Programming with State Machines

Some kinds of objects have a definite *state* that *changes* the way they behave. Examples are a stopwatch, a digital alarm clock, a calculator, and an ATM machine.

Example: A stopwatch with 2 buttons: StartStop and Hold. Pressing StartStop toggles between running and stopped. Press Hold while stopwatch is *running* freezes the display time but the stopwatch is still running. A Hold button press when watch is stopped has no effect.

Now we will describe how the system behaves as a *State Machine*.

Identify States

This stop watch has 3 states: STOPPED, RUNNING, and HOLD.

Events

The next step is to identify the *events* that cause the stopwatch to change state. *Events* can be external (someone presses a button) or internal ("alarm time reached" or "finished printing receipt").

The *events* for this stopwatch are:

StartStop Pressed

Hold Pressed

Actions and Activity

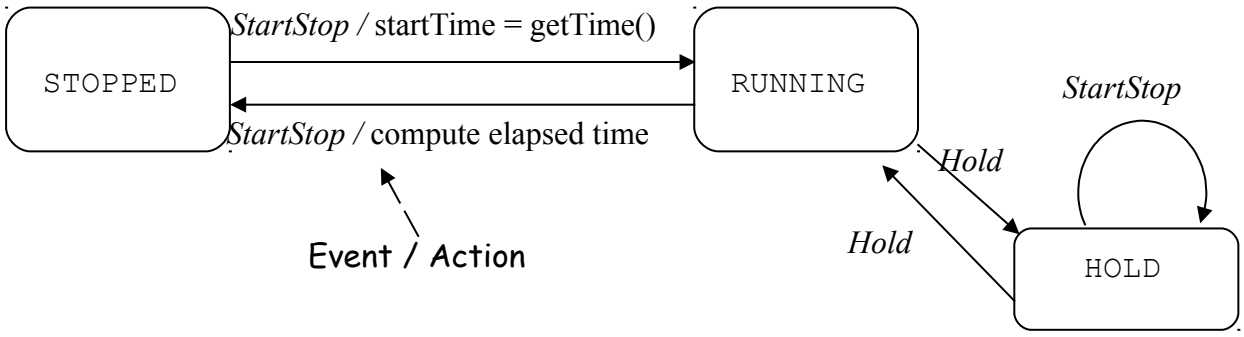
Some state machines perform an *action* when they change state. For example, when a stopwatch changes from STOPPED to RUNNING, it starts the timer.

Actions are something the component does in response to an event, or as part of the response. An *action* is something that is done instantaneously or takes very little time. In contrast, an *activity* is some behavior that is performed for a longer time, e.g. while the component is in a state.

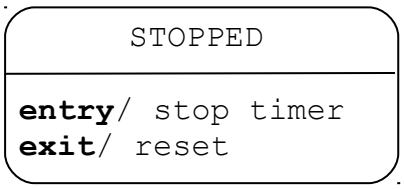
In the stopwatch, "starttime = current time" or "compute elapsed time" are *actions* – they effectively take no time to perform. "Update display (every millisec)" is an *activity*.

Draw a State Machine Diagram

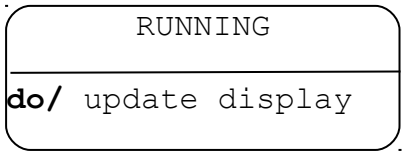
A UML *State Machine Diagram* is a visual representation of the states, events, actions, and activities. The state machine diagram for a stopwatch is:



A state machine can perform an action as soon as it enters a state or just before leaving a state. If this action is always performed, show it as part of the state using notation like this:



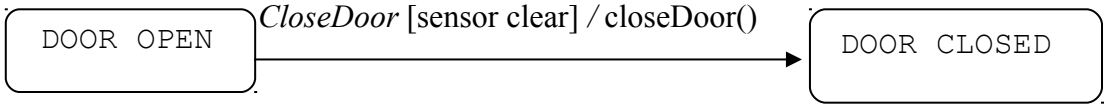
A state machine may perform an *activity* during the whole time it is in the state. You can show this activity using "do / *activity*" in a state box:



Guard Conditions

A State Transition may have a test condition that must be true in order for the transition to occur. These are call *guard conditions*. Guard conditions are boolean valued (true/false) and shown inside [square brackets]. You can use words to describe a guard condition.

Consider a door with a sensor that detects if an object is in the doorway. The door has a button which triggers a *CloseDoor* event. When the *CloseDoor* event occurs, the sensor test "sensor clear" (nothing in the doorway) must be true in order to close the door. The *guard condition* is "sensor clear".



As shown above, the UML notation for a transition with a guard condition is:

Event [guard condition] / *Action*

The *Event* is required; the guard condition and action are optional.

Programming a State Machine Model

We can implement states in code in two different ways:

1. use a simple variable to indicate the state (**`int`**, **`char`**, or **`enum`**) and a switch statement in the event handlers to describe what to do in each state. This is the C language style for coding a state machine.

2. Use an object for state. The state object encapsulates all state-dependent behavior. This is the basis for the *State Design Pattern*.

Technique 1: Use a State variable with Switch Statement

Use a primitive variable or *enum* type to keep track of the state, and a `switch` block to handle behavior for each state.

Define *named constants* for the different state values.

```
final int STOPPED = 0;
final int RUNNING = 1;
final int HOLD = 2;
private int state ;          // variable for the current state
```

Instead of `int`, you can use an *enum* for states. You can define the enum inside the class that uses it:

```
enum State { STOPPED, RUNNING,  HOLD };

private State state; // the current state, using enum type
```

Typically you will write one *method* to handle each *event*:

```
/** This method handles the "StartStop" button event. */
public void handleStartStop( ) {
      switch ( state ) {
      case STOPPED:
            starttime = gettime( );  // record start time
            state = RUNNING;         // change the state
            break;
      case RUNNING:
            elapsed = gettime( ) - starttime;
            state = STOPPED;         // change the state
            break;
      case HOLD:
            // don't do anything
            break;
      }
```

Technique 2: *States as Objects with Delegated Behavior*

This is the O-O approach to implementing state-dependent behavior. You define one object for each state and *delegate* behavior to the state object. This is the *State Design Pattern.*

There are 4 steps to implementing this pattern.

2.1 *Define an interface for all the behavior that depends on state*

```
public interface State {
   public void handleStartStop();
   public void handleHold();
   public void enterState();  // optional
   public void leaveState();  // optional
```

```
}
```

The methods `enterState()` and `leaveState()` are optional. They provide a way to implement the "`enter/`" and "`exit/`" actions of every state.

2.2 *Implement the interface for each State*. Create one class for each state. This class will handle the state-dependent behavior.

Since the states are performing behavior *for* the StopWatch (in this example), they need a *reference* to the StopWatch. Typically, you give the state objects a reference to the *context* via a constructor parameter. Another way is to define the states as *inner classes* (classes inside the StopWatch class), so they automatically have access to the "outer" class.

In this example, we supply a reference to the StopWatch as a constructor parameter:

```
public class StoppedState implements State {
     public StopWatch watch;

     public RunningState(StopWatch watch) {
          this.watch = watch;
     }
     public void handleStartStop() {
          watch.starttime = System.currentTimeMillis(); // do work
          watch.setState( watch.runningState );
     }
     public void handleHold() {
          // ignore hold in stopped state
     }
     public void enterState( ) {
          watch.stoptime = System.currentTimeMillis();
     }
```

2.3 In the context (StopWatch) define a state attribute to keep track of the current state:

```
public class StopWatch {

   private State state;
```

Don't forget to initialize the **`state`** !

2.4 Methods of the the StopWatch that depend on state will *delegate behavior* to the state object:

```
     public void handleStartStop() {
        state.handleStartStop();
     }
     public void handleHold() {
        state.handleHold();
     }
```

The **`state`** variable always refers to the current State, so the current **`State`** object receives the method calls. The Stopwatch is changing how it behaves (by delegating to different State objects) without using "if" or "switch".

2.5 Finally, create one State object for each state and provide a setState( ) method for changing the state. Since you only need one object for each actual state, you can define them as **`final`** references.

```
public class StopWatch {
   public final State runningState = new RunningState( this );
   public final State holdState = new HoldState( this );
   public final State stoppedState = new StoppedState(  this );
   /** variable for referring to the current state */
   private State state;
```

```
/** Change the state of the stopwatrch. */
public void setState( State newstate ) {
    state.leaveState( );
    state = newstate;
    state.enterState( );
}
```

When To Use O-O Approach?

Using the simple state variable approach using a switch statement is efficient. When the behavior is simple and not too many states (as in counting syllable in a word) the simple approach works well and may be faster.

However, if many methods depend on state then the simple approach will have a lot of switch statements spread across many methods. This increases the chance of error and makes the code harder to maintain.

When states are more complex or there are many methods that depend on state, the O-O approach can simplify your code and reduce the chance of overlooking state-dependent behavior.

References

For the object-oriented way of using states, see these two sources:

- Wikipedia, http://en.wikipedia.org/wiki/UML_state_machine examples of how to use a State Machine.
- Wikipedia, http://en.wikipedia.org/wiki/State_diagram how to read a State Diagram.
- "Programming Without Ifs Challenge" at http://programmingwithoutifs.blogspot.com.
- *Head First Design Patterns,* Chapter 10 "The State of Things". Humorous example.