level objects are performed by means of a metaobject protocol (MOP). The MOP's kernel should implement the interception and materialization mechanism so that the metaobjects can perform the extra computation related to the execution of the state machine. There are several different MOPs' implementations, thus we do not show a specific MOP's interaction; instead we make the assumption that a MOP's implementation performs all communications between the base and meta levels transparently, and materializes the operation with its basic informations. The materialized operation can be represented as an object, which should be passed as a parameter of the *handle()* method. In Figure 4, the dotted line represents a MOP's implementation. After the operation is materialized, its handling is delegated to the MetaController metaobject which initializes the handling of the service request.

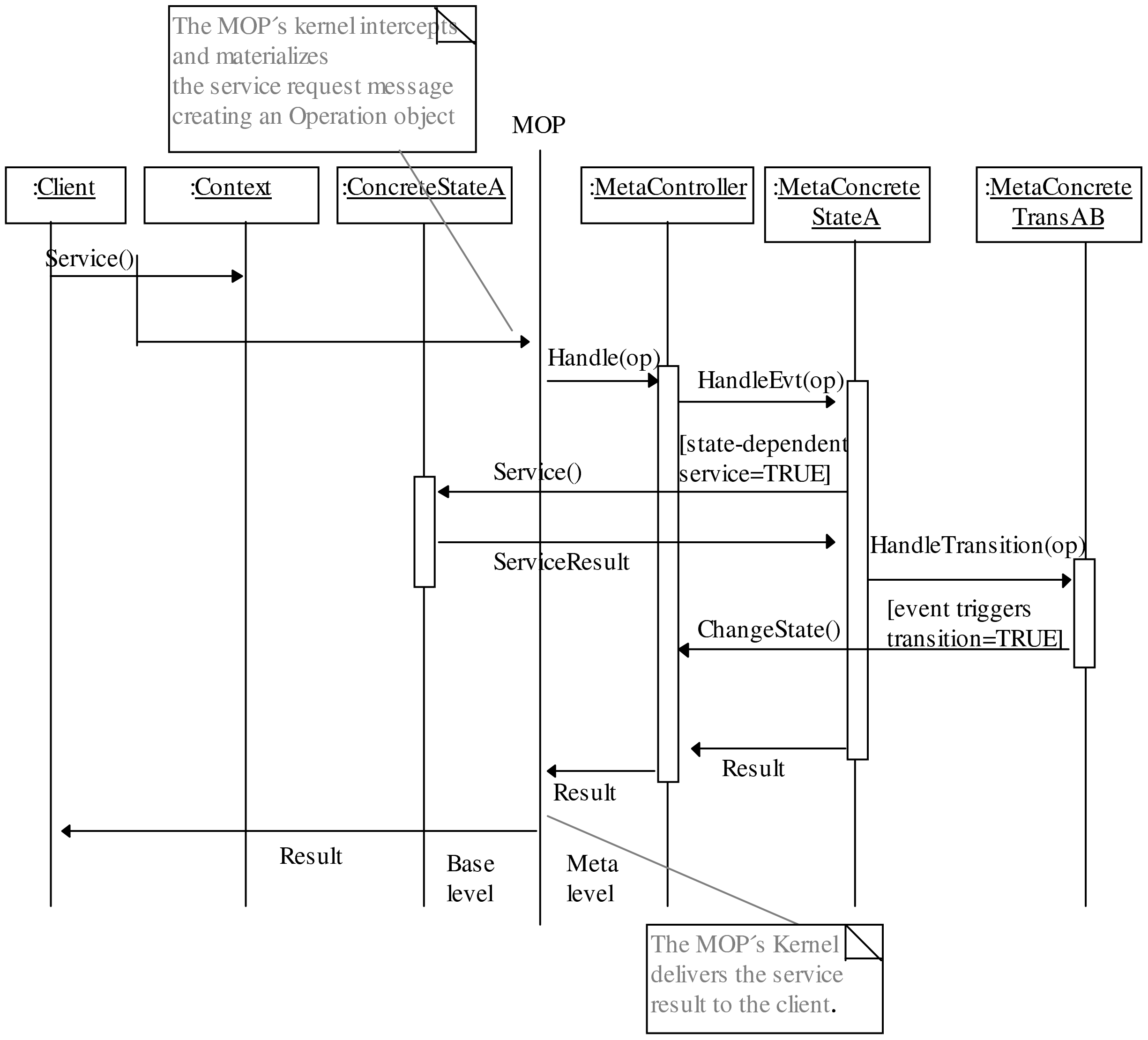
The MOP´s kernel intercepts and materializes the service request message creating an Operation object
MOP
:Client
:Context
:ConcreteStateA
:MetaController
:MetaConcrete StateA
:MetaConcrete TransAB
Service()
Handle(op)
HandleEvt(op)
[state-dependent service=TRUE]
Service()
ServiceResult
HandleTransition(op)
[event triggers transition=TRUE]
ChangeState()
Result
Result
Result
Base level
Meta level
The MOP´s Kernel delivers the service result to the client.

Figure 4: Interaction diagram for the Reflexive State Pattern

The MetaController metaobject intercepts the service request targeted to the Context object and delegates its handling to the current MetaConcreteState metaobject. The current MetaConcreteState metaobject verifies if the request corresponds to a state-dependent service. If so, the current MetaConcreteState metaobject delegates the service execution to its corresponding ConcreteState object at the base level. It also delegates the event's handling to the MetaConcreteTrans metaobject so that it can decide whether the message corresponds to an event that causes a transition or not. A MetaConcreteState can have a list of MetaTransitions, and it can delegate the handling of the transition sequentially or concurrently. If a MetaConcreteTrans metaobject verifies that its transition should be triggered, it requests the MetaController metaobject to change its current State, passing to it the reference to the next MetaConcreteState metaobject. After the service request has been handled, the service result is first returned to the MetaController metaobject, and then to the MOP's Kernel which delivers the result to the client.

Consequences

- The Reflective State pattern localizes state-specific behaviors and partitions behavior for different states, as in the case of the State pattern. The state objects make implementation of the state-dependent services more explicit, and consequently, the design becomes more structured and easier to understand, maintain and reuse.
- The Reflective State provides a solution for implementing the control aspects of the State pattern, separating them from the functional aspects implemented by the Context object and State objects. This characteristic is provided by the Reflection architectural pattern. This solution makes the implementation of the dynamic behavior of a class (that might be specified by a complex state diagram) more explicit, also making the design more structured, keeping the complexity of the system under control.
- The State and Context class hierarchies are independent and they can be designed to be highly coherent and loosely coupled, facilitating the adaptability of the system to the changes of requirements, its reuse and extension.
- The Reflective State pattern has some limitations related to the use of the Reflective architecture. In general, a Reflective architecture increases the number of indirections in the execution of a method, causing an impact in the system's performance.

Implementation

The Reflective State pattern proposes some criteria for the implementation decisions that have to be taken in order to implement the State pattern. It also adds other implementation decisions, mainly related to the use of reflective programming.

(1) *Where should the definition and initialization of the possible State objects be located?* The State objects are defined and instantiated by their respective MetaState metaobjects. The State objects' creation can be implemented using the Factory Method pattern. The MetaState hierarchy at the meta-level corresponds to the State hierarchy at the base level. The abstract MetaState class defines an interface for creating State objects, the abstract method *initState()*, that should be overridden in the MetaState subclasses, so that the concrete State objects can be created. This solution makes the extension of the State hierarchy easier: if a concrete State class is added, a MetaState class should also be added so that it can instantiate this State object. The MetaController can be reconfigured dynamically, adding a new MetaState object at runtime. A new MetaTransition metaobject that makes references to the new MetaState metaobject can also be added in the same manner. If dynamic reconfiguration is desired, the interface of the MetaController should define methods for adding MetaStates and MetaTransitions. Some MOP can also implement the reconfiguration mechanism in a more transparent manner.

Also, the State objects can be implemented as the Singleton pattern, so that many MetaState metaobjects of different meta state machines can share the same base-level State objects. As in the State pattern, the methods defined in the State objects should also receive a reference to a Context object as parameter, in order to obtain independence between the State objects and the Context object.

(2) *How and where should the input events and guard-conditions defined in the state machine be verified*? The Reflective State pattern establishes a well-defined criterion for this issue. The events of the state machine correspond to the method calls or field accesses targeted to the Context object, that are materialized and presented to the metaobjects. The MetaTransition metaobject is responsible for verifying the input event (the name of the method, the arguments, the result type) and the guard-conditions associated with the transition. The guard-conditions are

boolean expressions that associate incoming arguments, attribute's values or methods of the Context object. If the transition is verified, the MetaTransition metaobject should request the MetaController metaobject to change its current state, passing a reference to next MetaState as a parameter.

(3) *Where should the configuration of the meta level be performed?* The MetaController class interface defines some methods which perform the configuration of the state machine at meta level. The *CreateMetaStates()* and *CreateMetaTransitions()* methods should instantiate each metaobject according to the states and transitions of the Context class' statechart. The *ConfigMetaStates()* method should configure each MetaState metaobject with its corresponding next Metatransition metaobjects and the *ConfigMetaTransitions()* method should configure each MetaTransition metaobjects with a reference to its next MetaState metaobject. In order to obtain reusability, the MetaController class can be implemented as an Abstract Factory pattern. The MetaController class can define abstract methods to create and configure the metaobjects that implement a specific state machine. The concrete subclasses of the MetaController should override these methods creating and configuring the MetaState and the MetaTransition metaobjects according to a statechart specification for a specific Context class. This solution also makes the extension of the Context class easier: if a Context subclass is defined and the statechart is extended, a new MetaController subclass should also be defined, and it can redefine the configuration methods so that new MetaState and MetaTransition subclasses can be instantiated according to the new statechart specification.

(4) These three implementation guidelines have assumed some MOP characteristics. Thus, it is very important to choose a MOP that implements the desired characteristics and gives a full support to the main reflection mechanisms. The MOP should implement the interception and reification mechanisms, so that the metaobjects can inspect the service request targeted to the Context object at the base level, and also perform the extra computation related to the execution of the state machine. Ideally, the MOP should also provide some base classes that can be derived by the metaobjects, so that they can implement meta level behavior. For instance, the classes of the meta level can be derived from a MetaObject base class, which defines the default behavior of the metaobjects, such as the method *handle()*, called by the MOP's kernel. Other useful base classes may be a class to represent an operation (a service request materialized with its essential informations), or a class to represent the result of an operation that has been performed. The result objects can also be presented to the metaobjects so that they can inspect, modify or replace them.

Known Uses

The implementation of state machines has been widely discussed in the development of reactive systems. These system tend to be very large and complex, and the implementation of the state machine that controls its dynamics is not a trivial task. It has motivated the study of the state machine implementation based on the object-oriented approach [SM92] implementing the states and transitions more explicitly. This approach has also been discussed in many related design patterns[DA96] [OS96] [Ran95] [Pal97]. The use of the Reflection architectural pattern to implement the state machine in the meta-level has been discussed in [Buz94] [Cha96]. These works also implement the control of the transitions in the meta level, and define the State objects and the Context object at the base level.

We are applying the Reflective State pattern in the development of a framework for the environmental fault-tolerant train controller's domain. An environmental fault-tolerant train controller system implements some components that represent environmental entities that may be faulty, such as sensors and switches. These components, in the solution domain, should reflect the normal and abnormal behavior phase of the environmental entities. C.M.F.Rubira [Rub94] proposes a solution for the design of environmental fault-tolerant components using the State pattern, implementing the normal and abnormal behavior phases by means of State objects. In the framework's implementation, we should also consider some requirements such as extensibility to specific application requirements. The extensibility requirement has been the motivation for our study of the State pattern design decisions. Using the Reflective State pattern, the fault-tolerant component hierarchy (related to the Context hierarchy in the general structure) and the State hierarchy become more independent, and consequently, easier to extend and reuse. Also, the execution of the transitions and state-dependent services become more explicit, making the design easier to understand, which are essential features of a framework.

We are using the Reflective State pattern to implement the environmental fault-tolerant components of the framework; in the future, we intend to implement other fault-tolerance requirements, such as software and hardware fault tolerance. We have defined a System of Patterns for the fault-tolerant domain [FR98] which has the Reflective State pattern as the most generic pattern. Other patterns of the System derive from the Reflective State pattern structure, adding fault-tolerance semantics.

To implement the framework, we are using the MOP of Guaraná [OGB98], which emphasizes flexibility, reconfigurability, security and meta-level code reuse. We are using a free Java-based implementation of the Guaraná reflective architecture that is currently available, and has been developed in the Institute of Computing of the State University of Campinas.

Related patterns

In the literature, there are some recent patterns that have discussed the implementation problems related to the State pattern. The work of Dyson and Anderson [DA96] presents a state pattern language which classifies the State pattern into seven related patterns that refine and extend this pattern. The pattern language also discusses the implementation aspects of the state transition control and the initialization of the State objects. However, these patterns do not separate the implementation of the transition control aspects from the functional aspects.

The work of Odrowski and Sogaard [OS96] defines some variations of the State pattern that solve implementation problems related to objects' state and the dependency between states of related objects. This work shows solutions for the problem of combining the State pattern with other patterns, however it does not discuss the problem of transition control.

Alexander Ran's work [Ran95] presents a family of design patterns that can be used to cope with the implementation of complex, state-dependent representation and behavior. This pattern family is presented in the form of a design decision tree (DDT), that separates state-behavior in State classes, and implements the control of transitions and guard-conditions explicitly, using transition methods and predicative classes, respectively. The Reflective State pattern proposes the separation of these control aspects by means of the MetaTransition metaobjects that encapsulate the informations about the transition functions.

The paper by Günther Palfinger [Pal97] presents an extension of the State pattern, defining a State Mapper object which maps events to actions, using a list of event/action pairs. The list can be added/modified/deleted at runtime, providing easy adaptation to new requirements dynamically. In a similar manner, the Reflective State pattern can also be adapted dynamically using the metaobject protocol to implement changes of the state machine, such as addition of new states and transitions.

Sample Code

We exemplify the implementation of the TCPConnection class using Guaraná's MOP[OGB98]. It defines a base class, called MetaObject, that encapsulates the meta-level behavior, providing all interface and essential implementation for a metaobject so that it can handle operations, results, etc. The metaobject protocol of Guaraná establishes that an object can be directly associated with either zero or one metaobject, and this association is dynamic. The kernel of Guaraná implements a method *reconfigure()*

that associates an object with a metaobject, and can also replace an old metaobject with a new one, allowing dynamic meta reconfiguration. The MOP of Guaraná also defines a more specialized kind of metaobject, the Composer, which groups metaobjects that are commonly used together and delegates the operation's handling to them. The Composers allow many metaobjects to be indirectly associated with an object. The metaobjects that are directly and indirectly associated with an object form its meta configuration. A more specialized kind of Composer is the SequentialComposer that delegates the operation's handling sequentially.

The following example present a partial Java code for the TCPConnection example, using the Guaraná's MOP [OGB98]. The metaobject classes that implement the meta-level state machine are derived from the following base classes of Guaraná: MetaObject, Composer and SequentialComposer.

Meta-level classes

MetaController class: The MetaController class is derived from the Composer class of Guaraná, since the MetaController groups the metaobjects that implement the meta state machine, and delegates the operation's handling to them. In fact, the MetaController metaobject delegates only to the MetaState objects which are also Composers (SequentialComposers) that delegate to the MetaTransitions metaobjects.

```
import BR.unicamp.Guarana.*;

public abstract class MetaController extends Composer{
   protected MetaObject[] metaStatesArray;
   protected MetaState currentMetaState;

   protected abstract void createMetaStates();
   protected abstract void createMetaTransitions();
   protected abstract void configMetaStates();
   protected abstract void configMetaTransitions();

   public final void config(){
            createMetaTransitions();
            createMetaStates();
            configMetaTransitions();
            configMetaStates();
   }

   public void changeState(MetaState nextState){
            currentMetaState = nextState;
   }

   public Result handle(Operation operation, Object object){
            if (operation.isConstructorInvocation())
            // "null return"  means that the metaobjects do not handle constructor invocation
               return null;
            return currentMetaState.handle(operation,object);
   }
}
```

MetaControllerTCP class:

```
import BR.unicamp.Guarana.*;

public class MetaControllerTCP extends MetaController{
   protected  MetaEstablished metaEstablished;
   protected  MetaClosed metaClosed;
   protected  MetaTransEC metaTransEC;
   protected  MetaTransCE metaTransCE;
   protected void  createMetaStates(){
            metaEstablished = new MetaEstablished();
            metaClosed = new MetaClosed();
            currentMetaState = metaClosed;          //initializes with a default state.
   }

   protected void createMetaTransitions(){
            metaTransCE = new MetaTransCE(this);
            metaTransEC = new MetaTransEC(this);
   }

   protected void configMetaTransitions(){
            //Configures the metaTransitions with its next MetaStates
            metaTransCE.initProxState(metaEstablished);
            metaTransEC.initProxState(metaClosed);
   }

   protected void configMetaStates(){
            //Configures the MetaState metaobjects with the array of next MetaTransitions metaobject
            metaEstablished.initTransitions(new MetaTransition[]{metaTransEC});
            metaClosed.initTransitions(new MetaTransition[]{metaTransCE});

            //initiliazing the array of MetaStates that the MetaController delegates to.
            metaStatesArray = new MetaState[]{metaClosed,metaEstablished};
   }
}
```

MetaState abstract class:

```
import BR.unicamp.Guarana.*;
import java.lang.reflect.*;

public abstract class MetaState extends SequentialComposer{
    protected State stateObject;

   protected abstract void  initState(Object object);

   public void initTransitions(MetaTransition[] arrayNextTransitions){
            //calls the method in the SequentialComposer base class.
            super.setMetaObjectsArray(arrayNextTransitions);
   }
}
```

MetaEstablished concrete class

```
import BR.unicamp.Guarana.*;

public class MetaEstablished extends MetaState{
   public void initState(Object object){
            stateObject = new TCPEstablished();
   }

   public Result handle(Operation operation,Object object){
            //Verifies if an operation is a state dependent service.
            String name = operation.getMethod().getName();
            Class[] parameters = operation.getMethod().getParameterTypes();
            //it can modify the parameter array if the state method defines another parameter, as a TCPConnection reference.
            //....
            Result res = null;
            if (stateObject == null) initState(object);          //it's initialized only if it's necessary.
            if (operation.isMethodInvocation()){
                Object resultObj;
                try {
                        //returns a public method of the class.
                        Method methodEx = stateObject.getClass().getMethod(name,parameters);
                        Object[]arguments = operation.getArguments();
                        resultObj = methodEx.invoke(stateObject,arguments);
                        if (resultObj == null){
                            res = Result.returnVoid(operation);
                        }
                        else {
                            res = Result.returnObject(resultObj,operation);
                        }
                }
                catch (IllegalAccessException e1){
                        //do some exception handling
                }
                catch (NoSuchMethodException e2){}
                catch (InvocationTargetException e3){}
            }
            //Delegates the operation's handling to the MetaTransition metaobjects sequentially,
            // calling the handle() method of the SequentialComposer
            super.handle(operation,object);

            //can do some handling with the result, unless it is returned
            //......
            return res;
    }
}
```

MetaClosed concrete class

```
import BR.unicamp.Guarana.*;

public class MetaClosed extends MetaState{
    public void initState(Object object){
            stateObject = new TCPClosed();
    }

    public Result handle(Operation operation,Object object){
            //Verify if an operation is a state dependent service, like the MetaEstablished class do.
            // It can inspect the result and do some handling.
     }
}
```

MetaTransition abstract class

```
import BR.unicamp.Guarana.*;
import java.lang.reflect.Method;

public abstract class MetaTransition extends MetaObject{
    protected MetaController metaController;
    protected MetaState nextState;
    public MetaTransition(){}

    public MetaTransition(MetaController metaController){
            this.metaController = metaController;
    }

    public void initProxState(MetaState nextState){
            this.nextState = nextState;
    }
}
```

MetaTransEC concrete class

```
import BR.unicamp.Guarana.*;
import java.lang.reflect.Method;

public class MetaTransEC  extends MetaTransition{
   public MetaTransEC(MetaController metaController){
            super(metaController);
   }

   public Result handle(Operation operation,Object object){
            //define the transition function.
            String eventName = "close";
            protected int paramNum = 0;
            if (operation.isMethodInvocation()){
                        Method opMethod = operation.getMethod();
                        if ((eventName.equals(opMethod.getName())) &&
                          (opMethod.getParameterTypes().length == paramNum)){
                                    //the event is correct. It can also test some guard-conditions using the “object” parameter
                                    //....
                                    metaController.changeState(nextState);            //if the event and guard-conditions are verified
                        }
            }

            return null;
   }
}
```

MetaTransCE concrete class

```
import BR.unicamp.Guarana.*;
import java.lang.reflect.Method;

public class MetaTransCE  extends MetaTransition{
   public MetaTransCE(MetaController metaController){
            super(metaController);
   }

    public Result handle(Operation operation,Object object){
            //defines the transition function
            //verifies the transition testing the event name and arguments of the operation, and the guard-conditions.
            //...
   }
}
```

Base-level classes

The TCPConnection class and its respective State classes implement only their functional requirements, without any information about the execution control of the state machine.

TCPConnection class: The state-dependent methods do not have any implementation. Optionally, they can present some default behavior that can be executed if a TCPConnection object has not been associated with a MetaControllerTCP metaobject.

```
public class TCPConnection{
   public TCPConnection(){}

   public void open(){
            //some default behavior;
   }
   public void close(){}

   //other methods and attributes
}
```

TCPState class

```
public abstract class TCPState{
   //If there are some state attributes, defines them here

   public abstract close(TCPConnection);
   public abstract open(TCPConnection);
}
```

TCPEstablished class

```
public class TCPEstablished extends TCPState{
   public close(TCPConnection tcpCon){
            //closes the connection
   }

   public open(TCPConnection tcpCon){
            //does nothing, because the Connection is already open.
   }
}
```

TCPClosed class: The implementation is similar to the Established class.

TCPApplication class: This class represents the application class which implements the *main()* method. First, the *main()* method creates a MetaControllerTCP metaobject and calls the method *config()* that configures the MetaControllerTCP. Then, it creates a TCPConnection object and call the method *reconfigure()* of Guaraná class (that implements the Guaraná's Kernel). The *reconfigure()* method expects three parameters: (1) a reference to the object to be reconfigured, in this case, the TCPConnection object; (2) a reference to an oldMetaObject, if the object has been already configured with another one, and in this case it is null; (3) a reference to a newMetaObject, which the object is being reconfigured with, in this case, the MetaControllerTPC metaobject.

```
public class TCPApplication{
   public static void main(String[] argv){
            MetaControllerTCP metaControllerTCP = new MetaControllerTCP();
            metaControllerTCP.config();
            TCPConnection aTCPConnection = new TCPConnection();
            Guarana.reconfigure( aTCPConnection, null, metaControllerTCP);
   }
}
```

4 Acknowledgments

We would like to thank our shepherd Dr. Michel de Champlain for valuable comments and suggestions for improvement of the pattern.

This work is partially supported by FAPESP (*Fundação de Amparo à Pesquisa do Estado de São Paulo*), grant 97/11060-0 for Luciane Lamour Ferreira, grant 96/1532-9 for LSD-IC-UNICAMP (*Laboratório de Sistemas Distribuídos, Instituto de Computação, Universidade Estadual de Campinas*); and by CNPq (*Conselho Nacional de Desenvolvimento Científico e Tecnológico*), grant 131962/97-3.

5 References

[BMRS+96] F. Buschmann, R. Meunier, H Rohnert, P. Sommerlad, M. Stal. *A System of Patterns*: *pattern-oriented software architecture.* John Wiley & Sons, 1996.

[BRL97] L.E.Buzato, C.M.F.Rubira and M.L.Lisboa. A Reflective Object-Oriented Architecture for Developing Fault-Tolerant Software. *Journal of the Brazilian Computer Society,* 4(2):39-48, November, 1997.

[Buz94] L.E.Buzato. *Management of Object-Oriented Action-Based Distributed Programs*. Ph.D. Thesis, University of Newcastle upon Tyne, Department of Computer Science, December 1994.

[Cha96] M. de Champlain. A Design Pattern for the Meta Finite-State Machines. *Proceedings of the Circuits, Systems and Computers Conference (CSC'96),* Hellenic Naval Academy, Piraeus, Greece, June 1996.

[DA96] P.Dyson and B. Anderson. State Patterns. *Pattern Languages of Program Design 3,* Addison-Wesley, 1997. Eds. R.Martin, D.Riehle, F.Buschmann.

[FR98] L.L.Ferreira and C.M.F.Rubira. Integration of Fault Tolerance Techniques: a System of Pattern to Cope with Hardware, Software and Environmental Fault Tolerance. 28th International Symposium on Fault-Tolerant Computing(FastAbstract), june 1998.

[GHJV95] E.Gama, R. Helm, R Johnson e J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley Publishing, 1995.

[Har87] D.Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming,* 8: 231-274, North-Holland, 1987.

[JZ91] R.E.Johnson and J. Zweig. Delegation in C++. *Journal of Object-Oriented Programming*, 4(11):22-35, November 1991.

[Lis98] M.L.B.Lisboa. A New Trend on the Development of Fault-Tolerant Applications: Software Meta-Level Architectures. *Proceedings of the 1998 IFIP - International Workshop on Dependable Computing and its Applications,* Johannesburg, South Africa, January, 1998.

[OB98] A.Oliva, L.E.Buzato. An Overview of MOLDS: A Meta-Object Library for Distributed Systems. *Technical Report IC-98-15, Institute of Computing, State University of Campinas,* April 1998.

[OGB98] A. Oliva, I.C.Garcia, and L.E.Buzato. The reflexive architecture of Guaraná. *Technical Report IC-98-14, Institute of Computing, State University of Campinas*, April 1998.

[OS96] J. Odrowski and P. Sogaard. Pattern Integration - Variations of State. *PLoP'96 Writer's Workshop.* (http://www.cs.wustl.edu/~schmidt/PLoP-96/Worshops.html)

[Pal97] G.Palfinger. State Action Mapper. *PLoP'97 Writer's Workshop* (http://st-www.cs.uiuc.edu/hanmer/PLoP-97/Workshops.html).

[Ran95] A. Ran. MOODS: Models for Object-Oriented Design of State. *Pattern Languages of Program Design 2,* Addison-Wesley, 1996. Eds.J.M.Vlissides,J.O.Couplien e N.L. Kerth.

[Rub94] C.M.F. Rubira. *Structuring Fault-Tolerant Object-Oriented Systems Using Inheritance and Delegation*. Ph.D. Thesis, Dept. of Computing Science, University of Newcastle upon Tyne, October 1994.

[SM92] S.Shlaer and S.J.Mellor. *Object Lifecycles: Modeling the World in States*. Prentice-Hall, New Jersey, 1992.