State Machine Design Pattern

Anatoly Shalyto
Head of Programming Technologies Department
St. Petersburg State University of Information Technologies, Mechanics and Optics
14 Sablinskaya Street
Saint-Petersburg, Russia 197101
shalyto@mail.ifmo.ru

Nikita Shamgunov
Software Design Engineer, SQL Server Engine, Microsoft,
11407 183rd Pl NE #M1071
USA 98052, Redmond, WA
u04921@mail.ru

Georgy Korneev
Assistant Professor of Programming Technologies Department
St. Petersburg State University of Information Technologies, Mechanics and Optics
14 Sablinskaya Street
Saint-Petersburg, Russia 197101
kgeorgiy@rain.ifmo.ru

ABSTRACT
This paper presents a new object-oriented design pattern — *State Machine design pattern*. This pattern extends capabilities of State design pattern. These patterns allow an object to alter its behavior when its internal state changes. Introduced event-driven approach loosens coupling. Thus automata could be constructed from independent state classes. The classes designed with State Machine pattern are more reusable than ones designed with State pattern.

Keywords
design, pattern, automaton, automata, finite automata, finite state machine, behavior, state, transition, state chart

1. INTRODUCTION
Finite automata have been widely used in programming since the appearance of [Kle56] which introduced regular expressions and proved an equivalence of a finite automaton and of a regular expression.

Another area where finite automata are widely used is object oriented programming, in which they are used to design object logic. In this area states that have major impacts on object's behavior (*control states*) are being extracted. Note that these automata are significantly different from those used for regular expression matching. In particular, objects are designed in terms of interfaces and methods (terms that don't exist in classical automata) not in terms of recognizable strings. This paper discusses automata that are used in *OOP*.

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright UNION Agency – Science Press,
Plzen, Czech Republic

In *OOP*, when people think of object behavior, they consider the functionality of its methods. But in many real world applications this definition is insufficient — the internal state of an object should also be considered.

The most famous implementation of an object whose behavior depends on its state is the *State* pattern [Gamma98]. However, pattern description is far from being complete, in different sources [Ster01, Gra02] it is implemented in different ways, sometimes even too verbose. Another disadvantage of the pattern is that the implementation of states in different classes causes distribution of the transition logic among these classes. This adds dependencies between the state classes which lead to different issues in class hierarchies design. In spite of these issues State pattern is used in many practical projects including *JDO* [JDO01].

This paper addresses issues of *State* pattern by introducing a new pattern named *State Machine*. Note that [San95] introduced a pattern with the same name for parallel system programming in *Ada95* but still the authors have chosen this name.

To make reuse of state classes possible we introduce an event mechanism. Events are used to let the automaton know that the state should be changed. This allows centralization of the automaton transition logic and loosens coupling between state classes.

More than twenty possible implementations of *State* pattern are described in [Ada03]. *State Machine* pattern might continue this list. The closest pattern from the list is a combination of *State* and *Observer* patterns [Odr96]. However, this pattern is too complicated and it also introduces a new abstraction layer: `ChangeManager` class. In contrast to relatively verbose *Observer* implementation, in *State Machine* transitions between states are based on event-based mechanism. In [San95] another implementation of *State* was introduced. *State* classes coupling was loosened through a state change mechanism based on a state name. This implementation doesn't reduce semantic dependencies between classes and doesn't provide type safety.

2. Pattern Description

Intent

An intent of *State Machine* is the same as an intent of *State*: to make it possible for an object to alter its behavior when its internal state changes (it looks like an object has changed its class). More extensible design is required, than one provided by *State*.

Note that in the intent description so called *control states* are considered. The difference between *control* and *evaluation* states can be illustrated in the following example. In an imaginary bank management system it might make sense to identify two modes: normal mode and bankrupt mode. This modes would be *control* states. On the other hand particular amount of money on the clients' accounts would be an *evaluation* state.

Motivation

Consider a class `Connection` that represents a network connection. A simple connection has two control states: `Connected` and `Disconnected`. A transition between these states occurs either in case of an error or intentionally — via execution of methods connect or disconnect. In the `Connected` state a user can call methods `send` and `receive` of a `Connection` object. In case of an error `IOException` is thrown and connected breaks. If an object is in the `Disconnected` state, `send` and `receive` methods will throw an exception as well. Consider an interface, implemented by `Connection` class.

```
public interface IConnection {
   public void connect();
   public void disconnect();
   public int receive();
   public void send(int value);
}
```

The basic idea of *State Machine* is to separate classes which implement transition logic (*Context*) and state classes. To provide an interaction between *Context* and state classes we use events which are basically objects that state objects pass to *Context*. A difference from the *State* pattern is the way the next state is determined. In *State* next state is explicitly pointed out by the current state. In the proposed pattern it is done by notifying the *Context* with an event. After that it's a *Context's* responsibility to react and possibly change the state. This is done according to the state chart.

The advantage of this design solution is that state classes may be designed independently. They don't need to be aware of each other.

Note that the state charts that are used in *State Machine* are different from those described in [Aho85].

They consist only of states and transitions marked with events. Transition from the current state S to the next state S* occurs on receiving event E if there is a corresponding transition in the state chart.

State chart for the `Connection` class is shown on figure 1.

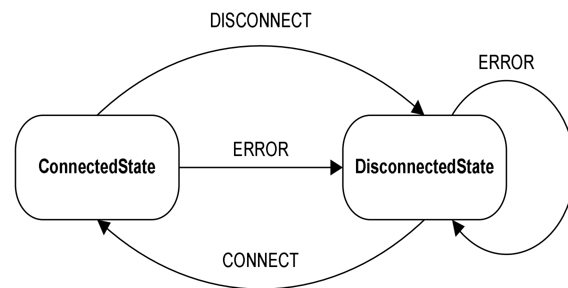

Figure 1. State Chart for class Connection

State classes are called `ConnectedState` and `DisconnectedState`. Event `CONNECT` is used to establish a connection and event `DISCONNECT` is used to break it. `ERROR` is used to indicate an i/o error.

To illustrate the work of the network connection let us take a closer look at its breach in case of an i/o error. If it were implemented through *State* its `ConnectedState` would tell context to switch to `DisconnectedState`. In the *State Machine* case it notifies the *context* through `ERROR` that an i/o error has occurred and the *context* changes its current state. Thus in *State Machine* case `ConnectedState` and `DisconnectedState` classes are not aware of each other.

Application

State Machine could be applied wherever *State* is applied but it also provides additional level of flexibility allowing to reuse the state classes in different automata. It also allows building state class hierarchies.

Structure

Figure 2 shows a structure of *State Machine*.

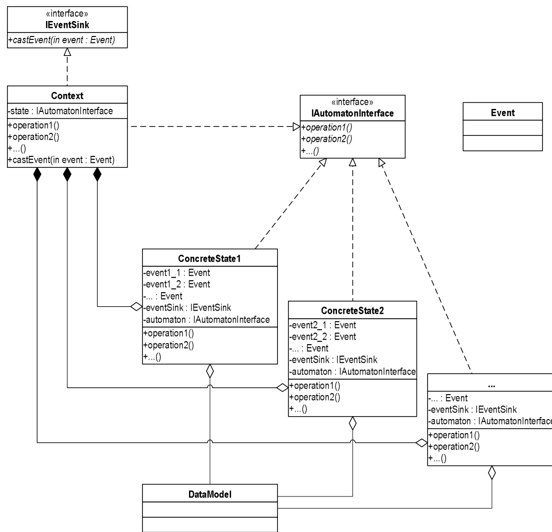

Figure 2. Structure of *State Machine*

`IAutomatonInterface` is an interface of an object to implement, `operation1`, `operation2`, … are the methods of this interface. This interface is implemented by the main class `Context` and by the state classes `ConcreteState1`, `ConcreteState2`, … . Events `event1_1`, `event2_1`, …, `event2_1`, `event2_2`, …, are used to change state. They are instances of the `Event` class. The `Context` class has references to all of the state classes (`ConcreteState1` and `ConcreteState2`) and a reference to the current state. The state classes have a reference to the data model (`dataModel`) and to the event notification interface (`eventSink`). For the purpose of brevity, relations between the state classes and the `Event` class are not shown in the figure.

Members

State Machine consists of the following parts.

- *Automata interface* (`IAutomatonInterface`) — is implemented by the context and is the only way of interaction between the automata and a client. This interface is also implemented by state classes.
- *Context* (`Context`) — is a class that encapsulates transition logic. It implements the automata interface and holds an instance of the data model and the current state.
- *State classes* (`ConcreteState1`, `ConcreteState2`, …) — determine behavior in a particular state. Each of them implements the automata interface.
- *Events* (`event1_1`, `event1_2`, ...) — initiated by the state classes and passed to the context that does a transition depending on the event and the current state.
- *Event notification interface* (`IEventSink`) — implemented by a context. This is the only way of interaction between the state classes and the context.
- *Data model* (`DataModel`) — is a class to provide a shared storage between the state classes.

Note that automata interface in the proposed pattern is implemented by the context and by the state classes. This allows making certain compile-time consistency check. In the *State* pattern such a check is impossible because the context interface doesn't match state classes' interfaces.

Relations

During its initialization the context creates an instance of data model and uses it to create instances of states. It passes the data model an event notification interface (which is a *this* pointer).

During its lifetime an automaton delegates its methods to the current state class. While executing a delegated method the state object might generate an event and notify the context using event notification interface.

The next state is determined by the context on the basis of the current state and the event.

Results

- As in the *State* pattern, the state-dependent behavior is localized in the state classes.
- Unlike the *State* pattern in the proposed pattern transition logic is separated from the behavior in a particular state. The state classes should only notify a context of a particular event.
- Implementation of an automata interface is trivial and could be generated automatically.
- Transition could be implemented as a simple index lookup.
- *State Machine* provides pure (no unneeded methods) interface to a client. To prevent a client from using `IEventSink` we could use private inheritance (in *C++*) or define a private constructor and a static method that creates an instance of `Context`.

- *State Machine*, unlike *State*, doesn't contain redundant interfaces for the context and the state classes — they all implement the same interface.
- It is possible to reuse state classes; moreover, state classes' hierarchies can be created. Note that it is mentioned in [Gam98] that new subclasses are easily added to the state classes. In fact, adding a subclass to a state class causes modification of all the rest of the state classes because the transition logic should be changed. Thus extension of a particular automaton implemented using *State* is being problematic.

Code Sample

The following sample in C# implements `Connection` class described in 2.2. It is a simplified model that allows transmitting and receiving data.

First let's describe interfaces and base classes that are used in this example. These classes are implemented in an assembly `ru.ifmo.is.sm`. Class diagram is shown on figure 3.

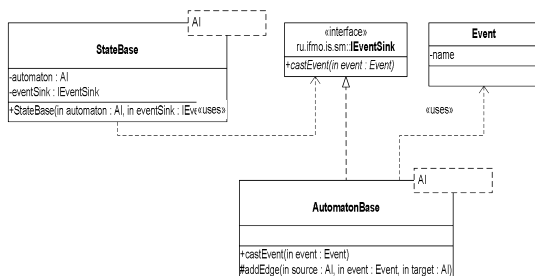

Figure 3. Class diagram for assembly ru.ifmo.is.sm

Let us describe all classes and events from this package:

- `IEventSink` — event notification interface:

```
public interface IEventSink {
   void castEvent(Event ev);
}
```

- `Event` — event class:

```
public sealed class Event {
   private readonly String name;

   public Event(String name) {
      if (name == null) throw new
            NullReferenceException();
      this.name = name;
   }

   public String getName() {
      return name;
   }
}
```

- `StateBase` — base class for all state classes.

```
public abstract class StateBase<AI> {
   protected readonly AI automaton;
   protected readonly IEventSink
         eventSink;

   public StateBase(AI automaton,
         IEventSink eventSink) {
      if (automaton == null || eventSink
         == null) {
         throw new
            NullReferenceException();
      }
      this.automaton = automaton;
      this.eventSink = eventSink;
   }

   protected void castEvent(Event ev) {
      eventSink.castEvent(ev);
   }
}
```

- `AutomatonBase` — base class for all automata. It provides a method `addEdge` for its subclasses. In addition `AutomatonBase` implements `IEventSink`:

```
public abstract class AutomatonBase<AI>
         : IEventSink {
   protected AI state;
   private Dictionary<AI,
         Dictionary<Event, AI>> edges
         =
      new Dictionary<AI,
         Dictionary<Event, AI>>();

   protected void addEdge(AI source,
         Event ev, AI target) {
      Dictionary<Event, AI> row =
         edges[source];
      if (null == row) {
         row = new Dictionary<Event,
            AI>();
         edges.Add(source, row);
      }
      row.Add(ev, target);
   }

   public void castEvent(Event ev) {
      state = edges[state][ev];
   }
}
```

Classes created according to the *State Machine* pattern form an assembly `Connection`. Class diagram is shown on a figure 5.

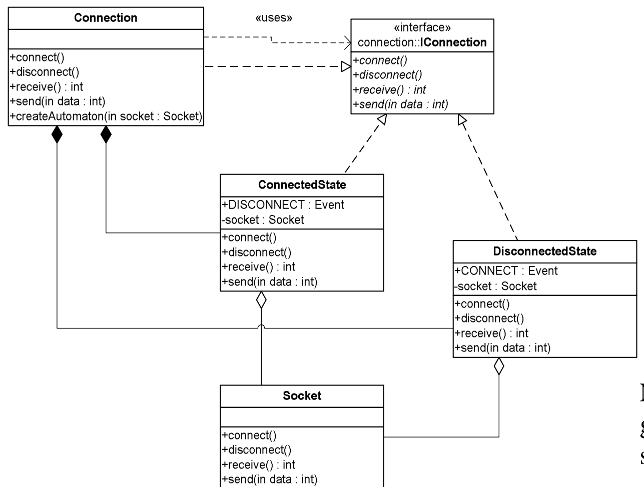

Figure 4. Class diagram for assembly connection

We use class `Socket` as a data model. It implements `IConnection` interface in this example. Control states of the automaton are `ConnectedState` and `DisconnectedState`. In `ConnectedState` we can expect `ERROR` and `DISCONECT` events and in `DisconnectedState` we can expect `CONNECT` and `ERROR` (figure 1).

The code of the state classes follows.

```
public class ConnectedState <AI>
   : StateBase<AI>, IConnection
        where AI : IConnection
{
     public static readonly Event
            DISCONNECT = new
            Event("DISCONNECT");
   public static readonly Event ERROR =
            new Event("ERROR");

   protected readonly Socket socket;

   public ConnectedState(AI automaton,
            IEventSink eventSink, Socket
            socket)
      : base(automaton, eventSink)
   {
      this.socket = socket;
   }

     public void connect() {
     }

     public void disconnect() {
        try {
           socket.disconnect();
        } finally {

            eventSink.castEvent(DISCONNEC
            T);
        }
     }

     public int receive() {
        try {
           return socket.receive();
        } catch (IOException e) {
           eventSink.castEvent(ERROR);
           throw e;
        }
     }

     public void send(int value) {
        try {
           socket.send(value);
        } catch (IOException e) {
           eventSink.castEvent(ERROR);
           throw e;
        }
     }
}
```

Note that state classes only partially specialize generic parameter of `StateBase`. It is used to support inheritance.

Class `DisconnectedState`:

```
public class DisconnectedState <AI>
   : StateBase<AI>, IConnection
      where AI : IConnection {
     public static readonly Event CONNECT
            = new Event("CONNECT");

     public static readonly Event ERROR =
            new Event("ERROR");

     protected readonly Socket socket;

     public DisconnectedState(AI
            automaton, IEventSink
            eventSink, Socket socket)
        : base(automaton, eventSink)
     {
        this.socket = socket;
     }

     public void connect() {
        try {
           socket.connect();
        } catch (IOException e) {
           eventSink.castEvent(ERROR);
           throw e;
        }
        eventSink.castEvent(CONNECT);
     }

     public void disconnect() {
     }

     public int receive() {
        throw new IOException("Connection
            is closed (receive)");
     }

     public void send(int value) {
        throw new IOException("Connection
            is closed (send)");
     }
}
```

Note that state classes define only event generation logic — transition logic is defined in the context.

3. Pattern extensibility

An extension of `Connection` will demonstrate how we can extend automata interface. Let's extend automata interface in the following way.

```
public interface IPushBackConnection :
IConnection {
       void pushBack(int value);
}
```

When calling `pushBack` the value passed as an argument is pushed on top of the stack to be popped in the next call of `receive`. If the stack is empty at the moment when `receive` is called, then the value is being pulled from the socket as in the previous example.

In this case the number of control states doesn't change but the state classes and the automaton must implement an extended interface. Let's call a context of the new automaton `PushBackConnection` and the new state classes `PushBackConnectedState` and `PushBackDisconnectedState`. Here is an implementation of `PushBackConnectedState`. Note that this class extends `ConnectedState` inheriting its logic.

```
public class PushBackConnectedState <AI>
            : ConnectedState<AI>,
            IPushBackConnection  where AI
            : IPushBackConnection
{
   Stack<int> stack = new
            Stack<Integer>();

   public PushBackConnectedState(AI
            automaton, IEventSink
            eventSink, Socket socket)
         : base(automaton, eventSink,
            socket) {
   }

   public int receive() {
      if (stack.empty()) {
         return base.receive();
      }

      return stack.pop();
   }

   public void pushBack(int value) {
      stack.push(new Integer(value));
   }
}
```

`PushBackDisconnectedState` class is implemented in the same way. So we'll only show the `PushBackConnection` code.

```
public class PushBackConnection :
            AutomatonBase<IPushBackConnec
            tion>, IPushBackConnection {
   private PushBackConnection() {
      Socket socket = new Socket();
      IPushBackConnection connected =
            new
            PushBackConnectedState<PushBa
            ckConnection>(this, this,
            socket);
      IPushBackConnection disconnected =
            new
            PushBackDisconnectedState<Pus
            hBackConnection>(this, this,
            socket);

      addEdge(connected,
            PushBackConnectedState<IPushB
            ackConnection>.DISCONNECT,
            disconnected);
      addEdge(connected,
            PushBackConnectedState<IPushB
            ackConnection>.ERROR,
            disconnected);
      addEdge(disconnected,
            PushBackDisconnectedState<IPu
            shBackConnection>.CONNECT,
            connected);

      state = disconnected;
   }

   public static IPushBackConnection
            createAutomaton() {
      return new PushBackConnection();
   }

   public void connect(){
            state.connect(); }
   public void disconnect() {
            state.disconnect(); }
   public int receive() { return
            state.receive(); }
   public void send(int value) {
            state.send(value); }
   public void pushBack(int value) {
            state.pushBack(value); }
}
```

A class diagram for `PushBackConnection` is shown on figure 5.

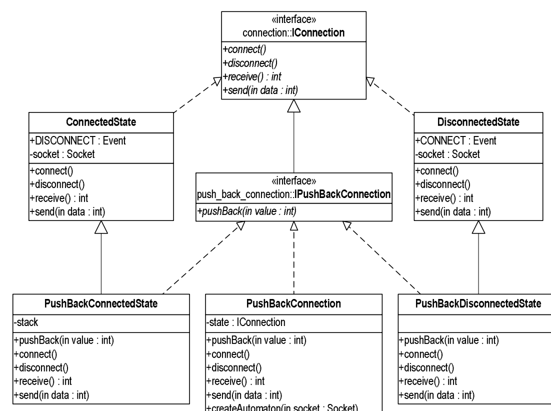

Figure 5. Class diagram interface extensibility example

In a similar way we can reuse state classes when creating a new automaton.

4. Conclusion

State Machine pattern improves *State* and inherits its main idea — to encapsulate the state-dependent behavior in a separate class.

The new pattern improves *State* in the following aspects.

- When using *State Machine* it is possible to design state classes independently. Thus the same state class could be used in several automata. This eliminates the major disadvantage of *State* — reuse issues.
- In *State* transition logic is distributed throughout state classes which introduces coupling between them. *State Machine* addresses this issue. It separates transition logic and the behavior in a particular state.
- As opposed to *State*, *State Machine* doesn't cause interface redundancy.

In *State Machine* you still need to implement trivial delegation of the automata interface methods to the current state. Such a delegation could be done automatically with the aid of *CASE* tools. Another option is to modify a programming language to support automata in a natural way. The authors are working on such language.

5. REFERENCES

[Aho85] Aho A., Sethi R., Ullman J. Compilers: Principles, Techniques and Tools. MA: Addison-Wesley, 1985, 500 p.
[Ada03] Adamczyk P. The Anthology of the Finite State Machine Design Patterns. http://jerry.cs.uiuc.edu/~plop/plop2003/Papers/Adamczyk-State-Machine.pdf
[JDO01] Java Data Objects (JDO). http://java.sun.com/products/jdo/index.jsp.
[Kle56] **Kleene S. C.** Representation of Events in Nerve Nets and Finite Automata, 1956 //Issue [6]. — P. 3–41
[Gamma98] Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns. MA: Addison-Wesley Professional. 2001. — 395
[Gra02] Grand M. Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML. Wiley, 2002. — 544 p.
[Odr96] Odrowski J., Sogaard P. Pattern Integration — Variations of State // Proceedings of PLoP96. http://www.cs.wustl.edu/~schmidt/PLoP-96/odrowski.ps.gz [San96] Sandén B. The state-machine pattern // Proceedings of the conference on TRI-Ada '96 http://java.sun.com/products/jdo/index.jsp.
[San95] Sane A., Campbell R.. Object-Oriented State Machines: Subclassing, Composition, Genericity // OOPSLA '95. http://choices.cs.uiuc.edu/sane/home.html.
[Ster01] Steling S., Maassen O. Applied Java Patterns. Pearson Higher Education. 2001, P. 608