Object-Oriented State Machines

Ted Faison

116 Agostino Irvine, Ca 92714
(714) 261-2752 (H)   (714) 753-0777 (W)

June 7, 1993

Ted Faison is a developer and writer, and has authored several articles and books on C++. He has been programming with the language since 1988, and holds a BSEE from California State University, Fullerton. Ted is president of Faison Computing, a firm which develops C++ class libraries for DOS and Windows. He can be reached at tedfaison@msn.com.

Finite state machines have been in use in computer programming for many years. Systems that are primarily reactive in nature are often well described in terms of input events and outcomes. Such event-driven systems can be expressed as a collection of *states*, each representing a mode in which the system has a specific set of behaviors or reactions to input events. States can react to events by switching the system to another state, by ignoring the event, or by generating other events. State machines represent a conceptually simple way to handle disparate programming tasks, and indeed find their way into many different kinds of programs - from grammar parsers to CD players. State machines often simplify software designs dramatically, because they mimic the way systems work conceptually, allowing difficult problems to be broken down into groups of simpler ones.

The main tool in designing a state machine is the *state transition diagram*, or *state diagram* for short. A state diagram is a simply directed graph whose nodes represent states and whose arrows represent transitions from one state to another. The arrows are labeled with the name of the event that caused the transition. Some transitions might occur on conditionally, and in these cases, the condition is shown next to the event name, in parentheses. Figure 1 shows a simple state diagram.

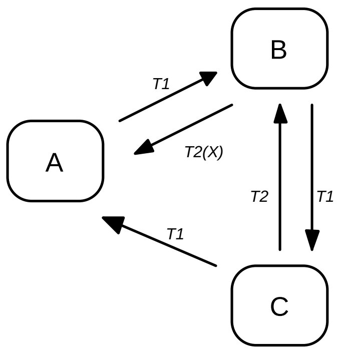

Figure 1 - A simple state transition diagram.

The system in figure 1 has the 3 states denoted by the nodes A, B and C. The system also recognizes the two events T1 and T2, and is allowed to change state only as a result of an event. When the system is in state A, a T1 event will cause it to switch to state B. In state B, a T2 event will provoke a transition back to state A, but only if the condition X is true. A T1 event will always cause a transition from state B to C. Although the system deals with 2 types of events, not all the nodes in figure 1 have a response for the two events. For example, node A has no reaction to T2 events. Events that are don't cause a state transition are often not shown on a state diagram. At times, just to underscore that an event is handled without causing a state transition, a transition arrow is shown for it, going from one state back to the same state. Note that if an event causes no state transition, it doesn't necessarily mean that the event is ignored by the system. Moreover, the state diagram is a graphical representation of the overall control logic of a system, and doesn't shown the internal processing activity at each node.

Conventional programming techniques have used the approach of a *state machine engine* to execute a state diagram. The diagram is often encoded in software as an array of pointers to handling routines, sometimes known as the *dispatch table*. When an input event occurs, the system indexes into the dispatch table, and invokes one of the table's functions. The index into the dispatch table is derived from the system state (and often is the state variable itself). This technique works, but is not without problems. The biggest drawback is that the programmer is responsible for writing low level code for table look-ups, index management and bounds checking. On top of this there needs to be code to detect invalid events, allowable state transitions and events that result in no state changes. Using OOP, the handling of these situations can be shifted from the programmer to the programming language. This article is not a formal and in-depth study of state machines in general. Its objective is to present a new object-oriented approach that simplifies some of the well-known problems associated with non-trivial state-oriented systems, such as dealing with states that share behaviors, dealing with time-outs, counted-event boundary conditions, and the elimination of the stand-alone state machine engine.

The traditional approach

State diagrams are convenient, because they are easy to create, manage and understand. Even better, there is an almost one-to-one relationship between the diagram and the code of the system it represents. Let's step back a minute and look at the way state machines are normally built today. Consider again the state diagram in figure 1. The diagram includes a set of allowed nodes (A, B and C), recognized events (T1, T2), and transitions that occur when events are received. Each node identifies the system in a particular state.

The traditional state machine implementation separated the process responsible for handling and dispatching input events from the management of the data and code that represent the states. There was a dispatch table, governed by a dispatching routine, and there were the state handlers, represented by disjoint functions and data that the dispatcher would call or use. When an event was received, the dispatching routine would use the dispatch table to look-up the handler, based on the system's state. The handler would then be invoked and the event processed. The crucial element is all of this was the state variable, used to look up event handlers. The dispatching routine would use this variable, while the individual event handlers would change it. Thus the *producer* and *consumer* of the variable were not only separate, but completely oblivious of one another, giving rise to a weakness in the system software.

Once an event was dispatched to the correct handler, the handler would classify it and process it. For each possible event, the handler needed to have a data structure to indicate the *next state*, thus each handler would have a list of next states, leading to a system shown in figure 2.

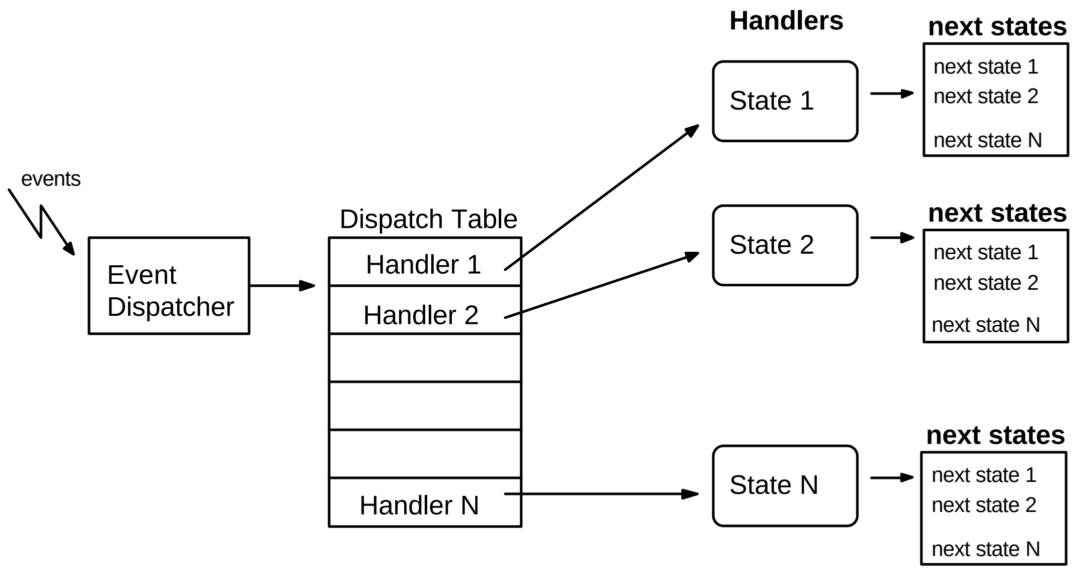

Figure 2 - A common non-OOP event dispatching technique.

The figure illustrates the separation between data and functions. The flow of control is messy, going from the code in the Event Dispatcher. through a dispatch data structure, to more code, where the next-state tables are accessed. In an OOP implementation, there is a merging of the data structures with their related functions. The resulting flow of control is cleaner, involving code only, as shown in the following sections.

An object-oriented approach

Having recognized that a system state is defined by the association of its input events to its next states, it makes sense to incorporate all this knowledge into a single software object. The system can thus be completely represented by such an object. Each system state can be encoded into a separate object, but only one state object need exist at any given time. The current state of a system will be indicated by the object I'll call the *current state object*. The role of the system in response to an input event will be to direct the event to the current state object, as shown in figure 3.

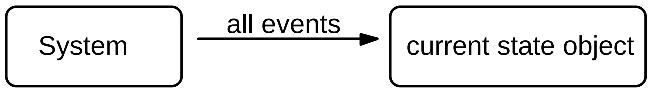

Figure 3 - Using a state object to handle events.

The state object is responsible for handling all the input events, so the system can unconditionally give all the events to the object. This object will process the event, and then return control back to the system. The current state object is not just any kind of object, but a *self-mutating* one. When an event causes a state transition, the current state object transforms itself into a new state object, but without knowledge or intervention from the system. The system need not know what class the current state object represents, only that the object will handle all input events passed to it. In the course of handling events, the current state object's *type* will determine the state of the system. This type will change dynamically, but not for all events. Many events in a system will cause no state transitions, in which case no type mutation will occur.

OOP programming makes it easy -- almost natural -- to handle these kinds of events, using inheritance. OOP also makes it easier to build robust state machines, because both the state handling and dispatching logic are in the same object.

From state diagrams to class hierarchies

Up to this point, all I've described is a state object that looks pretty much like a block of data, holding code that maps input events into next states. The key to keeping the system simple is to organize the possible nodes of a system's state diagram into a class hierarchy, whose root class offers a single, polymorphic interface to the system. The state machine of figure 1 could be implemented with the class hierarchy on figure 4:

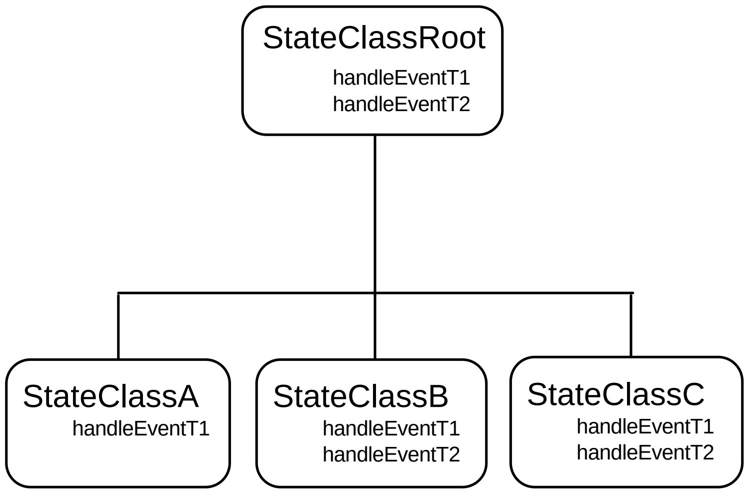

Figure 4 - A class hierarchy representing the nodes shown in figure 1.

In figure 4, class StateClassRoot is the single, polymorphic interface to all the states in the system. The class has two behaviors -- called functions in certain programming languages -- that polymorphically process the two events T1 and T2. The root class StateClassRoot is the functional interface to the class hierarchy. Each derived class represents a node in the state diagram. Each derived class changes the default event behavior of the parent class by defining its own specialized handlers. Note that if a derived classes omits a handler for a particular event, it automatically inherits the handler form the parent class. For example, state A has no response for T2 events. This is shown in the class hierarchy by the absence of a T2 handler in class StateClassA. Being able to inherit event handlers turns out to be very useful in complex systems, especially when dealing with super-states and state clusters, which I'll discuss briefly at the end of the article.

Handling state transitions

When a state machine starts up, it is put into an initial state, represented by an instance of the one the classes {StateClassA..StateClassC}. The behaviors handleEventT1 and handleEventT2 are polymorphic: each time an event is received, the system need only invoke the T1 or T2 handler of the current state object -- without any knowledge of what state the system is currently in. For example, when a T1 event is received, a handleEventT1 message will be sent to the current state object. The system will dynamically bind the message to an actual state object at runtime.

The polymorphic interface of the class hierarchy makes it unnecessary for a system to maintain separate dispatching and handling routines for the input events. The system keeps track of its current state by instantiating an object of one of the classes {StateClassA..StateClassC}. Each time an event is received, the system dispatches it to the current state object. Each class will have its own handlers for events T1 and T2, unless a handler is inherited from the parent. These handlers will be responsible for processing the events, and also for switching the system to its next state. The class hierarchy's polymorphic interface has taken on the role of the traditional function dispatcher, with the individual classes corresponding to states in the state diagram. The state switching code instantiates a new object of the appropriate class. Once a new state is switched to, the old state object is discarded. Figure 5 gives a graphical depiction of the event handlers for a class hierarchy representing the system in figure 1:

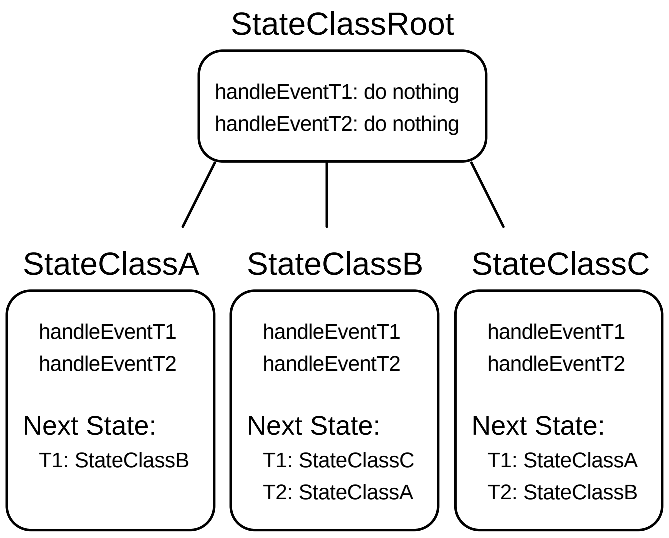

Figure 5 - The future vectors used by the state class hierarchy.

The *next states* are not saved as data, but as code in the event handlers. When an event requires a state transition, the current state object will remove itself from the system and instantiate a new state object of the correct type. When the system invokes the T1 or T2 handler of the current state object, it doesn't know which type of object will actually handle the event. The late binding nature of OOP languages lets the system keep track of runtime function bindings, as shown in figure 6.

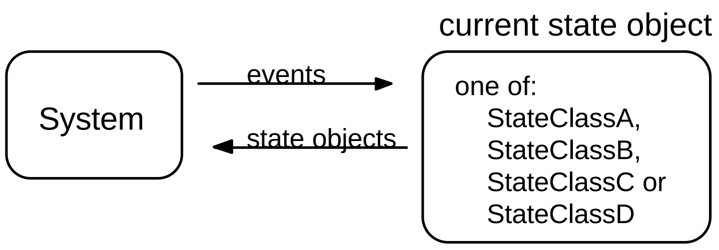

Figure 6 - How the OOP system handles events and state transitions.

All of the knowledge regarding the handling of input events and state transitions is relegated to the class hierarchy. From the system's perspective, there really are no states, merely a handler object. Of course the handler changes type dynamically at runtime in response to input events, but without any intervention from the system. Changes in the system's state have become changes to the current state object's type. The system is still a state machine, but with an emphasis on object types rather than the value of a state variable.

A C++ Example

To bring the concepts completely into focus, I will show a simple example of the technique described. Up to this point, my discussion has been essentially language-independent, but now I will use C++ to illustrate the details of a simple object-oriented state machine. The example will be a simple 2-story elevator control system. The system's state diagram is represented with the diagram in figure 7:

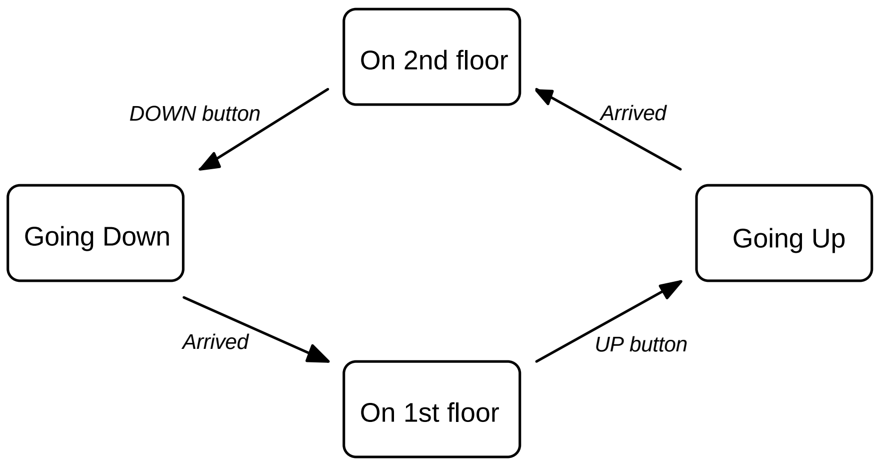

Figure 7 - The state diagram for a simple 2-floor elevator system.

The diagram shows only the events that result is a state change. All events not shown result in no state change. For example, pressing the UP or DOWN button while the elevator is going up produces no state change. Events that cause no state change are not necessarily ignored altogether. Consider a traffic control system that manages the traffic lights at an intersection. When the system is in the RED LIGHT state for one of the streets, it may be counting the number of cars that arrive at the red light. Each arriving car triggers an input event, which the system keeps track of. The more cars arrive, the sooner the system will need to switch to the GREEN LIGHT state. The car events don't produce a state change directly, but they do have an effect on the system.

The state diagram in figure 7 assumes there are 3 events: pressing the UP button, the DOWN button, and arriving at a floor. Since there are only 3 events to be handled, each class in the state class hierarchy should have 3 member functions to process the events. Actually, all state-independent processing can be inherited from a common base class. All events that result in no state changes are ignored by the state classes and can be handled by the base class. As it turns out, each state need only supply a single event handler. Figure 8 shows a graphical depiction of the class hierarchy for the elevator example.

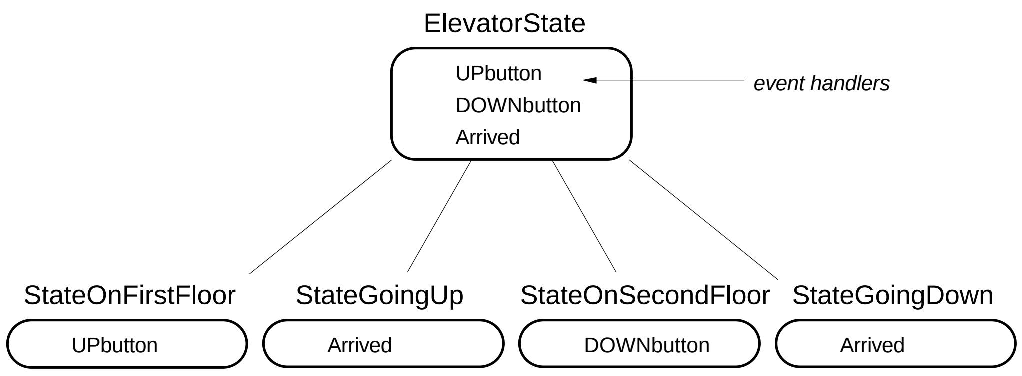

Figure 8 - The class hierarchy used in the elevator example.

In figure 8, each box represents a class, whose name is shown at the top of the box. Inside the boxes are the names of the handler functions supported by each class. For example. class StateOnFirstFloor has only the UPbutton handler, since all other events are inherited from the parent class ElevatorState. The classes shown in figure 8 are declared with the code shown in listing 1.

```
class ElevatorState {

public:

  virtual ~ElevatorState() {}
  virtual ElevatorState* UPbutton() {return this;}
  virtual ElevatorState* DOWNbutton() {return this;}
  virtual ElevatorState* Arrived() {return this;}
  virtual char* Description() {return "Invalid State";}
};


class StateOnFirstFloor: public ElevatorState {

public:

  virtual ~StateOnFirstFloor() {}
  ElevatorState* UPbutton();  // start moving up
  char* Description() {return "On First Floor";}

};

class StateGoingUp: public ElevatorState {

public:

  virtual ~StateGoingUp() {}
  ElevatorState* Arrived();   // stop the elevator
  char* Description() {return "Going Up";}

};

class StateOnSecondFloor: public ElevatorState {

public:

  virtual ~StateOnSecondFloor() {}
  ElevatorState* DOWNbutton();      // start moving down
  char* Description() {return "On Second Floor";}

};

class StateGoingDown: public ElevatorState {

public:

  virtual ~StateGoingDown() {}
  ElevatorState* Arrived();   // stop the elevator
  char* Description() {return "Going Down";}

};
```

Listing 1 - The declaration of the class hierarchy used in the elevator example.

There is essentially a one-to-one correspondence between the class hierarchy in figure 8 and the class declarations in listing 1. Each class has an additional member function named **Description**, used in the example to display the name of the state the elevator system is in. This function is unrelated to event handling, hence its absence from figure 8. The code for the class hierarchy is very short, and is shown in listing 2.

```
ElevatorState* StateOnFirstFloor::UPbutton()
{
  // turn on the UP indicator light
  // and start the UP motor
  // ...
  // switch to the next state
  delete this;
  return new StateGoingUp;
}

ElevatorState* StateGoingUp::Arrived()
{
  // turn off the UP indicator light, sound the
  // arrival bell and stop all motors
  // ...
  // switch to the next state
  delete this;
  return new StateOnSecondFloor;
}

ElevatorState* StateOnSecondFloor::DOWNbutton()
{
  // turn on the DOWN indicator light and start
  // the DOWN motor
  // ...
  // switch to the next state
  delete this;
  return new StateGoingDown;
}

ElevatorState* StateGoingDown::Arrived()
{
  // turn off the DOWN indicator light, sound the
  // arrival bell and stop all motors
  // ...
  // switch to the next state
  delete this;
  return new StateOnFirstFloor;
}
```

Listing 2 **-** The code used by the elevator system example.

Notice the null destructor in each class. The destructor is declared virtual to allow the statements

```
delete this;
```

that appear in the various classes to polymorphically delete the right kind of object **-** not merely an ElevatorState object **--** because the **this** pointer references objects derived from ElevatorState. Listing 3 shows a short program that uses the ElevatorState class hierarchy. The program displays the current system state, then prompts the user for an input event. By designating events with integer values, the program classifies the events and invokes a polymorphic event handler using the function **ProcessNextEvent**.

```
#include <iostream.h>

// include the ElevatorState class hierarchy declaration

void PromptForNextEvent(ElevatorState& state,
                  int& event)
{
  // prompt for the next input event
  cout << endl << "Current State = "
       << state.Description() << endl;
  cout << "Event types:" << endl
       << "1 - UP button" << endl
       << "2 - Arrived at floor" << endl
       << "3 - DOWN button" << endl
       << "Next event type ? " << endl;
  cin >> event;
}

// Event classifier and handler
ElevatorState* ProcessNextEvent(ElevatorState& state,
                        int event)
{
  const int
    UPbutton   = 1,
    Arrived    = 2,
    DOWNbutton = 3;

  // identify the event type, and invoke an action
  // according to the event's classification
  switch (event) {

    case UPbutton:
      return state.UPbutton();

    case Arrived:
      return state.Arrived();

    case DOWNbutton:
      return state.DOWNbutton();
    default:
      return 0;
  }
}

void main()
{
  // set the initial state
  ElevatorState* currentState = new StateOnFirstFloor;

  // run the state machine
  do {

    // get the next input event
    int event;
    PromptForNextEvent(*currentState, event);

    // classify the event and execute it according
    // to the current state
    currentState = ProcessNextEvent(*currentState, event);

  } while (currentState);

  // clean up
  delete currentState;
}
```

Listing 3 **-** A short program that demonstrates the use of a polymorphic class hierarchy to implement a state machine.

There is no code to look up handler functions based on the system state. Once an event is classified in the function **ProcessNextEvent**, a single polymorphic function call is bound at runtime to the class object that represents the current state. State transitions are represented by class instantiations, which occur deep inside the class hierarchy.

<u>State Clusters and Super-States</u>

Although the elevator example is very simple, it shows the basic simplicity and robustness that an object-oriented approach offers. But for the OOP technique to be truly useful, it must scale upwards, to handle complex systems. In many systems, it is common for multiple states to share certain some behaviors in response to events. Consider the state diagram in figure 9.

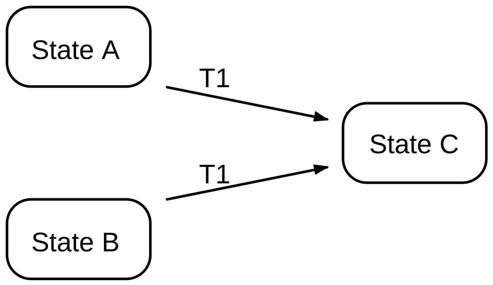

Figure 9 **-** Two objects sharing a common response to an event.

Both states A and B respond to event T1 by switching the system to state C. Since the states share a common behavior, at least as far as event T1 is concerned, it has been suggested [Harel] to abstract the common features out of A and B, and migrate them to a super-state Z. States A and B are then said to be a state cluster associated to the super-state Z, as shown in figure 10:

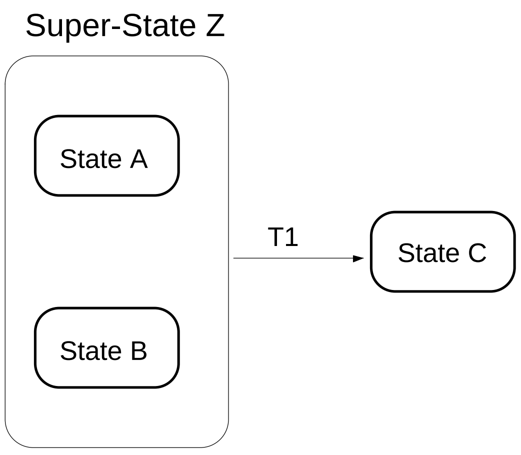

Figure 10 - A super-state with an associated state cluster.

Super-states simplify state complex state diagrams, because they isolate the common behaviors of states, reducing the number of transition arrows and the amount of code duplication. The super-state concept lends itself to an elegant object-oriented approach. State clusters become derived classes of the class representing the super-state, so figure 10 would become the class hierarchy shown in figure 11.

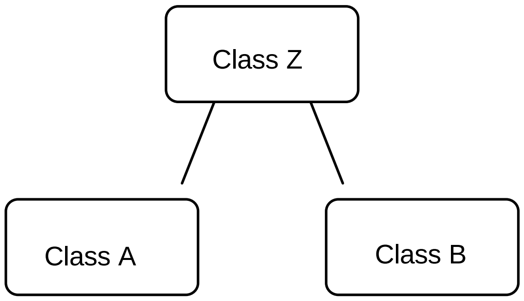

Figure 11 - Representing a super-state and state cluster using a class hierarchy.

Extending the concept, multiple levels of inheritance can be used to represented super-states of super-states, as shown in figure 12:

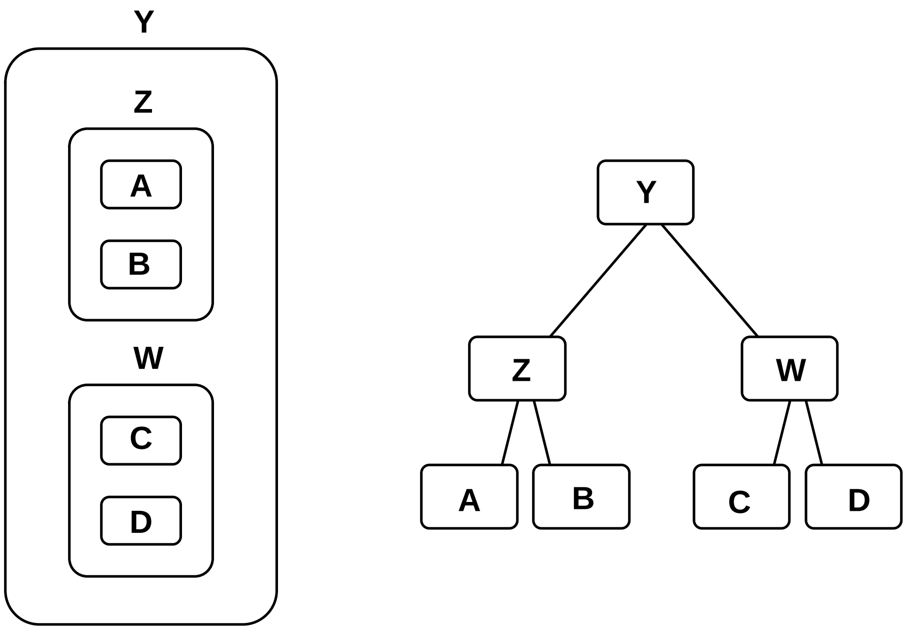

Figure 12 - Using multiple levels of inheritance to implement super-states of super-states.

The class hierarchy will grow by one level each time a super-state is added to the system. The hierarchy is also easy to work with, because it encodes in a single graph both the structure of the system (the states), and the way the system is actually implemented (the classes).

Handling time with a class hierarchy

Having shown a generic use of a class hierarchy, I'll now address a more specific situation: the handling a time. Real-time systems, in particular those whose input events are quasi-random, such as in user interfaces, the are often time limits posed on states. If the system doesn't leave a state within a given amount of time, the system times-out and switches to another state. Consider a bank teller machine. When you put your ATM card in, the system checks the card and requests you to enter you ID number. If you fail to do so, the system should time-out and take some default action, rather than wait forever. Timing out requires keeping track of the passage of time. If a system tracks time by issuing a tick event, the active state needs to process this event, and take action when a time-out occurs. Many states in a system will typically require time-out support, suggesting that the corresponding classes be part of a common sub-hierarchy of the system, as shown in figure 13:

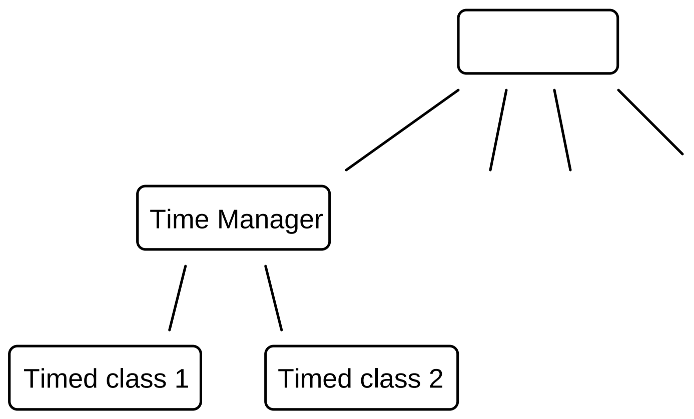

Figure 13 - Managing time with a class sub-hierarchy.

A time-out condition can be considered a special case of a more general process, namely the handling of boundary conditions for counted events. When a timed state is first entered, a timer will be set to an initial value. For each time event, the timer will be affected, until the timer reaches it limit - or boundary - condition. The management of the timer is assigned to the base class (super-state) called Time Manager in figure 13. When a boundary condition is met, the event is handled by the derived class.

Handling timed events is really only a matter of counting events. If the events are cars passing at an intersection, or the number of pills dropped into a bottle, it makes little difference. Considering timed events as a special case of counted events, one could implement a class hierarchy to include both, as shown in figure 14:

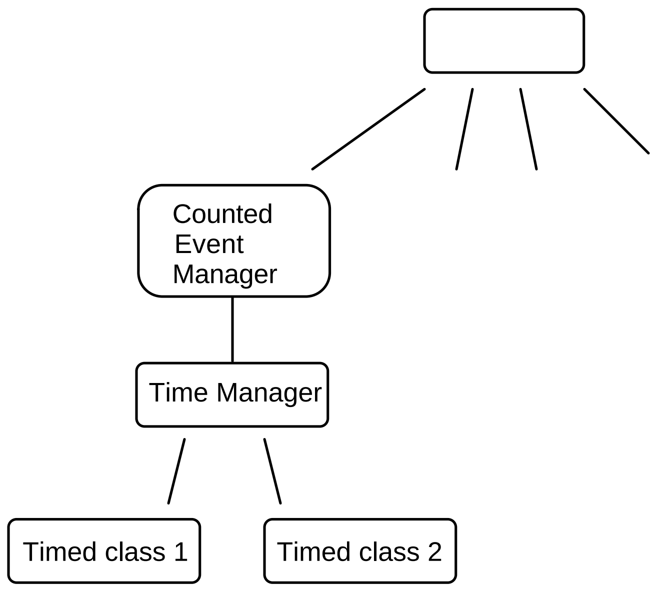

Figure 14 - Handling timed event as a special case of counted events.

The system might be implemented in C++ as shown in listing 4:

```
// the parent of the counted event sub-hierarchy
class RootClass {};

class CountedEvents: public RootClass {

protected:

  int count, limit;

  CountedEvents() {count = 0; limit = 0;}
  virtual ~CountedEvents() {}
  virtual RootClass& handleCountedEvent() {
    count += 1;
    return *this;
  }

};

class TimedEvents: public CountedEvents {

public:

  virtual ~TimedEvents() {}
  virtual RootClass& LimitReached() {
    return *this;
  }
  virtual RootClass& handleCountedEvent() {
    CountedEvents::handleCountedEvent();
    if (count == limit)
      return LimitReached();
    else
      return *this;
  }
};

class TimedClass1: public TimedEvents {

public:

  TimedClass1() {limit = 10;}
  virtual ~TimedClass1() {}
  RootClass& LimitReached();
};

class TimedClass2: public TimedEvents {

public:

  TimedClass2() {limit = 20;}
  virtual ~TimedClass2() {}
  RootClass& LimitReached();
};



RootClass& TimedClass1::LimitReached()
```

```
{
  // switch to state 2
  delete this;
  return *new TimedClass2;
}

RootClass& TimedClass2::LimitReached()
{
  // switch to state 1
  delete this;
  return *new TimedClass1;
}
```

Listing 4 - Implementing a timed-event manager with a class sub-hierarchy.

Overlapping states

When a state belongs to a state cluster, the super-class becomes responsible for handling certain events. When a state has more than one super-state, the parent super-states are said to be overlapping. Consider the case shown in figure 15:

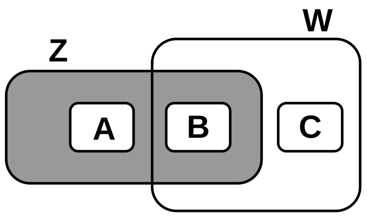

Figure 15 - Overlapping super-states.

Overlapping states can be expressed with class hierarchies using multiple inheritance. If state B is a cluster state of both states Z and W, it follows that in the class hierarchy, B will be multiply derived from classes Z and W, as shown in figure 16:

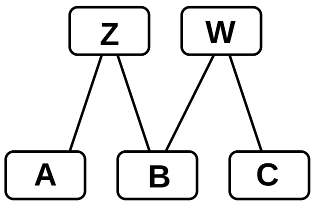

Figure 16 - Using multiple inheritance to implement overlapping states.

To show an application of this technique, consider the implementation of a simple program to blink the emergency lights in an automobile. The operator starts the blinking by pressing an ENABLE button, and stops the blinking by pressing a DISABLE button. The system can be described directly by its class hierarchy, as shown in figure 17.

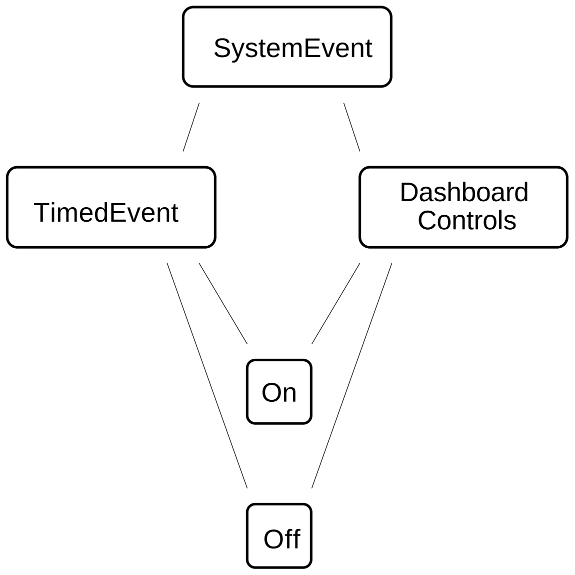

Figure 17 **-** A class hierarchy included in an automotive system to control the emergency lights.

The On and Off classes cycle the emergency lights during the blinking process. When the lights are disabled, the system is in the class (i.e. state) DashboardControls. When the ENABLE button is pressed, the system switches to the On class, which then kicks off the blinking sub-hierarchy. When the On class times out, the system switches to the Off class. When the Off class times out, it switches to the On state. The process repeats indefinitely, until the operator presses the DISABLE button, which is handled by the DashboardControls class. The code for such a system is shown in listing 5.

```
class SystemEvent {

public:

  virtual ~SystemEvent() {}
  virtual SystemEvent* Tick() {return this;}
  virtual SystemEvent* EnableLights() {return this;}
  virtual SystemEvent* DisableLights() {return this;}
  virtual char* Description() {return "";}
};

class TimedEvent: public virtual SystemEvent {

protected:

  int tickCount;

public:

  TimedEvent(int value) {tickCount = value;}
  virtual ~TimedEvent() {}
  virtual SystemEvent* Tick();
  virtual SystemEvent* TimeOut() {return this;}
};

class DashboardControls: public virtual SystemEvent {

public:
  virtual ~DashboardControls() {}
  virtual SystemEvent* EnableLights();
  virtual SystemEvent* DisableLights();
  virtual char* Description() {return "IdleState";}
};

class On: public TimedEvent, public DashboardControls {

public:

  On() : TimedEvent(3) {}
  virtual ~On() {}
  SystemEvent* TimeOut();
  virtual char* Description() {return "OnState";}
};


class Off: public TimedEvent, public DashboardControls {

public:

  Off() : TimedEvent(3) {}
  virtual ~Off() {}
  SystemEvent* TimeOut();
  virtual char* Description() {return "OffState";}
};


SystemEvent* TimedEvent::Tick()
{
  if (--tickCount == 0)
    return TimeOut();

  else
    return this;
}


SystemEvent* On::TimeOut()
{
  delete this;
  return new Off;
}


SystemEvent* Off::TimeOut()
{
  delete this;
  return new On;
}


SystemEvent* DashboardControls::EnableLights()
{
  delete this;
  return new On;
}

SystemEvent* DashboardControls::DisableLights()
{
  delete this;
  return new DashboardControls;
}
```

Listing 5 - A C++ implementation for the control of a car's emergency blinking lights.

Listing 5 shows the use of virtual destructors at every level of the class hierarchy. As mentioned earlier, destructors are required to be polymorphic in order for a generic object to delete itself, without knowing exactly what kind of object it is.

Using multiple inheritance, the classes On and Off become trivial, and are in no way involved in the higher level handling of the ENABLE and DISABLE buttons. Using class sub-hierarchies can simplify large systems, putting order in the chaos of events and event diagrams.

Conclusion

I focused on the more practical aspects of the use of class hierarchies to manage states, by showing some C++ examples. Other languages, such as Smalltalk and Objective-C yield equivalent implementations. In fact, there seem to be only two major requirements of a general-purpose programming language to support OO state machines as described: late binding and multiple inheritance. I have only scratched the surface of object-oriented state machines in this article, but hopefully enough to encourage others to explore other OO refinements, such as hierarchical OO state machines, the use of classes with Augmented Transition Networks and the dynamics of multiple state objects built from different class hierarchies, to name only a few.

References

Rumbaugh, James et al. *Object-Oriented Modeling and Design*, Englewood Cliffs, N. J., Prentice Hall, 1991, pp-84-114.

Booch, Grady. *Object Oriented Design with Applications*, Redwood City, Calif, Benjamin/Cummings Publishing Co, Inc., 1991, pp-167-180.

Harel, David. "Statecharts: a visual formalism for complex systems.", *Science of Computer Programming*, Volume 8 (1987), pp 231-274.

Shlaer, Sally and Mellor, Stephen. *Object Life Cycles: Modeling the World in States*, Englewood Cliffs, N. J. Yourdon Press, 1990.