UML Tutorial: Finite State Machines

Robert C. Martin

Engineering Notebook Column

C++ Report, June 98

In my last column I presented UML sequence diagrams. Sequence diagrams are one of the many tools in UML that support dynamic modeling. In this column we will be discussing another kind of dynamic modeling tool in UML, the Finite State Machine (FSM).

UML has a very rich notation for describing FSMs; too rich, in fact, to cover in a single article. Therefore, this article will focus upon the core FSM notation within UML. This notation is a rather traditional version of the State Transition Diagram (STD) with a few interesting wrinkles.

However, the notation for FSMs in UML also includes elements of flow charts and Petri-nets. It is one of the most complete FSM notations every gathered into a single notation. We'll be looking at those other facets of FSM notation in subsequent articles.

What is a Finite State Machine?

Consider a subway turnstile. This simple device is governed by an equally simple FSM. Figure 1 shows part of that FSM. The round rectangles are states. The turnstile has only two states. It can be locked, or it can be unlocked. When the turnstile is locked, a person can drop a coin into its slot. This will cause the turnstile to change to the Unlocked state. This is shown in the diagram by the arrow leading from the Locked state to the Unlocked state. This arrow is called a transition, because it describes how the FSM transitions from one state to another.

Coin / Unlock

Locked   Unlocked

Pass / Lock
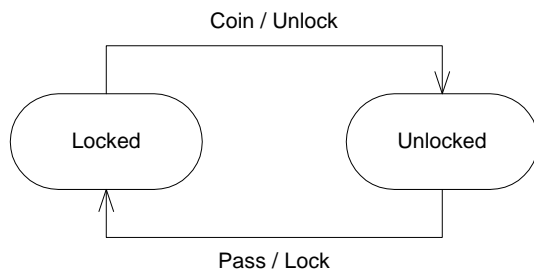
Figure 1: Subway Turnstile

The label on a transition has two parts separated by a slash. The first is the name of the event that triggers the transition. The second is the name of an action to be performed once the transition has been triggered. We can interpret Figure 1 as follows:
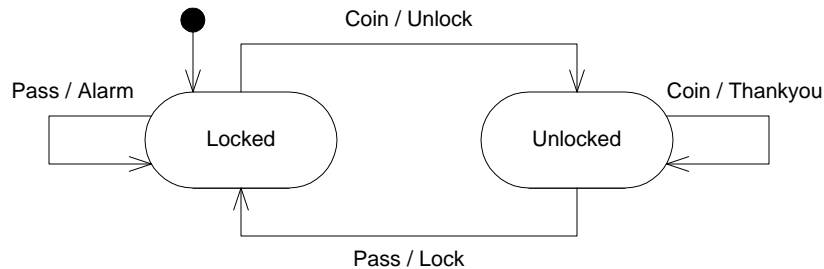
- If the turnstile is in the Locked state, and a Coin event occurs, then the turnstile transitions to the Unlocked state, and the Unlock action is performed.

- If the turnstile is in the Unlocked state, and a Pass event occurs, then the turnstile transitions to the Locked state, and the Lock action is performed.

This describes how the turnstile works when things go as planned. Presume that the turnstile begins in the Locked state. When a customer wants to pass through the turnstile they must deposit a coin. This causes the Coin event to occur. The Coin event, in the Locked state, causes the turnstile to transition to the Unlocked state, and causes the Unlock action to be invoked. Next the customer passes through the turnstile. This causes the Pass event to occur. The Pass event, in the Unlocked state, causes the turnstile to go back to the Locked state, and to invoke the Lock action.

Abnormal Logic

Clearly, the diagram in Figure 1 is an elegant way to describe the logic of the turnstile. The visual presentation of the logic is a powerful aspect of STDs. What makes it even more powerful is the ease with which we can judge the completion of the design.

Notice that there are two possible events. Once we have identified all the events and states, it is a simple matter to apply each event to each state. See Figure 2.

Figure 2 : Turnstile with abnormal events.

What should we do if the turnstile is in the Locked state, but the user passes through[1] anyway? Clearly we should sounds some kind of alarm. Note that the transition that handles this does not change the state. The turnstile remains in the Locked state. The other abnormal condition is when the turnstile is already unlocked and the customer deposits another coin. In this case, we light up a little "thank you" light.[2]

Abnormal conditions, such as those above, occur when normal events happen at unexpected times. The perversity of the world guarantees us that these conditions will certainly occur, but not until we have placed the system into production. It is easy to overlook these conditions and get yourself into trouble. FSMs are a very good way to discover them early, and figure out what to do when they occur.

In Figure 2, note the black dot with the arrow pointing to the Locked state. The black dot is called the Initial Pseudo state. It signifies that the Locked state is the initial state for this FSM. When the FSM is first activated, it will begin in the Locked state.

A better way to handle alarms.

Remaining in the Locked state is probably not the best approach for dealing with someone who has forced entry through the turnstile. Rather, we probably want to enter some kind of Violation state. Moreover, we probably want to remain in that state until a repairman signals that the turnstile is ready for service. See Figure 3.

---

[1] Probably with the help of a sledge hammer.

[2] One of the engineers on this project actually suggested that we ought to refund the coin! At last report, that engineer had become a chicken farmer.
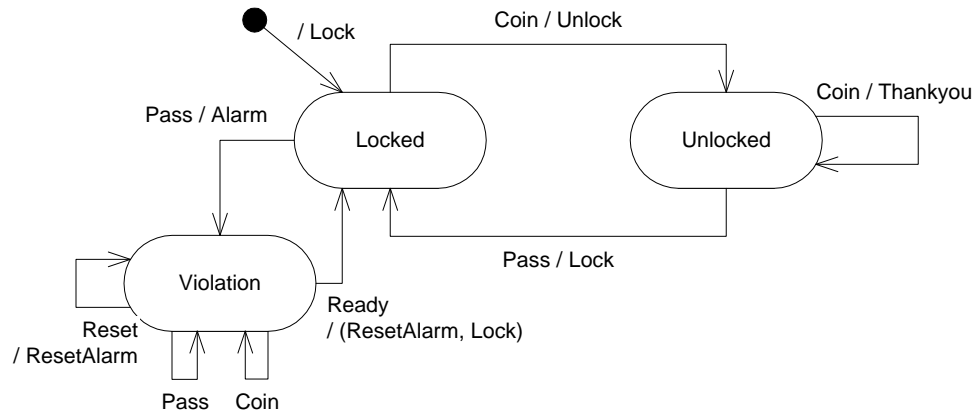
Figure 3 : Turstile with Violation state.

Note that the only way out of the Violation state is through the Ready event. The transition ensures that the alarm is turned off and that the turnstile is locked again. I have also added a special event that the technician can use to turn the alarm off while working on the turnstile. As for the Pass and Coin events, while in the Violation state, they are dutifully ignored.

I have also added an action to the arrow that connects the Initial Pseudo State to the Locked state. This arrow is the very first transition that occurs when the FSM is started up. This action guarantees that the turnstile will be locked at startup.

Note that I have not added Ready events to the Locked and Unlocked states. This event is meaningless to those states, and there is noway to respond to them other than to declare a runtime error. As a default behavior, I expect FSMs to declare some kind of fatal error when events occur for which there are no transitions.

Diagnostic Mode

It is very likely that maintenance technicians are going to want to put the turnstile into a special maintenance mode so that they can check out its functions. In this mode the technician should be able to command the various functions of the turnstile and test its sensors. Figure 4 shows a possible design.
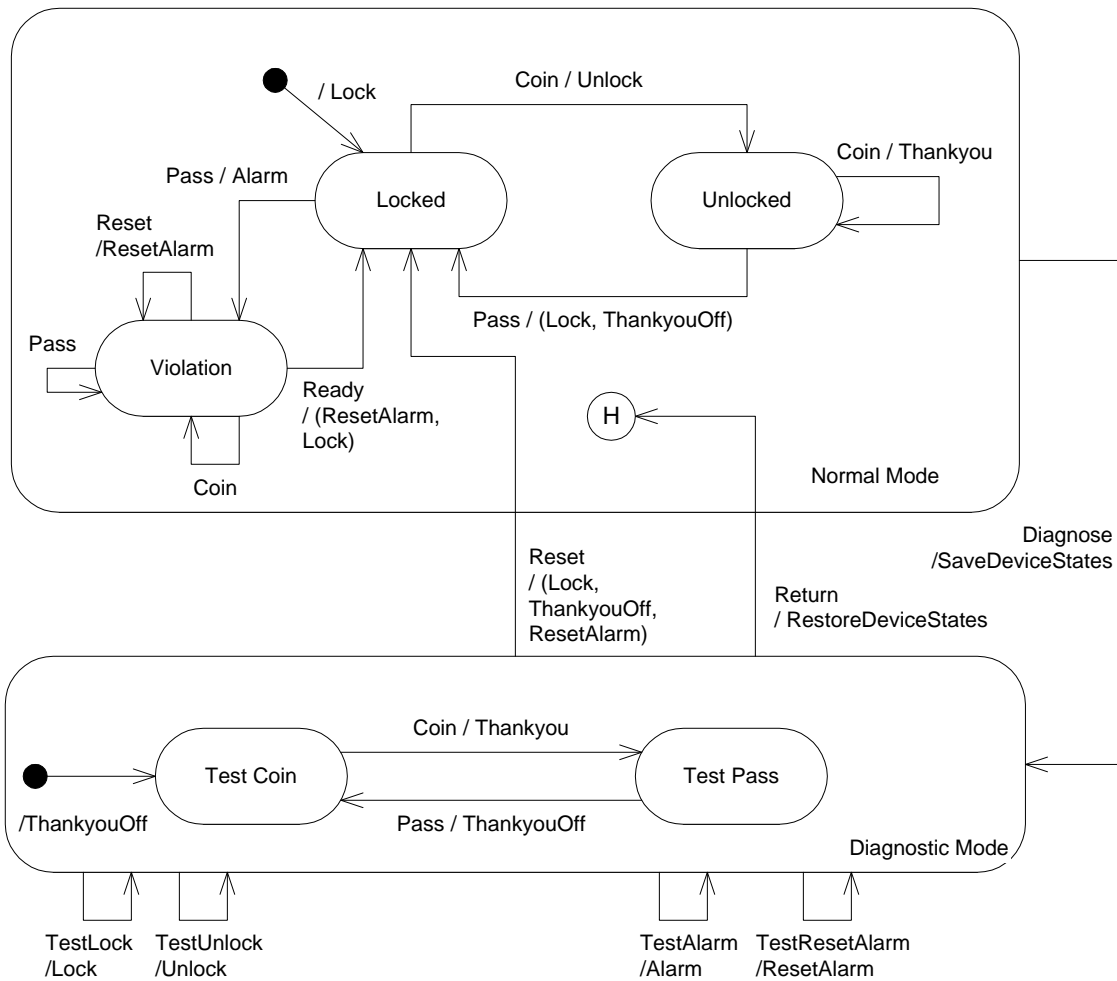
Figure 4: Turnstile with Diagnostic Mode.

Certainly things have gotten a bit more complicated. Diagnostic modes have a tendency to do that to projects.

The most striking features that we see are the large round rectangles that surround the states. These large round rectangles are called super states. There are two super states in the turnstile system. They are Normal Mode and Diagnostic Mode. We refer to the states within the superstates as substates.

As you can see, the states and transitions for the normal operation of the turnstile are contained within Normal Mode. Nothing has changed with regard to them, except that I have added some an action to turn off the thankyou light when the user passes through the turnstile.

I discovered this omission as I considered the fact that within diagnostic mode, the technician ought to be able to turn the thankyou light on and off. And that got me to ask the question, "how does the thankyou light get turned off in *normal* operation?" This is just another reminder that revelations about deficiencies in your designs can come from unexpected sources.

How do we get into diagnostic mode? Somehow the technician asserts the Diagnose event. Notice that the transition labeled with this event leaves from the Normal Mode super state, and not from any of its substates. This implies that, regardless of which substate of Normal Mode we happen to be in, we will leave that substate and enter the Diagnostic Mode superstate. We will also invoke the SaveDeviceStates action which simply remembers the state of the thankyou light, the lock, and the alarm.

Superstates are rather like abstract classes. Abstract classes can only be instantiated as part of a derivative, and superstates can only be entered as part of a substate. When the Diagnostic Mode superstate is entered, one of its substates must also be entered. Since the Diagnose transition does not terminate on one of the

substates, we look to the Initial Pseudo State icon to show us which substate to enter. In this case the Diagnostic Mode begins in the Test Coin state with the thankyou light off.

The technician can test the coin sensor and the thankyou light by depositing a coin. This will force a transition to the Test Pass state, and will turn on the thankyou light. The technician can now test the sensor that detects when a customer passes through. This will extinguish the thankyou light, and return to the Test Coin state.

At any time the technician may also test the alarm and the lock. These tests will not alter the current substate of Diagnostic Mode.

Once the technician has completed all tests, he can return the turnstile to normal mode in one of two ways. He can assert the Reset event, which will put the turnstile back into the Locked state with the alarm off and the thankyou light off. Or he can assert the Return event. This triggers a transition that restores the state of the devices, and then enters the History Pseudo State. The History Pseudo State is the little circle with the H inside. It indicates that the substate within Normal Mode to be entered is the substate within Normal Mode that was last exited. Thus, the Return event will put the turnstile into exactly the same condition it was in before Diagnostic Mode was entered.

Implementing State Machines.

There are a number of techniques for implementing a finite state machine. One of the most common is nested switch case statements. Consider the code in Listing 1 which shows the implementation of the FSM in Figure 2.

Listing 1: Nested switch/case FSM implementation.

```
enum State {Locked, Unlocked};
enum Event {Pass, Coin};
void Unlock();
void Lock();
void Thankyou();
void Alarm();

void Transition(Event e)
{
  static State s = Locked;
  switch(s)
  {
  case Locked:
    switch(e)
    {
    case Coin:
      s = Unlocked;
      Unlock();
    break;

    case Pass:
      Alarm();
    break;
    }
  break;

  case Unlocked:
    switch(e)
    {
    case Coin:
      Thankyou();
    break;

    case Pass:
```

```
            s = Locked;
            Lock();
         break;
            }
      break;
         }
   }
```

Although this technique is common, it is not pretty. As the FSM grows, the nested switch/case statements become very difficult to read. They can go on for page after page after page of code that all looks the same. Moreover, the logic and behavior of the FSM are inextricably bound together. In C++ there is a better way.

Using the State pattern.

We can employ the State pattern from the *Design Patterns*[3] book. Figure 5 shows how this pattern can be used to translate a STD into a set of classes and relationships. We begin with a class named Turnstile, which implements only the four action functions: Lock, Unlock, Alarm, and Thankyou. Derived from this is a class named TurnstileFSM, which implements the Coin, and Pass events. TurnstileFSM contains a pointer to the TurnstileState interface. The event functions of TurnstileFSM delegate to their counterparts in TurnstileState. There are two derivatives of TurnsstileState: LockedState and UnlockedState. Instances of these derivatives are held as static variables of TurnstileState.

If the FSM is in the Locked state, then TurnstileFSM will point at the LockedState derivative of TurnstileState. If a Coin event occurs, the Coin member function of TurnstileFSM will be called. This will delegate to the Coin method of TurnstileState, which will be virtually deployed down to the Coin method of LockedState. This method will call SetState on the TurnstileFSM passing a pointer to the static instance of the UnlockedState, and will then call the Unlock function on TurnstileFSM which TurnstileFSM inherits from Turnstile.

This design has a number of advantages. The behavior is seperated nicely from the logic. All the logic is contained in the TurnstileState hierarchy. All the behavior is contained in the Turnstile hierarchy. If we want to preserve behavior but change logic, we can either change TurnstileState, or create a new derivative of Turnstile and tie it to completely new states. On the other hand, if we want to change the behavior but preserve the logic, all we need do is derive from TurnstileFSM and override the behaviors.

Thus, we have preserved both degrees of freedom. We can modify behavior and logic independently. This is a good thing.

---

[3] Design Patterns, Gamma, et. al, Addison Wesley, 1995.
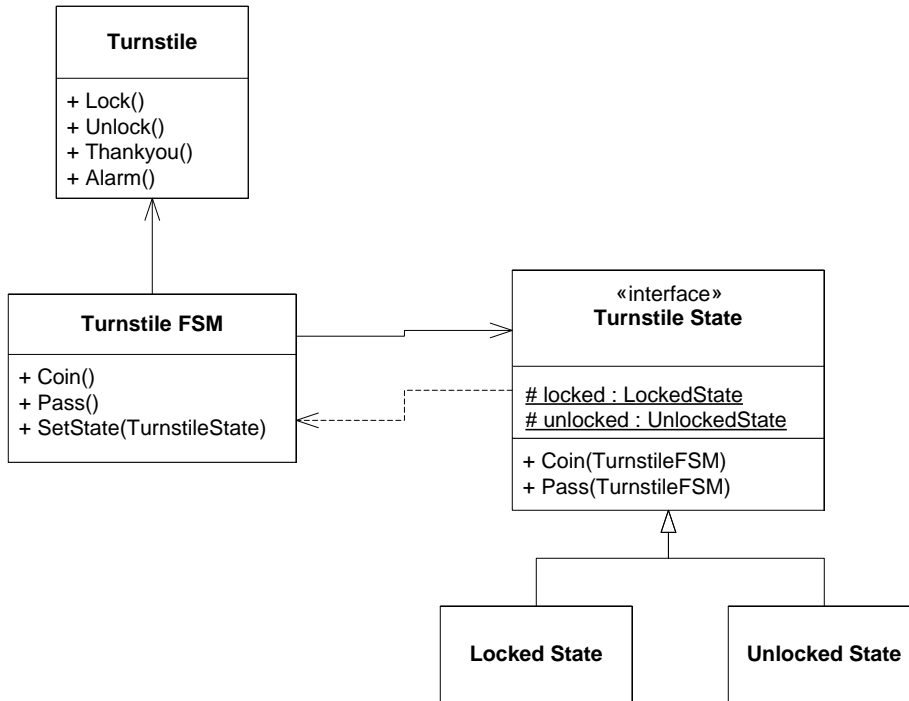
Figure 5: Turnstile FSM designed with the State pattern.

On the other hand, the code is not pretty. An example is in Listing 2. It takes quite a bit of code to implement this simple state machine. What's worse, the vast bulk of that code is all repetitious boiler plate. Again, there is a better way.

Listing 2: Turnstile FSM implemented in C++ using the State pattern.

```
class Turnstile
{
  public:
    virtual void Lock();
    virtual void Unlock();
    virtual void Thankyou();
    virutal void Lock();
};

class TurnstileFSM;
class LockedState;
class UnlockedState;
class TurnstileState
{
  public:
    virtual void Coin(TurnstileFSM*) = 0;
    virtual void Pass(TurnstileFSM*) = 0;
  protected:
    static LockedState lockedState;
    static UnlockedState unlockedState;
};

class TurnstileFSM : public Turnstile
{
  public:
    void SetState(TurnstileState* s) {itsState = s;}
    void Coin() {itsState->Coin(this);}
    void Pass() {itsState->Pass(this);}
```

```
    private:
      TurnstileState *itsState;
  };

  class LockedState : public TurnstileState
  {
    public:
      virtual void Coin(TurnstileFSM* t)
      {
        t->SetState(&unlockedState);
        t->Unlock();
      }

      virtual void Pass(TurnstileFSM* t)
      {
        t->Alarm();
      }
  };

  class UnlockedState : public TurnstileState
  {
    public:
      virtual void Coin(TurnstileFSM* t)
      {
        t->Thankyou();
      }

      virtual void Pass(TurnstileFSM* t)
      {
        t->SetState(&lockedState);
        t->Lock();
      }
  };

  LockedState TurnstileState::lockedState;
  UnlockedState TurnstileState::unlockedState;
```

Generating the code

The state machine in question can be completely described by using the following sixteen words:

```
Locked      Coin     Unlocked    Unlock
Locked      Pass     Locked      Alarm
Unlocked    Coin     Unlocked    Thankyou
Unlocked    Pass     Locked      Lock
```

This table is known as a State Transition Table (STT). It can be interpreted as follows:

- If we are in the Locked state, and we get the Coin event, transition to the Unlocked state and invoke the Unlock action.

- If we are in the Locked state, and we get the Pass Event, then stay in the Locked state and invoke the Alarm action.

- If we are in the Unlocked state, and we get the Coin event, then stay in the Unlocked state and invoke the Thankyou action.

- If we are in the Unlocked state, and we get the Pass event then transition to the Locked state and invoke the Lock action.

An STT a very dense and desirable representation. Although Listings 1 and 2 are much more complex than the STT, they contain no more semantic information. They say precisely the same thing, but take far more words to say it.

What we need is a way to translate an STT into code that follows the pattern in Listing 2. And there is such a way. Long ago, (1988 to be exact) I was creating lots and lots of state machines using a technique similar to Listing 2. I coded the first few by hand. But I soon realized that what I was doing was a rote mechanical process that the computer could do for me.

So I brushed off my Yacc book and wrote a compiler named SMC. SMC translates STTs into the kind of C++ code you see in Listing 2. Actually, the programmer must write one class called the "context". This is a class that has all the action functions declared. In our example this is the Turnstile class. Then the programmer writes the STT pretty much as follows:

```
Context Turnstile
{
Locked     Coin     Unlocked    Unlock
Locked     Pass     Locked      Alarm
Unlocked   Coin     Unlocked    Thankyou
Unlocked   Pass     Locked      Lock
}
```

The Context statement simply tells SMC what class to inherit from. From this input the compiler generates all the other classes. Since SMC generates code that inherits from your context class, you never have to look at the C++ it outputs. There is never a need to edit or touch it in any way. You just run SMC and compile what comes out.

You can get this compiler from the 'freeware' section of my website http://www.oma.com. It is called SMC. It comes complete with lots of documentation and source code. And best of all, its free!

Conclusion

Finite state machines represent a very powerful way of describing and implementing the control logic for applications. I have used them to implement communication protocols, to control the interactions in a GUI, and many many other applications. FSMs are powerful because they are a very dense representation. A lot of logic can be represented by a relatively small diagram.   They are powerful because they follow very simple rules, and are easy to verify. And they are powerful because they can be used to generate code.

There is much more to say about finite state machines. I'll continue with this topic in my next column.