Journal of Software Engineering

ISSN 1819-4311

Journal of Software Engineering 4 (3): 215-230, 2010
ISSN 1819-4311

Using Java Enums to Implement Concurrent-Hierarchical State Machines

Jauhar Ali
College of Engineering and Computer Science,
Abu Dhabi University, Abu Dhabi, UAE

Abstract: The aim of this study is to find an easy way for implementing concurrent-hierarchical state machines into efficient and encapsulated Java code. State Machine is one of the important diagrams in Unified Modeling Language (UML). In UML, State Machine is used to represent reactive behavior of a class of objects. Implementing a state machine has been difficult for programmers because the commonly used Object-Oriented Programming languages do not provide any explicit support. We present a new approach to implement state machines using Java Enums. In our approach, each state is represented as an enum-value. Hierarchical states and concurrent orthogonal regions within state machines are implemented by linking the related enum-values to each other. Our approach offers several benefits. First, using Java enums makes the resulting code compact, efficient and easy to understand. Second, the structure of the state machine is obvious in the implementation code. Third, the whole state machine's behavior is encapsulated within a single class (called StateMachine). The proposed code can serve as a Java implementation pattern for state machines.

Key words: State machine diagram, UML, object-oriented programming, java

INTRODUCTION

Unified Modeling Language (UML) (Booch *et al.*, 2005; Rumbaugh *et al.*, 2010; OMG, 2010) is a standardized modeling language for designing software systems. State Machine is an important UML diagram that is used to represent the behavior of a class of objects in response to events or messages received from other objects. UML State Machine Diagram is a variation of Harel's statechart (Harel, 1987) that incorporates hierarchical states (OR-substates) and concurrent states (AND-substates) into the traditional state transition diagram.

Implementing state machines is difficult for most programmers due to the lack of support from object-oriented programming languages. Various approaches exist to implement state machines at the application programming level. As discussed in the Related work and Discussion sections below, these approaches have several problems. Most of these approaches do not handle concurrent-hierarchical states. Some approaches introduce too many additional classes, requiring creating a new object whenever the current state changes. This makes the result code very inefficient.

In this study, we present a new approach to efficiently implement state machines having hierarchical and concurrent states. Our approach encapsulates the state machine behavior within the owner class and keeps the structure of the state machine obvious at the programming level.

SIMPLE STATES

To demonstrate our approach, we use the example of a simple Air-Condition (AC) Controller. Figure 1 shows the behavior of the AC controller. The controller will be in one of three states: Off (default), FanOnly and AC. If Off is the current state and the PowerBut event occurs, the state will change to FanOnly. Similarly, if FanOnly or AC is the current state and the PowerBut event occurs, the stopFan action will execute and the state will change to Off. The ACBut event will change the state from FanOnly to AC and vice versa. Whenever the AC state is entered, the startCondenser action will execute. Similarly, whenever the AC state is exited, the stopCondenser action will execute.

To implement the AC Controller's state machine, we use a nested class, called StateMachine, inside the ACController1 class (Listing 1). The StateMachine class encapsulates almost all aspects of the state machine. All actions in the state machine become methods in the ACController1 class (lines 10- 25). The ACController1 and the StateMachine classes have references to each other (lines 2 and 33), through which they can call each other's methods. All events received by the ACController1 are delegated to the StateMachine (lines 28- 29).

Inside the StateMachine class, we use two Java enums: Event (line 45) and State (line 48). The Event enum represents all events and the State enum represents all states in the state machine. Each event and state becomes an enum value. For example, off (line 49) and FanOnly (line 62) become enum values inside State. The state (line 34) reference inside StateMachine represents the current state of the state machine.

Java allows having methods and data members inside enums (Sun Microsystems, 2010). Each enum value can override the methods. The State enum has empty entry (line 102) and exit methods (line 103) . The AC state (line 79) overrides these methods because it has entry and exit actions in the state machine (Fig. 1). The State enum has also an abstract method, named process (line 101), which is overridden by all states. It is called by the StateMachine on the current state whenever an event is delegated to the StateMachine (lines 41 and 42).

All transitions from a state are implemented in the process method for that state. The process method takes an event as parameter and chooses one case from the switch statement depending on the event. Each case corresponds to one transition. For example, the first case in the process method of the FanOnly state implements the transition on the ACBut event (line 65). Inside each case (which corresponds to a transition), three methods are called in the given order: (1) the exit method of the current state, (2) the action method (if any) for the transaction and (3) the entry method of the new state.

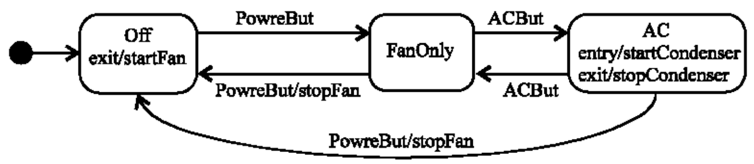

Fig. 1: Simple state machine diagram for air-condition controller

Listing 1: ACController1.java

```
public class ACController1 {
  StateMachine stateMachine;

  ACController1(){
     stateMachine = new StateMachine(this);
     // ... other stuff
  }

  // The action methods
  private void startCondenser() {
     /* To be replaced with appropriate code */
     System.out.println("startCondenser executed");
  }
  private void stopCondenser(){
     /* To be replaced with appropriate code */

     System.out.println("stopCondenser executed");
  }
  private void startFan(){
     /* To be replaced with appropriate code */
     System.out.println("startFan executed");
  }
  private void stopFan(){
     /* To be replaced with appropriate code */
     System.out.println("stopFan executed");
  }

  // Events delegated to StateMachine
  public void powerBut() { stateMachine.powerBut();}
  public void acBut() {stateMachine.acBut();}

  // The StateMachine class
  static class StateMachine {
     ACController1 context;
     State state;

     StateMachine(ACController1 context){
        this.context = context;
        state = State.Off;//default
     }

     private void powerBut(){state.process(this, Event.PowerBut);}
     private void acBut(){state.process(this, Event.ACBut);}

     // All of the events
     enum Event {PowerBut, ACBut}

     // All of the states
     enum State {
        Off {
           void exit(StateMachine sm) {sm.context.startFan();}

           void process(StateMachine sm, Event e){
                 switch(e){
                    case PowerBut:
                       this.exit(sm);
                       sm.state = FanOnly;
                       sm.state.entry(sm);
                    }
           }
        },

        FanOnly {
```

Listing 1: ACController1.java

```
           void process(StateMachine sm,Event e){
                      switch(e){
                         case ACBut:
                            this.exit(sm);
                            sm.state = AC;
                            sm.state.entry(sm);
                            break;
                      case PowerBut:
                            this.exit(sm);
                            sm.context.stopFan();
                            sm.state = Off;
                            sm.state.entry(sm);
                      }
           }
        },
        AC {
           void entry(StateMachine sm) {
               sm.context.startCondenser();}
        void exit(StateMachine sm) {
               sm.context.stopCondenser();}

        void process(StateMachine sm, Event e){
           switch(e){
               case ACBut:
                  this.exit(sm);
                  sm.state = FanOnly;
                  sm.state.entry(sm);
                  break;
               case PowerBut:
                      this.exit(sm);
                      sm.context.stopFan();
                      sm.state = Off;
                      sm.state.entry(sm);
                  }
           }
        };

        abstract void process(StateMachine sm, Event e);
        void entry(StateMachine sm){}
        void exit(StateMachine sm){}
     }// end of enum State
  }// end of class StateMachine
}// end of class ACController1
```

HIERARCHICAL STATES

State machines may have hierarchical states where the substates inherit transitions from its superstate. Figure 2 shows the behavior of the AC controller with a superstate (Running). If the Running state is active, the controller will be either in FanOnly (default) or in AC state. In any of the two substates, if the PowerBut event occurs, the state will change to Off. There are entry and exit actions for the Running state as well.

Listing 2 shows the Java code corresponding to Fig. 2. To implement the state hierarchy, we use the parent reference (line 119) initialized by the constructor (line 120) in the State enum. For example, the FanOnly constructor call (line 88) determines that Running is its parent state. There is always a default case in the process method of any substate, which calls the process method in the parent state (lines 96 and 114). This is how we make the superstate transitions executable in the substates and give priority to the transitions from a substate. To allow exiting from or entering to nested states, we put exitAll and enterState

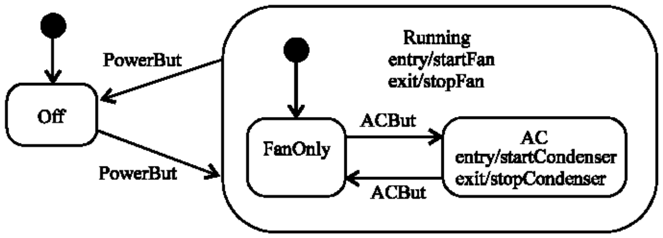

Fig. 2: State machine with hierarchical states

methods in the StateMachine class. They are called from the state's process method when needed. The exitAll method ensures that the exit action for a substate is executed before its superstate's exit action. Similarly, the enterState method ensures that the entry action of a superstate executes before its substate's entry action.

Listing 2: ACController2.java

```
public class ACController2 {
    StateMachine stateMachine;

    ACController2(){
        stateMachine = new StateMachine(this);
        // ... other stuff
    }
    // The action methods
    private void startCondenser(){
        /* To be replaced with appropriate code */
        System.out.println("startCondenser executed");
    }
    private void stopCondenser(){
        /* To be replaced with appropriate code */
        System.out.println("stopCondenser executed");
    }
    private void startFan(){
        /* To be replaced with appropriate code */
        System.out.println("startFan executed");
    }
    private void stopFan(){
        /* To be replaced with appropriate code */
        System.out.println("stopFan executed");
    }

    // Events delegated to stateMachine
    public void powerBut() {stateMachine.powerBut();}
    public void acBut() {stateMachine.acBut();}

    // The StateMachine class
    static class StateMachine {
        ACController2 context;
        State state;

        StateMachine(ACController2 context){
            this.context = context;
            state = State.Off;//default
        }

        private void powerBut(){state.process(this, Event.PowerBut);}
        private void acBut(){state.process(this, Event.ACBut);}

        private void enterState(State... states){
```

Listing 2: ACController2.java

```
        for (State s: states) s.entry(this);
        state = states[states.length-1];
    }

    private void exitAll(State child, State parent){
        State s = child;
        while (true) {
            s.exit(this);
            if (s == parent) break;
            s = s.parent;
        }
    }

    // All of the events
    enum Event {PowerBut, ACBut}

    // All of the states
    enum State {
        Off (null){
            void process(StateMachine sm, Event e){
                switch(e){
                    case PowerBut:
                        this.exit(sm);
                        sm.enterState(Running,FanOnly);
                }
            }
        },
        Running(null){
            void entry(StateMachine sm) {
                sm.context.startFan();}
            void exit(StateMachine sm) {
                sm.context.stopFan();}

            void process(StateMachine sm, Event e){
                switch(e){
                    case PowerBut:
                        sm.exitAll(sm.state, this);
                        sm.enterState(Off);
                }
            }
        },

        FanOnly(Running){
            void process(StateMachine sm,Event e){
                switch(e){
                    case ACBut:
                        this.exit(sm);
                        sm.enterState(AC);
                        break;
                    default:
                        parent.process(sm, e);
                }
            }
        },

        AC(Running){
            void entry(StateMachine sm) {
                sm.context.startCondenser();}
            void exit(StateMachine sm) {
                sm.context.stopCondenser();}

            void process(StateMachine sm, Event e){
                switch(e){
```

Listing 2: ACController2.java

```
                    case ACBut:
                        this.exit(sm);
                        sm.enterState(FanOnly);
                        break;
                    default:
                        parent.process(sm, e);
                }
            }
        };

        State parent;
        State(State p){parent = p;}
        abstract void process(StateMachine sm, Event e);
        void entry(StateMachine sm){}
        void exit(StateMachine sm){}
    }// end of enum State
  }// end of class StateMachine
}// end of class ACController2
```

CONCURRENT STATES

State machines may have concurrent substates, which means that all the substates are active when their superstate is active. Figure 3 shows the AC Controller's behavior with concurrent states. When the AC is in Running state, both ACMode and Speed regions (concurrent substates) are active. In the ACMode region, either FanOnly or AC will be active. Similarly, in the Speed region, either Low or High will be active.

To implement concurrent states, few things are added to the StateMachine nested class (Listing 3). Like the state reference (which represents the current state in the state machine), two more references (line 43) are included to represent the current state in each concurrent region. When the current state is Running, the acmode and speed references will refer to the current active state in the ACMode and Speed regions, respectively.

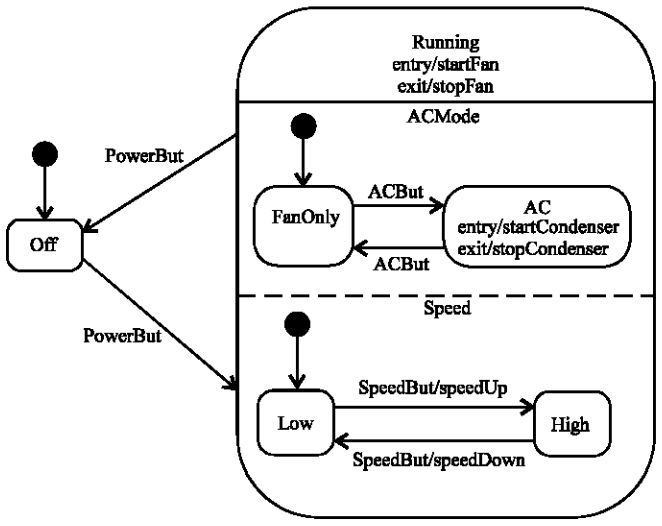

Fig. 3: State machine with concurrent states

Like the enterState method, two more methods are included, which allow to enter nested states in each region (lines 64 and 71). These methods are called whenever the Running state becomes active (lines 97- 99).

The Running's process method delegates the events it receives to the process methods of the acmode and speed (lines 117- 120). In the process method of the ACMode, transitions change current state within the acmode region only (line 152). This way, transitions within the regions are implemented.

Listing 3: ACController3.java

```
public class ACController3 {
    StateMachine stateMachine;

    ACController3(){
        stateMachine = new StateMachine(this);
    }

    // The action methods
    private void startCondenser(){
        /* To be replaced with appropriate code */
        System.out.println("startCondenser executed");
    }
    private void stopCondenser(){
        /* To be replaced with appropriate code */
        System.out.println("stopCondenser executed");
    }
    private void startFan(){
        /* To be replaced with appropriate code */
        System.out.println("startFan executed");
    }
    private void stopFan(){
        /* To be replaced with appropriate code */
        System.out.println("stopFan executed");
    }
    private void speedUp(){
        /* To be replaced with appropriate code */
        System.out.println("speedUp executed");
    }
    private void speedDown(){
        /* To be replaced with appropriate code */
        System.out.println("speedDown executed");
    }

    // Events delegated to stateMachine
    public void powerBut() {stateMachine.powerBut();}
    public void acBut() {stateMachine.acBut();}
    public void speedBut() {stateMachine.speedBut();}

    // The StateMachine class
    static class StateMachine {
        ACController3 context;
        State state;
        State acmode, speed;

        StateMachine(ACController3 context){
            this.context = context;
            state = State.Off;//default
        }

        private void powerBut() {
            state.process(this, Event.PowerBut);}
        private void acBut() {
            state.process(this, Event.ACBut);}
```

Listing 3: ACController3.java

```
    private void speedBut() {
        state.process(this, Event.SpeedBut);}

    private void enterState(State... states){
        for (State s: states){
            s.entry(this);
        }
        state = states[states.length-1];
    }

    private void enterACMode(State... states){
        for (State s: states){
            s.entry(this);
        }
        acmode = states[states.length-1];
    }

    private void enterSpeed(State... states){
        for (State s: states){
            s.entry(this);
        }
        speed = states[states.length-1];
    }

    private void exitAll(State child, State parent){
        State s = child;
        while (true) {
            s.exit(this);
            if (s == parent) break;
            s = s.parent;
        }
    }

    // All of the events
    enum Event {PowerBut, ACBut, SpeedBut}

    // All of the states
    enum State {
        Off (null){
            void process(StateMachine sm, Event e){
                switch(e){
                    case PowerBut:
                        this.exit(sm);
                        sm.enterState(Running);
                        sm.enterACMode(ACMode,FanOnly);
                        sm.enterSpeed(Speed,Low);
                }
            }
        },

        Running(null){
            void entry(StateMachine sm) {
                sm.context.startFan();}
            void exit(StateMachine sm) {
                sm.context.stopFan();}

            void process(StateMachine sm, Event e){
                switch(e){
                    case PowerBut:
                        sm.exitAll(sm.acmode, ACMode);
                        sm.exitAll(sm.speed, this);
                        sm.enterState(Off);
                        break;
```

Listing 3: ACController3.java

```
                case ACBut:
                case SpeedBut:
                    sm.acmode.process(sm, e);
                    sm.speed.process(sm, e);
            }
        }
    },

    ACMode(Running){
        void process(StateMachine sm,Event e){}
    },

    FanOnly(ACMode){
        void process(StateMachine sm,Event e){
            switch(e){
                case ACBut:
                    this.exit(sm);
                    sm.enterACMode(AC);
                    break;
                default:
                    parent.process(sm, e);
            }
        }
    },

    AC(ACMode){
        void entry(StateMachine sm) {
            sm.context.startCondenser();}
        void exit(StateMachine sm) {
            sm.context.stopCondenser();}

        void process(StateMachine sm, Event e){
            switch(e){
                case ACBut:
                    this.exit(sm);
                    sm.enterACMode(FanOnly);
                    break;
                default:
                    parent.process(sm, e);
            }
        }
    },

    Speed(Running){
        void process(StateMachine sm,Event e){}
    },

    Low(Speed){
        void process(StateMachine sm,Event e){
            switch(e){
                case SpeedBut:
                    this.exit(sm);
                    sm.context.speedUp();
                    sm.enterSpeed(High);
                    break;
                default:
                    parent.process(sm, e);
            }
        }
    },

    High(Speed){
        void process(StateMachine sm,Event e){
```

Listing 3: ACController3.java

```
                switch(e){
                    case SpeedBut:
                        this.exit(sm);
                        sm.context.speedDown();
                        sm.enterSpeed(Low);
                        break;
                    default:
                        parent.process(sm, e);
                }
            }
        };

        State parent;
        State(State p){parent = p;}
        abstract void process(StateMachine sm, Event e);
        void entry(StateMachine sm){}
        void exit(StateMachine sm){}
    }// end of enum State
  }// end of class StateMachine
}// end of class ACController3
```

CODE TESTING

We have tested the code and the StateMachine nested class works as expected. Listing 4 shows the test code for ACController3.java (Listing 3). Listing 5 shows the output trace of the test code.

Listing 4: ACController3Test.java

```
import java.io.*;
public class ACController3Test{
  static PrintStream out;
  static ACController3 ac = new ACController3();

  public static void main(String[] a) throws IOException {
    String log = "OutputLog.txt";
    System.out.println("Output is written to: " + log);

    FileOutputStream fos = new FileOutputStream(log);
    out = new PrintStream(fos);
    System.setOut(out);

    out.println("Current State: " + ac.stateMachine.state);
    sendEvent(ACController3.StateMachine.Event.ACBut);
    sendEvent(ACController3.StateMachine.Event.PowerBut);
    sendEvent(ACController3.StateMachine.Event.PowerBut);
    sendEvent(ACController3.StateMachine.Event.PowerBut);
    sendEvent(ACController3.StateMachine.Event.ACBut);
    sendEvent(ACController3.StateMachine.Event.SpeedBut);
    sendEvent(ACController3.StateMachine.Event.PowerBut);
    sendEvent(ACController3.StateMachine.Event.PowerBut);
    sendEvent(ACController3.StateMachine.Event.ACBut);
    sendEvent(ACController3.StateMachine.Event.ACBut);
    sendEvent(ACController3.StateMachine.Event.SpeedBut);
  }
  static void sendEvent(ACController3.StateMachine.Event e){
    out.println("\n--> Event Occured: "+ e);
    switch (e){
        case PowerBut: ac.powerBut();break;
        case ACBut:                          ac.acBut();          break;
        case SpeedBut: ac.speedBut();
    }
```

Listing 4: ACController3Test.java

```
        out.println("Current state: " + ac.stateMachine.state);
        if (ac.stateMachine.state ==
                  ACController3.StateMachine.State.Running){
             out.println("\tACMode state: " + ac.stateMachine.acmode);
             out.println("\tSpeed state: " + ac.stateMachine.speed);
        }
    }
}
```

Listing 5: Output from the test code

```
Current State: Off

--> Event Occured: ACBut
Current state: Off

--> Event Occured: PowerBut
startFan executed
Current state: Running
        ACMode state: FanOnly
        Speed state: Low

--> Event Occured: PowerBut
stopFan executed
Current state: Off

--> Event Occured: PowerBut
startFan executed
Current state: Running
        ACMode state: FanOnly
        Speed state: Low

--> Event Occured: ACBut
startCondenser executed
Current state: Running
        ACMode state: AC
        Speed state: Low

--> Event Occured: SpeedBut
speedUp executed
Current state: Running
        ACMode state: AC
        Speed state: High

--> Event Occured: PowerBut
stopCondenser executed
stopFan executed
Current state: Off

--> Event Occured: PowerBut
startFan executed
Current state: Running
        ACMode state: FanOnly
        Speed state: Low

--> Event Occured: ACBut
startCondenser executed
Current state: Running
        ACMode state: AC
        Speed state: Low

--> Event Occured: ACBut
stopCondenser executed
Current state: Running
```

Listing 5: Output from the test code

```
        ACMode state: FanOnly
        Speed state: Low

--> Event Occured: SpeedBut
speedUp executed
Current state: Running
        ACMode state: FanOnly
        Speed state: High
```

RELATED WORK

The UML state machine is an improved version of finite state machine (FSM). The earliest technique to implement finite state machines is to use switch statement (Aho and Ullman, 1979). Based on the current active state, it performs a jump to the code for processing the event. States are represented as data values. This technique works well for classical flat state machines and is mostly used in non-object-oriented procedural languages.

In object-oriented systems, the behavior of a class of objects is implemented as a set of methods in the class. For classes having complex behavior, the methods will perform differently depending on the current state of the object. An object-oriented extension to the state machine has been done by Coleman *et al.* (1992). They introduced inheritance into state machines and linked class methods to the transitions in the corresponding state machine.

Rumbaugh (1993) proposes an object-oriented approach to implement state machines. He suggests using class-inheritance to represent state hierarchy. The State pattern (Gamma *et al.*, 1995) guides how to implement multi-state classes. Each state is implemented as a different object, which changes at runtime. It does not handle state hierarchy and concurrency. Sane and Campbell (1995) say that states can be represented as classes and transitions as operations. They implement embedded states by making a table for the superstate and do not consider concurrent states. MOODS (Ran, 1996) is a complex variant of the State Pattern. In this variant, the state class hierarchy uses multiple inheritance to model nested states.

Ali and Tanaka (1998) use classes to represent individual states and methods to represent events/transitions. They implement state hierarchy with class inheritance. Their resulting code has too many classes and a class' behavior is not encapsulated within the class.

Douglass (1998) proposed the State Table Pattern to implement state machine diagrams. States and transitions are modeled as classes. Gurp and Bosch (1999) developed a framework of few classes to instantiate and execute finite state machines (FSM). The FSM is not hard coded in the source code. Instead, it is read from an XML file and appropriate objects are created from the framework classes that together represent the FSM. Each state is represented as an object (not a class) and actions as attributes.

Kohler *et al.* (2000) presented an approach for code generation from state machines. Their approach adapts the idea of generic array based state-table but uses object structure to represent state-table at runtime. They use objects to represent states of a state machine and attributes to hold the entry and exit actions. Knapp and Merz (2002) described a set of tools called Hugo for the code generation of UML State Machine Diagram. A generic set of Java classes provides a standard runtime component for the state machine. Every state of a state machine is represented by a separate object, which provides methods for activation, deactivation, initialization and event handling. Events, guards and actions are also implemented as classes.

Chauvel and Jezequel (2005) discuss different approaches to implement state machines. For more efficient code, they suggest to use enumerated values to represent states and events. To handle hierarchical states, they suggest first flattening the state hierarchy. For more flexible code, they suggest to use the State pattern (Gamma *et al*., 1995).

There are many tools (Tiella *et al*., 2007; Jakimi and Elkoutbi, 2009) that generate executable code from state machines. However, the papers do not give details of how states, events and transitions are represented in the generated code.

DISCUSSION

We represent states and events as Java enums, which makes the code efficient. Java enums are loaded when the enclosing class is loaded. Its performance is comparable to primitives. State hierarchy is represented as enum hierarchy and concurrent states are represented by using the concept of object composition. Java does not support enum inheritance explicitly. We used a parent reference in the State enum to handle state hierarchy.

As noted by Chauvel and Jezequel (2005), representing states as enumerations is more efficient than using state classes. They suggest using enumeration for flattened state machines only. We use Java enums to implement all types of state machines including concurrent-hierarchical ones. We improve performance (by using enums) without compromising on flexibility (by using enums-hierarchy).

Java enums, after compilation, are equivalent to objects. Therefore all the approaches (Rumbaugh, 1993; Gamma *et al*., 1995; Sane and Campbell, 1995; Ran, 1996; Ali and Tanaka, 1998; Douglass, 1998; Gurp and Bosch, 1999; Knapp and Merz, 2002) which suggest representing states as classes (or objects) are in the support of our approach.

Among the many approaches we have reviewed, only few (Ran, 1996; Ali and Tanaka, 1998) support concurrent-hierarchical state machines. However the resulting code has too many classes, is less efficient and is not encapsulated in the owner class. Our proposed code is well-structured and the state machine details are obvious in the resulting source code. This makes it easy to reverse-engineer the code back to a state machine, if needed. Furthermore, validating the code against the corresponding state machine is straight-forward. In fact, the code can be generated automatically because there is almost a one-to-one correspondence between state machine elements and our proposed code.

In our proposed code, the behavior of a class, represented by the corresponding state machine, is completely encapsulated inside the class and implemented as a nested class. In principle, the nested StateMachine class and all its members should be private. In the example listings, we did not declare them private so that the test code (Listing 5) can work properly. The StateMachine class is static because Java enum types are static by default and they require a static environment.

Other Elements in State Machines

The UML State Machine Diagram allows events having arguments. In the above listings, we represented each event with an enum-value inside Event enum. This approach is not suitable for events having arguments. Events having arguments can be implemented as separate empty methods like entry and exit methods in the State enum. If a state has transition on that event, the state enum-value will override the method.

In the examples, we have not shown internal events within states and guard conditions on transitions. An internal event causes an action to be executed without changing the current state. A guard represents a boolean expression on a transition. The transition will execute only when the event of the transition occurs and the guard condition is true. They can easily be accommodated in the proposed code. For internal events, the corresponding switch-case in the process method should call only the action method related to the event. It should not change the current state. For transitions having guard conditions, the corresponding code in the switch-case of the process method should be enclosed in an if statement.

CONCLUSIONS

State machine implementation is an important and difficult task in software development. Commonly used programming languages do not support state machines at the language level. Our approach to implement state machines by using Java enums works well for hierarchical and concurrent states. The resulting code is well-structured, efficient and keeps the state machine structure obvious at the code level. We expect that the proposed code will serve as a Java implementation pattern for state machines.

REFERENCES

Aho, A. and J. Ullman, 1979. Principles of Computer Design. Addison Wesley, Massachusetts.

Ali, J. and J. Tanaka, 1998. An object oriented approach to generate executable code from the OMT-based dynamic model. J. Integrated Des. Process Technol., 2: 65-77.

Booch, G., R. James and J. Ivar, 2005. The Unified Modeling Language User Guide. 2nd Edn., Addison Wesley Professional, New York, ISBN: 0-321-26797-4.

Chauvel, F. and J. Jezequel, 2005. Code generation from UML Models with semantic variation points. Proceedings of the 8th International Conference MoDELS 2005, Oct. 2-7, Montego Bay, Jamaica, pp: 54-68.

Coleman, D., F. Hayes and S. Bear, 1992. Introducing objectcharts or how to use statecharts in object-oriented design. IEEE Trans. Software Eng., 18: 8-18.

Douglass, B.P., 1998. Real Time UML - Developing Efficient Objects for Embedded Systems. Addison-Wesley, Massachusetts.

Gamma, E., H. Richard, R. Johnson and J. Vlissides, 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Massachusetts.

Gurp, J.V. and J. Bosch, 1999. On the implementation of finite state machines. Proceedings of the 3rd Annual IASTED International Conference on Software Engineering and Applications, Oct. 6-8, Scottsdale, Arizona, pp: 1-15.

Harel, D., 1987. Statecharts: A visual formalism for complex systems. Sci. Comput. Program., 8: 231-274.

Jakimi, A. and M. Elkoutbi, 2009. Automatic code generation from UML statechart. Int. J. Eng. Technol., 1: 165-168.

Knapp, A. and S. Merz, 2002. Model checking and code generation for UML state machines and collaborations. Proceedings of the 5th Workshop on Tools for System Design and Verification, (WTSDV'02), Reisenburg, Germany, pp: 59-64.

Kohler, H., U. Nickel, J. Niere and A. Zundorf, 2000. Integrating UML diagrams for production control systems. Proceedings of the 22nd International Conference on Software Engineering, (ICSE'00), Limerick, Ireland, pp: 241-251.

OMG, 2010. Unified modeling language. http://www.omg.org/spec/UML/.

Ran, A., 1996. MOODS: Models for Object-Oriented Design of State. In: Pattern Languages of Program Design 2, Vlissides, J.M., J.O. Coplien and N.L. Kerth (Eds.). Addison-Wesley, Boston, USA.

Rumbaugh, J., 1993. Controlling code: How to implement dynamic models. J. Object-Oriented Programming, 6: 25-30.

Rumbaugh, J., I. Jacobson and G. Booch, 2010. Unified Modeling Language Reference Manual. 2nd Edn., Addison Wesley, New York.

Sane, A. and R. Campbell, 1995. Object-oriented state machines: Subclassing, composition, delegation and genericity. ACM SIGPLAN Notices, 30: 17-32.

Sun Microsystems, 2010. The java tutorial. http://java.sun.com/docs/books/tutorial/java/javaOO/enum.html.

Tiella, R., A. Villafiorita and S. Tomasi, 2007. FSMC+, a tool for the generation of java code from statecharts. Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, Sept. 5-7, Lisboa, Portugal, pp: 93-102.