The Art of State

Representing the State of an Object in C++

Charles Weir
© Object Designers Limited, June 1994

Introduction: The Toggle Class

Objects have state. This is a key principle of object orientation. But what exactly does it mean? The simplest interpretation is to say that each object contains data that affects its behaviour. In C++ terms we say that each instance of a class has data members; the behaviour of each instance depends on these data members.

Consider for example an instance of a String class: the behaviour of that instance in a print function (that is, the characters printed) depends on its state (the character data it references). In a normal function to print a string, the code makes no particular decisions based on the string contents. Supposing, however, there was a need to handle a special case, the null string, which must be printed as "*null*". In this case, the member function makes a *decision* based on the state; this is typically represented as an "if" or "switch" statement. This article examines some techniques to implement and simplify such decisions, and examines how they are affected by inheritance.

For the purposes of this article, we are going to examine a very simple class - one of the simplest which still makes decisions based on its state. This class is a toggle: it is either "on" or "off", and has a function (`doSwitch`) which toggles it from one state to the other. It also knows how to print itself to a stream (`operator<<`).

Let us start with a simple and obvious implementation (Figure 1). Its only data member is the character string to output. As an implementation this may not win any prizes, but it satisfies the requirements and the external interface.

```
class Toggle {
public:
   Toggle()          : outputString( "off" ) {};
   void doSwitch()
   { outputString = (strcmp( outputString, "off" ) == 0)
      ? "on" : "off"; }

friend  ostream& operator<<( ostream& out, const Toggle& t)
   { out << t.outputString;   return out; };

private:
   const char* outputString;
};
```

Figure 1: Toggle Implementation using internal data

This implementation represents the state of an instance using the values of its data - in this case, the characters in the string. Its member functions make decisions based only on the values of the instance data.

State Transition Diagrams

This approach is fine for simple classes. However what happens when there are more variables involved in defining the state of each instance? Consider, for example, a class representing a modem. This class, `Modem`, may have many different data members (including perhaps buffers, phone numbers, flags, and time-out counts), and between them they allow for an infinite number of possible states for any given instance. Yet as far as processing is concerned, only certain combinations and ranges of values are important. For example, a simple implementation might have the states shown in Figure 2.

Idle	No connection or activity.
Dialing	Sending pulses or tones to the telephone system
Waiting for connection	Waiting for a message back from the remote system to say it's accepted the link.
Connected	Connection established; can send and receive data.
Terminating	Has initiated a hang-up, and are waiting for the line to drop.
Disconnected	A termination signal or a line fault has been detected; the user has not yet been notified.

Figure 2: Possible States for a Modem Object

An instance of this Modem class will *transition* from one state to another in response to external *events*; in C++ these events are presented to it as member function calls. We can represent the possible situations and significant events with a picture - a state transition diagram. Here, the boxes represent states and the arrows represent transitions. We annotate the lines with the event (the function) which causes the transition. The event that creates the instance (the constructor) is shown as the transition from a blob.

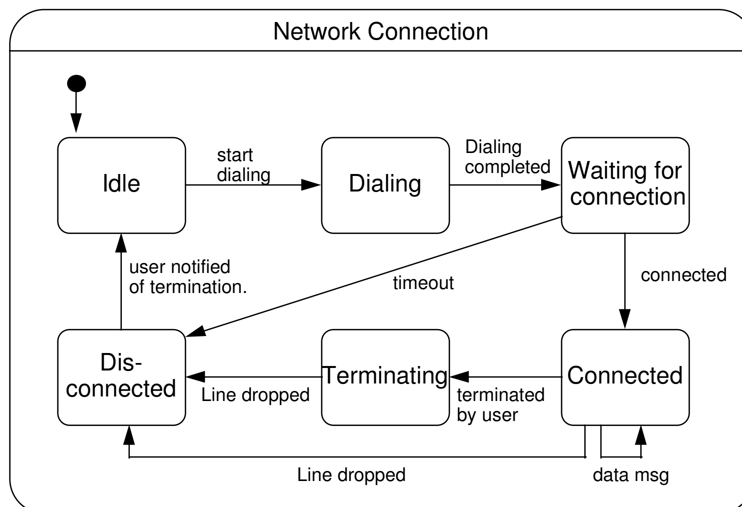

Figure 3: State Chart for a Complicated Object

This is a major simplification. From an infinite number of possible states, we have expressed much of the interesting functionality of the class in just six. We should like to reflect this simplification somehow in the code? But how?

To tackle this question while keeping the example code simple, we'll return to the Toggle class; Figure 4 shows its state transition diagram.

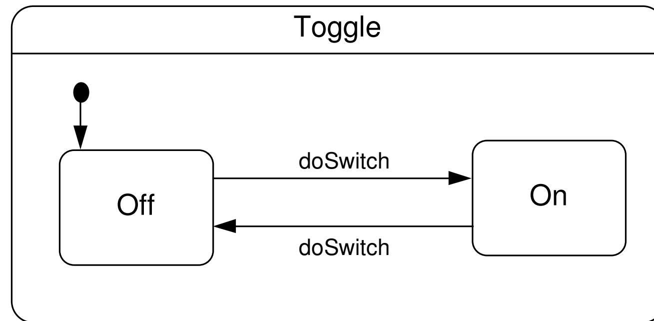

Figure 4: State Transition Diagram for Toggle

It is pretty trivial, certainly, but it does highlight the salient behaviour of the class. In particular, it shows how the behaviour of an instance in the `doSwitch` function varies according to its state.

How are we to express this in C++? There is a direct, easy, and effective solution: we identify states as integer numbers, and have a *state variable* as an instance data member. The value of this variable will represent the state of the instance. C++ makes it easy to assign names to unique numbers using the enumerated data type (`enum`).

Figure 5 shows the resulting implementation. Where the behaviour of a member function depends on the state, it checks the state variable; to change state, it simply changes that variable. Note how we prevent name clashes (another class might use the name "`ON`", for example), we declare the enumerated type within the class definition rather than outside. Observe also the sanity check using a function to terminate - here, we use `abort()` - if the state variable is invalid.

```
class Toggle {
public:
   Toggle()          : state( OFF ) {};
   void doSwitch();

friend  ostream& operator<<( ostream& out, const Toggle& t)
   { out << ((t.state == OFF) ? "off" : "on");  return out; };

private:
   enum State { OFF, ON };
   State state;
};

void Toggle::doSwitch()
{
   switch (state)
   {
   case ON:  state = OFF; break;
   case OFF: state = ON; break;
   default:  abort();
   }
}
```

Figure 5: Implementation of Toggle using a state variable

The Effect of Inheritance: Conformance

How does inheritance affect the representation of an object's state? Suppose that our project requires the Toggle class as a base class for other classes - so that we can use a reference or pointer to Toggle as a pointer to instances or derived classes. What implications does this have for the implementation of Toggle?

In a good design, any classes which inherit from Toggle must be *conformant*. What this means is that an instance of a derived class will appear to behave in the same way as would an instance of Toggle itself. In particular, they must appear to have the same state behaviour. For example, if we call the `doSwitch` function twice on any instance of a derived class, we expect it to appear to

be in the same state (ON, or OFF) afterwards, as before. This severely limits how we can use inheritance. An example of a class that conforms to the Toggle interface might be a simple motor controller: this has a button (function) which toggles it on and off, and another which toggles fast and slow. Figure 6 shows its state transition diagram.

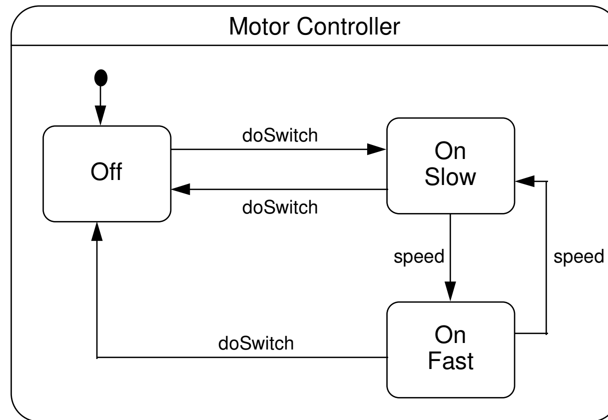

Figure 6: Simple State Diagram for Motor Controller

We can simplify this diagram, and make it show clearly its conformance with the state transition diagram for Toggle, using Hurel's notation to nest states. This is the notation which is used by Rumbaugh's method, and enhanced in Object Designers' method, Syntropy. Figure 6 shows the result.

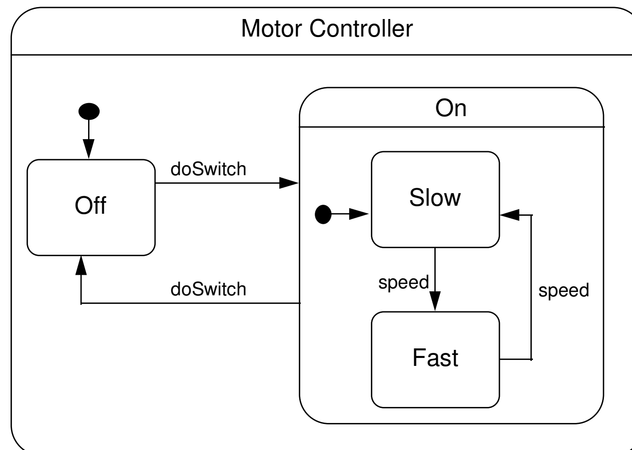

Figure 7: Motor Controller State Diagram using Hurel's Notation.

Implementation of Inheritance

Coding this derived class, MotorController, presents some problems. We can access the state values in the base class, certainly, and we can extend the enumeration of states. But the state "ON" no longer exists - instead, it is divided in two.

We have two possible options to express this division: we can add more possible states for the existing state variable (`Toggle::state`); or we can create a second state variable to represent the new states. The latter option appears to have an advantage: code written for the Toggle class can read only the first state variable and understand all of the values it takes. However, in practice, the business of setting two variables rather than one makes this solution very messy: we have to decide how to handle combinations of the two variables that may not be legal (OFF and FAST, for example).

So instead we extend the states taken by the state variable already in class Toggle. This is straightforward. To extend the enumeration of states in the derived classes, we introduce a

dummy state, `LAST`, into the enumeration for Toggle. We also make other changes to Toggle to allow derivation: a virtual destructor ensures that deleting a `Toggle*` pointer will invoke the correct destructor; both the `doSwitch` and output functions need to be virtual (note the technique of creating a new function for the output operator to call); and the state variable and enumeration are now `protected`, to grant access to the derived classes.

Ideally, of course, in writing the code for MotorController, we don't want to have to re-write any of the code for Toggle's functions. To allow this, we need to restrict Toggle so that it accesses its state variable only through virtual functions. There are several possible ways to do this, but in practice the approach which gives the most flexibility is to define a separate function to set each state, and a separate function to retrieve the state *as seen by each class.* The latter functions are `stateForToggle()` and `stateForMotorController()` in the example. This approach means that there is no need to redefine in MotorController any of the important member functions in Toggle. Figure 8 shows the resulting code.

```
class Toggle {
public:
    Toggle() : state( OFF ) {};
    virtual void doSwitch();

friend ostream& operator<<( ostream& out, const Toggle& t)
    { t.output( out ); return out; }

protected:
    virtual void output( ostream& ) const;

    enum State { OFF, ON, LAST };
    virtual void on()  { state = ON; };
    virtual void off() { state = OFF; };
    virtual int stateForToggle() const { return state; }

    int state;
};

void Toggle::output( ostream& out ) const
{
    out << ((stateForToggle() == OFF) ? "OFF" : "ON");
}

void Toggle::doSwitch()
{
    switch (stateForToggle())
    {
    case ON:  off();             break;
    case OFF: on();              break;
    default:  abort();
    }
}

class MotorControl : public Toggle {
public:
    virtual void changeSpeed();

protected:
    enum State { SLOW=Toggle::LAST, FAST, LAST };

    virtual void fast() { state = FAST; }
    virtual void slow() { state = SLOW; }
    virtual void on()   { state = SLOW; }    // Redefinition
    virtual int stateForToggle() const;      // Redefinition
    virtual int stateForMotorControl() const
            { return state; }
    virtual void output( ostream& ) const;   // Redefinition
};
```

```
int MotorControl::stateForToggle() const
{
    switch (stateForMotorControl())
    {
    case FAST: case SLOW: return ON;
    case ON:              abort();
    default:              return Toggle::stateForToggle();
    }
}

void MotorControl::changeSpeed()
{
    switch (stateForMotorControl())
    {
    case SLOW:  fast();           break;
    case FAST:  slow();           break;
    case OFF:   /* No action */  break;
    default:    abort();
    }
}

void MotorControl::output( ostream& out ) const
{
    switch (stateForMotorControl())
    {
    case SLOW:  out << "SLOW";          break;
    case FAST:  out << "FAST";          break;
    default:    Toggle::output( out ); break;
    }
}
```

Figure 8: Implementation of Toggle and Motor Controller

It has taken some significant changes to make the class Toggle suitable for derivation. Other approaches might use less code, but at the cost of a less robust - and less clear - representation.

Conclusion

To summarise, then, this article has introduced three key concepts:

- Objects have state, which can often be represented effectively using a *state transition diagram*.
- Inheritance is most useful when the derived classes are *conformant*, so that they have compatible state transition diagrams.
- We can encode the diagrams directly in C++ using a *state variable*. If we control access to this with suitable member functions, we can make this work well under inheritance.

In practice, we might seldom need the ability to extend state behaviour under inheritance; we may often define a simple state variable for a class; and we will always want to have state transition diagrams available as a tool when we need them. These techniques give us a way to control and manage the complexity of any individual class: we should use them.