Dmitry Babitsky

Hierarchical State Machine Design in C++

Translating statechart diagrams into C++

The Finite State Machine (FSM) is a design pattern in which actions are determined by events and the current context of the system. The driver code dispatches events to the FSM that forwards it to the current state. Functions processing the events decide what should be the next system state. After that, the process repeats.

The Hierarchical FSM is an extension of the FSM concept. In this case, any state can be a substate of some larger state. For example, the `vehiclesGreen` and `vehiclesYellow` states of a traffic light in Figure 1 may be substates of `VehiclesEnabled`.

Unlike traditional flowchart-based designs, the FSM design pattern is often more suitable for event-driven applications. It can help bring structure to code. In this article, I focus on translating FSM statechart diagrams into C++. This is a follow-up to Miro Samek's articles in *CUJ* [1].

There are many ways to implement state machines, including:

- Nested `switch()` statements over all states and over all events.
- State-transition tables; states on one axis, events on the other, and cells containing a function to execute and the next state.
- Dynamic tree-like structures traversed at runtime.

Miro presented an approach where `State` is a pointer to a `[member]` function, and `Event` is an enumerated integer. The Quantum Framework source code available with Miro's book [2] implements the hierarchy using a tree-like navigation from child to parent state and from parent to child that is rather complex. (The Quantum Framework is a minimal realization of an active object-based application framework designed specifically for embedded real-time systems; see http://www.quantum-leaps.com/).

I evaluated several approaches to FSM and, for any number of reasons, none met my needs. Consequently, I decided to build my own. My motivating factors for revisiting the subject was that I wanted to:

- Simplify the framework code and use. In fact, I prefer not to call it a "framework" at all—it's simply a way of doing things and a sample implementation. The smaller and shorter it is, the more likely people will want to use and expand it.
- Get rid of `switch()` statements over events in every state function.
- Avoid macros because I believe they unnecessarily obfuscate the code.

The solution I present here is based in part on the `State` design pattern [3] in which `State` is a class and `Event` a pointer to a function in that class. There are a number of advantages to this approach; foremost among them that hierarchy in C++ is naturally expressed with inheritance. By making state classes that can derive from each other, the need to create artificial hierarchy with tree-like code is eliminated.

The language automatically calls the most-derived virtual function processing the event. The need to handle events in every state function with `switch()` code is eliminated, making code easier to read and maintain. Events are functions, not integers. Moreover, there's no need to have the `isHandled` flag in state methods. And since states are classes, they can store their own mini-states; for instance, how many times has this state been entered, how long on average has the system spent in this state, and so on. This is more difficult to achieve if the state is a function. However, the state of the entire system is still stored in the class implementing the state machine. State classes do not share data. The entire "framework" consists of one class and fits into some 70 lines of Standard C++ (available at http://www.cuj.com/code/).

As a proof of concept, I implemented Miro's Pelican and Calculator state machine diagrams (see [1] and [4]) and a text file processor that removes extra blanks and newlines from files.

Listing 1 illustrates the idea using the fictitious Aggregator FSM. Each event function returns a pointer to the next state object, which has been created on the stack. `State` classes do not have to be nested inside the main `Aggregator` class; it just more clearly shows affiliation.

Listing 2 is the complete implementation of the `Fsm` class. The `STATE` class passed as the template parameter is expected to implement `on_enter()` and `on_exit()` functions that can be used to initialize the hierarchy. Whether they actually do anything is irrelevant and is up to the implementer.

I spent time deciding whether all state objects should be created on the stack or dynamically allocated, what model the `Fsm` class should support, and the pattern of usage. *Design Patterns* [3] addresses this issue.

The major advantage of dynamically creating states is that constructors and destructors can be used for initialization and cleanup. The language automatically generates code to initialize and cleanup base and derived states as part of the constructor and destructor call hierarchy and you don't need to do anything special for it. If the states have been created, you can use `on_enter`/`on_exit` functions instead

Dmitry Babitsky is a developer at Goldman Sachs. He can be reached at dbabits@hotmail.com.

for similar effect. The disadvantage of dynamic creation is that it is more expensive and error prone.

There is a way to make use of constructors/destructors, even in the case of preallocated state objects. I experimented with using `placement new`, which invokes the constructor and just returns a pointer to an existing object, coupled with explicitly calling the destructor. This works. The construction/destruction chain is invoked correctly, but the destructor is called twice—once more when the scope is exited. It is possible, of course, to make it reentrant, but that means you have to code it in a special way. In the end, I ruled against that idea; using `on_enter`/`on_exit` was less of a headache.

The most difficult and nonobvious part of the hierarchical FSM is what Quantum Framework calls LCA (Least Common Ancestor) state, and it has to do with the way `on_enter()`/`on_exit()` functions are invoked. Remember, they work like constructors and destructors, but there's more to it. The state classes form a tree and during the transition, `state::on_enter()` should only be invoked if you are not already in this state by way of inheritance or, in other words, are not coming in from a child state. Similarly, `on_exit()` should only happen if our next state is not a child of the state we are transitioning from.

And `on_exit()`/`on_enter()`, for everything in between the source and the destination states, must be chained appropriately to mimic the way constructors/destructors work: Every `on_enter()` must call its parent's class `on_enter()` first, and every `on_exit()` must call its parent's `on_exit()` last.

Listing 1

```
class Aggregator;

class NO_VTABLE State:public IState_base<Aggregator,State>
{
public:
  virtual State* event1(Aggregator*,const boost::any&){return this;}
  virtual State* event2(Aggregator*,const boost::any&){return this;}
};
/*******************************************************************************/
     IState_base<Aggregator>
          |
        State    <<-app-specific, prototypes every event handled by the system
        / | \
       /  |  \
      /   |   \
state3  state4 state1 <<each state implements events that it cares about
                |
               state2
*******************************************************************************/
class Aggregator:public Fsm<Aggregator,State>{
    class state1:public State{
        virtual State* event1 (Aggregator* p,const boost::any&)
        {return &p->m_state2;}
        virtual State* event2 (Aggregator* p,const boost::any&){return this;}
    } m_state1;

    class state2:public state1{
        virtual State* event1(Aggregator* p,const boost::any&)
        {return &p->m_state1;}
    } m_state2;
};

Aggregator::Event get_event(boost::any& arg){
   return (random)? State::event1 : State::event2;
}

main(){
 for(;;){
   boost::any arg;
   Aggregator::Event e=get_event(arg);
   State* next_state=a.dispatch(e,arg);
   if(0==next_state) break;
 }
}
```

It is important to understand that, for purposes here, I'm not talking about object layout or memory representation, but exclusively about the class hierarchy. Look at the Pelican diagram in Figure 1 as an example [5]. Both `vehiclesEnabled` and `pedestriansEnabled` derive from `operational`. If these classes were to be created and destroyed dynamically, there would be two objects with an `operational` in each of them. So, when transitioning from `vehiclesYellow` to `pedestriansWalk`, one `operational` would be destroyed and another one created—clearly not what's intended. You never leave `operational` state until shutdown. Instead, with such transition, you would want `vehiclesYellow`, then `vehiclesEnabled` destroyed, `operational` left alone, then `pedestriansEnabled` and `pedestriansWalk` constructed, in that order. Unfortunately, this is not possible with dynamic activation. This is the reason why in the end, after much pondering, I had to abandon dynamic state creation via `new`, even though it worked fine for these sample programs.

This leaves you with preallocating all state objects on the stack. The next thing to decide was how to make the behavior just described happen and with as little pain as possible. My first idea was this:

```
while(!next_state->derives_from(curr_state))
    curr_state=curr_state->on_exit() //returns parent state.
```

Unfortunately, it's not so easy with `on_enter()` because you must first find all parents up to LCA, then call them top-down—exactly the kind of code I wanted to avoid. This is more or less how the Quantum Framework does it and it is complex. (That said, the Quantum Framework is designed to also work for C.)

My solution was somewhat of a compromise. First, I had to use RTTI to make `derives_from()` work. While I'm generally against it, in this case it's probably better than handwritten loops anyway. And second, there's some burden being placed on implementers in that they have to chain `on_enter()`/`on_exit()` correctly. It's not too bad; the template is subsequently provided.

Bear in mind that you do not have to provide these functions for every state in your hierarchy—only if a state has some initialization/cleanup to do, just as you would use constructors/destructors.

I would be interested in any ideas that achieve the same results without RTTI (and without macros). Also, note that `derives_from()` is a template function in a class template and to use it, I had to abandon Visual C++ 6. It will only compile with Version 7; otherwise, the same line can be restated much less elegantly. Essentially, all it does is `dynamic_cast<>`, plus some tracing output, but I believe it is still worth having because it makes the intent more explicit. I did not try other compilers, but it's all Standard C++, so conformant compilers should work.

I keep the version of the code that lets you dynamically create state classes. I keep it because of some other techniques that were developed for it that I would like to have around for the future. In short, when using it, state-event handlers do not themselves create the next state dynamically because that makes a new state before the old one is destroyed. Instead, they return the next state's factory (a generic template), invoked when appropriate by the `Fsm` class. (If you're interested, look at the fsm_test project at http://www.cuj.com/code/, although the part involving initialization and cleanup is not complete.)

With the Pelican intersection diagram in Miro's example, the system should be either in a `vehiclesEnabled` or `pedestriansEnabled` state at any given time, but never in both (that would be life-threatening). So when the system transitions from one to another, should you first call `current_state->on_exit()`, or `new_state->on_enter()`? I decided that `on_exit` should happen first, partly because this is how the Quantum Framework does it and I wanted to stay compatible. However, the system's states have to be designed in such a way that transition is atomic. But, if you assume that exiting a state never throws, but entering may, the system could be left in an inconsistent state. The exception safety part needs to be revisited.

Listing 2

```
namespace boost{class any;}

template <class DERIVED_FSM,class STATE>
class NO_VTABLE Fsm{
public:

    //for stack allocated states
    //note that constructors/destructors should not be used in this case,
    //instead rely on on_enter/on_exit
    typedef typename STATE* (STATE::*Event)(DERIVED_FSM*,const boost::any&);

    STATE* dispatch(Event e, const boost::any& arg){
        if(curr_state && e){
            STATE* next_state=(curr_state->*e)(that(),arg);
            if(next_state!=curr_state) transition(next_state);
        }
        return curr_state;
    }

    void transition(STATE* next_state){
        if(curr_state==next_state) return;

        if(curr_state) curr_state->on_exit(that(),next_state);
        if(next_state) next_state->on_enter(that(),curr_state);

        curr_state=next_state;
    }

protected:
    Fsm():curr_state(0){}
    virtual ~Fsm(){}
private:
    STATE* curr_state;
    inline DERIVED_FSM* that(){ return static_cast<DERIVED_FSM*>(this);}
};

template<class FSM,class STATE>
class NO_VTABLE IState_base{
protected:
    virtual ~IState_base(){}
public:
    virtual void on_exit (FSM*,STATE* new_state){}
    virtual void on_enter(FSM*,STATE* old_state){}

    #if _MSC_VER < 1200
     #error derives_from() will not compile in VC++ before version 7./
       create this as stand-alone fn then.
    #endif

    template<class T>
    bool derives_from(T*){
        const char* derives= 0!=dynamic_cast<T*>(this)?"YES":"NO";
        printf("\n%s derives from %s ?-%s",
               typeid(*this).name(),typeid(T).name(),derives);
        return 0!=dynamic_cast<T*>(this);
    }
};
```

State-specific variables are members of a state class and not of a larger `Fsm` class. In Pelican, for example, `isPedestrianWaiting_` only makes any difference when the machine is in `vehiclesGreen` state; that's why it's defined there.

Using Hierarchical FSM

To use the Hierarchical FSM:

1. Create the state class that derives from `IState_base` and describes all events that your system handles. All event methods in this class should return `this`. That means no transition and is the default action to take if the individual state does not care about this event:

```
template<class FSM>
     class NO_VTABLE
        State:public IState_base<FSM,State>{
        State<FSM>*,const boost::any&)
          {return this;}
}
```

2. Create the main state machine class like this:

```
class Pelican : public Fsm<Pelican,
                      State<Pelican>>{}
```

3. Create individual state classes that must directly or indirectly derive from `State` and override event functions that this state cares about. Such classes may or may not be nested inside the main class (in C++ the difference is purely notational).
4. Implement this function and return a pointer to the next state:

```
class pedestriansWalk:public
                     pedestriansEnabled{
    virtual State<Pelican>*
                     timeout(Pelican*
                     p,const boost::any&){
     return &p->m_pedestriansFlash;
    }
};
```

5. If the event function determines that the machine should stay in the same state, it should return `this`.
6. If the event function determines that it's time to exit the state machine, it should return 0. It is up to the driver code to take appropriate action.
7. If you need to provide initialization and/or cleanup, implement `on_enter`/`on_exit` for your state. Here's the template:

```
void state::on_enter(Fsm* p,
                     State* old_state)
{
  if (old_state &&
      old_state->derives_from(this))
    return;
  super::on_enter(p,old_state);
  //Your code
}

void state::on_exit(Fsm* p,
                     State* new_state)
{
  if (new_state &&
      new_state->derives_from(this))
    return;
  //Your code
  super::on_exit(p, new_state);
}
```

Although it looks like the first line exits when the condition is `true`, it is not possible to make this check in the caller and not make the call because of the chaining of these functions.

Conclusion

I presented yet another way of translating standard statechart diagrams into C++ code. The hierarchy in the state machine is achieved by using C++ inheritance and polymorphism to handle the same event differently based on the context (or state) of the system. The state is an instance of a class derived from a common root that defines all events that this `Fsm` will handle. The type of object pointed to by a root state pointer determines the current state. All events are dispatched through this pointer.

I extended the Calc state machine to respond to the `on_equals` event from the `opEntered` state (the statechart should reflect that). This is how the standard Windows calculator works.

Listing 3

```
#if !defined DMITRY_PELICAN_JAN_2004
#define DMITRY_PELICAN_JAN_2004

#include "../include/fsm.h"
#include "../include/announcer.h"

class Pelican;

//this just declares all possible events for this application,
//across all states.
template<class FSM>
class _declspec(novtable) State:public dbabits::IState_base<FSM,State>
{
public:
    //no transition by default
    virtual State<FSM>* off(FSM*,const boost::any&) {return this;}
    virtual State<FSM>* pedestrianWaiting(FSM*,const boost::any&){return this;}
    virtual State<FSM>* timeout(FSM*,const boost::any&){return this;}
};

enum PelicanTimeouts {              // various timeouts in milliseconds
   VEHICLES_GREEN_MIN_TOUT = 6000,  // minimum green for vehicles
   VEHICLES_YELLOW_TOUT    = 2000,  // yellow for vehicles
   PEDESTRIANS_WALK_TOUT   = 4000,  // walking time for pedestrians
   PEDESTRIANS_FLASH_TOUT  = 500,   // flashing interval for ped.
   PEDESTRIANS_FLASH_NUM   = 3,     // number of flashes for ped.
};

class Pelican : public dbabits::Fsm<Pelican,State<Pelican> > {
public:
   Pelican();
   void init();

    class Off:public State<Pelican>,announcer<Off>
    {
    protected:
        virtual void on_enter(Pelican*,State<Pelican>* old_state);
    } m_Off;

    class operational:public State<Pelican>
                    ,announcer<operational>
    {
        virtual State<Pelican>* off(Pelican*,const boost::any&);

    public:
        enum VehiclesSignal { RED, YELLOW, GREEN };
        enum PedestriansSignal { DONT_WALK, BLANK, WALK };

        static void signalVehicles(enum VehiclesSignal sig);
        static void signalPedestrians(enum PedestriansSignal sig);
    };

    class vehiclesEnabled:public operational,announcer<vehiclesEnabled>
    {
    protected:
        virtual void on_enter(Pelican*,State<Pelican>* old_state);
        virtual void on_exit(Pelican*, State<Pelican>* new_state);

    };

    class vehiclesGreen:public vehiclesEnabled
                       ,announcer<vehiclesGreen>
    {
    protected:
        virtual void on_enter(Pelican*,State<Pelican>* old_state);
        virtual void on_exit(Pelican*, State<Pelican>* new_state);

    private:
        virtual State<Pelican>* pedestrianWaiting(Pelican*,const boost::any&);
        virtual State<Pelican>* timeout(Pelican*,const boost::any&);

        BOOL m_isPedestrianWaiting;
    } m_vehiclesGreen;

    class vehiclesGreenInt:public vehiclesEnabled
                         ,announcer<vehiclesGreenInt>
    {
    protected:
        virtual void on_enter(Pelican*,State<Pelican>* old_state);
    private:
        virtual State<Pelican>* pedestrianWaiting(Pelican*,const boost::any&);
    } m_vehiclesGreenInt;

    class vehiclesYellow:public vehiclesEnabled,announcer<vehiclesYellow>{
    protected:
        virtual void on_enter(Pelican*,State<Pelican>* old_state);
        virtual void on_exit(Pelican*, State<Pelican>* new_state);
    private:
        virtual State<Pelican>* timeout(Pelican*,const boost::any&);
    } m_vehiclesYellow;

    class pedestriansEnabled:public operational,announcer<pedestriansEnabled>
    {
    protected:
        virtual void on_enter(Pelican*,State<Pelican>* old_state);
    } m_pedestriansEnabled;

    class pedestriansWalk:public pedestriansEnabled,announcer<pedestriansWalk>
    {
    protected:
        virtual void on_enter(Pelican*,State<Pelican>* old_state);
        virtual void on_exit(Pelican*, State<Pelican>* new_state);

    private:
        virtual State<Pelican>* timeout(Pelican*,const boost::any&);
    } m_pedestriansWalk;

    class pedestriansFlash:public
pedestriansEnabled,announcer<pedestriansFlash>{
    protected:
        virtual void on_enter(Pelican*,State<Pelican>* old_state);
        virtual void on_exit(Pelican*,State<Pelican>* new_state);

    private:
        int  pedestrianFlashCtr_;
        virtual State<Pelican>* timeout(Pelican*,const boost::any&);
    } m_pedestriansFlash;

   static BOOL CALLBACK dlgproc(HWND hwnd, UINT iEvt, WPARAM wParam,
     LPARAM lParam);
};

#endif //DMITRY_PELICAN_JAN_2004
```

Figure 1: ***State diagram of the Pelican crossing.***

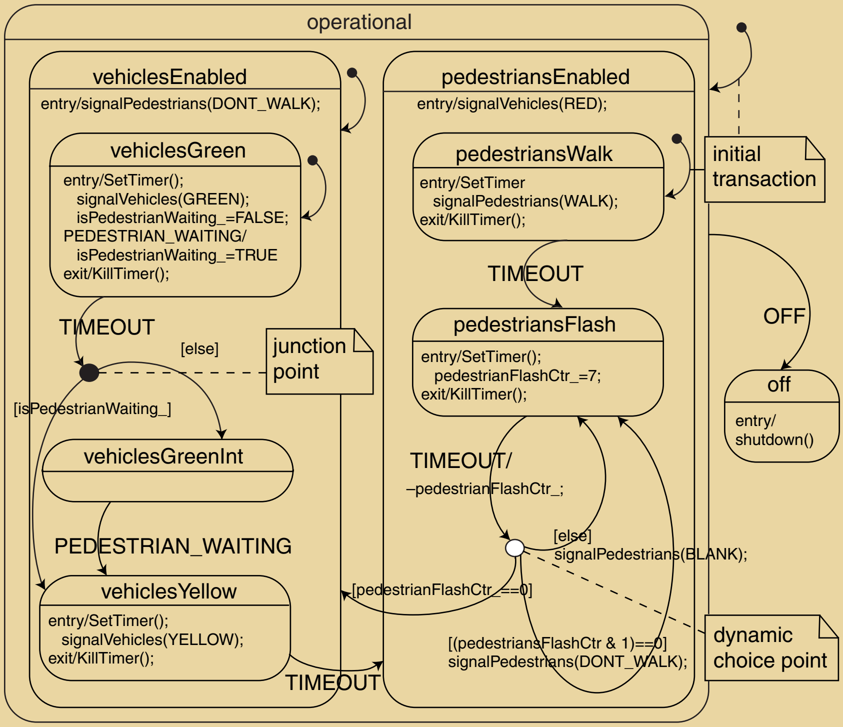
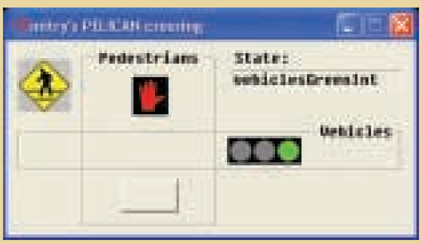


Figure 2: ***A sample run that shows what's going on, generated by the announcer class template.***




```
class Pelican::pedestriansFlash +(this=0x411bb9)->
  vehiclesEnabled::on_enter->vehiclesGreen::on_enter
class Pelican::vehiclesGreenInt derives from class Pelican::vehiclesGreen ?
  -NO->vehiclesGreen::on_exit
class Pelican::vehiclesGreenInt derives from class Pelican::vehiclesEnabled ?
  -YES-stay in vehiclesEnabled
class Pelican::vehiclesGreen derives from class Pelican::vehiclesGreenInt ?-NO
class Pelican::vehiclesGreen derives from class Pelican::vehiclesEnabled ?
  -YES-already in vehiclesEnabled->vehiclesGreenInt::on_enter
class Pelican::vehiclesYellow derives from class Pelican::vehiclesEnabled ?
  -YES-stay in vehiclesEnabled
class Pelican::vehiclesGreenInt derives from class Pelican::vehiclesYellow ?-NO
class Pelican::vehiclesGreenInt derives from class Pelican::vehiclesEnabled ?
  -YES-already in vehiclesEnabled->vehiclesYellow::on_enter
class Pelican::pedestriansWalk derives from class Pelican::vehiclesYellow ?
  -NO->vehiclesYellow::on_exit
class Pelican::pedestriansWalk derives from class Pelican::vehiclesEnabled ?
  -NO->vehiclesEnabled::on_exit
class Pelican::vehiclesYellow derives from class Pelican::pedestriansWalk ?-NO
class Pelican::vehiclesYellow derives from class Pelican::pedestriansEnabled ?
  -NO->pedestriansEnabled::on_enter->pedestriansWalk::on_enter
class Pelican::pedestriansFlash derives from class Pelican::pedestriansWalk ?
  -NO->pedestriansWalk::on_exit
class Pelican::pedestriansWalk derives from class Pelican::pedestriansFlash ?-NO
class Pelican::pedestriansWalk derives from class Pelican::pedestriansEnabled ?
  -YES-already in pedestriansEnabled->pedestriansFlash::on_enter
class Pelican::vehiclesGreen derives from class Pelican::pedestriansFlash ?
  -NO->pedestriansFlash::on_exit
class Pelican::pedestriansFlash derives from class Pelican::vehiclesGreen ?-NO
class Pelican::pedestriansFlash derives from class Pelican::vehiclesEnabled ?
  -NO->vehiclesEnabled::on_enter-
```

Listing 4

```
#include "stdafx.h"
#include "resource.h"
#include "pelican_static.h"

HWND hWnd_=0;

void Pelican::Off::on_enter(Pelican*,State<Pelican>* old_state){
    if (old_state && old_state->derives_from(this)) {
        printf("-already in Off");
        return;
    }
    printf("->Off::on_enter");
    EndDialog(hWnd_, 0);
}

void Pelican::vehiclesEnabled::on_enter(Pelican* p,State<Pelican>* old_state){
    if (old_state && old_state->derives_from(this)) {
        printf("-already in vehiclesEnabled");
        return;
    }
    operational::on_enter(p,old_state);

    printf("->vehiclesEnabled::on_enter");
    signalPedestrians(DONT_WALK);
}

void Pelican::vehiclesGreen::on_enter(Pelican* p,State<Pelican>* old_state){
    if (old_state && old_state->derives_from(this)) {
        printf("-already in vehiclesGreen");
        return;
    }
    vehiclesEnabled::on_enter(p,old_state);

    printf("->vehiclesGreen::on_enter");

    m_isPedestrianWaiting=FALSE;
    SetDlgItemText(hWnd_, IDC_STATE, "vehiclesGreen");
    SetTimer(hWnd_, 1, VEHICLES_GREEN_MIN_TOUT, 0);
    signalVehicles(GREEN);
}

void Pelican::vehiclesGreenInt::on_enter(Pelican* p,State<Pelican>* old_state){
    if (old_state && old_state->derives_from(this)) {
        printf("-already in vehiclesGreenInt");
        return;
    }
    vehiclesEnabled::on_enter(p,old_state);

    printf("->vehiclesGreenInt::on_enter");

    SetDlgItemText(hWnd_, IDC_STATE, "vehiclesGreenInt");
    signalVehicles(GREEN);
}

void Pelican::vehiclesYellow::on_enter(Pelican* p,State<Pelican>* old_state){
    if (old_state && old_state->derives_from(this)) {
        printf("-already in vehiclesYellow");
        return;
    }
    vehiclesEnabled::on_enter(p,old_state);
    printf("->vehiclesYellow::on_enter");

    SetDlgItemText(hWnd_, IDC_STATE, "vehiclesYellow");
    SetTimer(hWnd_, 1, VEHICLES_YELLOW_TOUT, 0);
    signalVehicles(YELLOW);
}

void Pelican::pedestriansEnabled::on_enter(Pelican* p,State<Pelican>* old_state){
    if (old_state && old_state->derives_from(this)) {
        printf("-already in pedestriansEnabled");
        return;
    }
    operational::on_enter(p,old_state);

    printf("->pedestriansEnabled::on_enter");

    signalVehicles(RED);
}

void Pelican::pedestriansWalk::on_enter(Pelican* p,State<Pelican>* old_state){
    if (old_state && old_state->derives_from(this)) {
        printf("-already in pedestriansWalk");
        return;
    }
    pedestriansEnabled::on_enter(p,old_state);

    printf("->pedestriansWalk::on_enter");

    SetDlgItemText(hWnd_, IDC_STATE, "pedestriansWalk");
    signalPedestrians(WALK);
    UINT_PTR t=SetTimer(hWnd_, 1, PEDESTRIANS_WALK_TOUT, 0);
}

void Pelican::pedestriansFlash::on_enter(Pelican* p,State<Pelican>* old_state){
    if (old_state && old_state->derives_from(this)) {
        printf("-already in pedestriansFlash");
        return;
    }
    pedestriansEnabled::on_enter(p,old_state);

    printf("->pedestriansFlash::on_enter");

    SetDlgItemText(hWnd_, IDC_STATE, "pedestriansFlash");
    SetTimer(hWnd_, 1, PEDESTRIANS_FLASH_TOUT, 0);
    pedestrianFlashCtr_ = PEDESTRIANS_FLASH_NUM*2 + 1;
}

void Pelican::vehiclesEnabled::on_exit(Pelican* p,State<Pelican>* new_state){
    if (new_state && new_state->derives_from(this)) {
        printf("-stay in vehiclesEnabled");
        return;
    }

    printf("->vehiclesEnabled::on_exit");

    operational::on_exit(p,new_state);
}


void Pelican::pedestriansWalk::on_exit(Pelican* p,State<Pelican>* new_state){
    if (new_state && new_state->derives_from(this)) {
        printf("-stay in pedestriansWalk");
        return;
    }
    printf("->pedestriansWalk::on_exit");

    KillTimer(hWnd_, 1);
    pedestriansEnabled::on_exit(p,new_state);
}

void Pelican::vehiclesYellow::on_exit(Pelican* p,State<Pelican>* new_state){
    if (new_state && new_state->derives_from(this)) {
        printf("-stay in vehiclesYellow");
        return;
    }
    printf("->vehiclesYellow::on_exit");

    KillTimer(hWnd_, 1);
    vehiclesEnabled::on_exit(p,new_state);
}
void Pelican::vehiclesGreen::on_exit(Pelican* p,State<Pelican>* new_state){
    if (new_state && new_state->derives_from(this)) {
        printf("-stay in vehiclesGreen");
        return;
    }

    printf("->vehiclesGreen::on_exit");

    KillTimer(hWnd_, 1);
```

Continued on Next Page

`_declspec(novtable)` is a Microsoft-specific optimization that tells the compiler not to generate a `vtable` for this class because it's supposed to be derived from.

Note how RTTI can be used for tracing in debug mode, even if you have no other uses for it; see (`calc::on_enter()`): `p->dispState(typeid(*this).name())`.

Acknowledgments

Thanks to Miro Samek, whose articles got me interested in the state machine design pattern in the first place.

References

[1] Miro Samek's *CUJ* articles on state machine design: "Who Moved My State?" April 2003 and "Déjà Vu" June 2003.
[2] Samek, Miro. *Practical Statecharts in C/C++*, CMP Books, 2002.
[3] Gamma, Erich, et. al. *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995.
[4] Miro Samek's web site: designing a Calculator HSM (http://www.quantum-leaps.com/cookbook/recipes.htm).
[5] Miro Samek's *CUJ* June 2003-design of a Pelican crossing (http://www.quantum-leaps.com/writings.cuj/samek0306.pdf). ❑

Listing 4 ***continued***

```
    vehiclesEnabled::on_exit(p,new_state);
}

void Pelican::pedestriansFlash::on_exit(Pelican* p,State<Pelican>* new_state){
    if (new_state && new_state->derives_from(this)) {
        printf("-stay in pedestriansFlash");
        return;
    }

    printf("->pedestriansFlash::on_exit");

    KillTimer(hWnd_, 1);

    pedestriansEnabled::on_exit(p,new_state);
}

State<Pelican>* Pelican::operational::off(Pelican* p,const boost::any&){
    return &p->m_Off;    //can simply return 0;
}

State<Pelican>* Pelican::vehiclesGreen::pedestrianWaiting(Pelican*,const
  boost::any&){
    m_isPedestrianWaiting = TRUE;
    return this;
}
State<Pelican>* Pelican::vehiclesGreen::timeout(Pelican* p,const boost::any&){
    if (m_isPedestrianWaiting)return &p->m_vehiclesYellow;
    else                      return &p->m_vehiclesGreenInt;
}

State<Pelican>* Pelican::vehiclesGreenInt::pedestrianWaiting(Pelican* p,const
  boost::any&){
    return &p->m_vehiclesYellow;
}

State<Pelican>* Pelican::vehiclesYellow::timeout(Pelican* p,const boost::any&){
    return  &p->m_pedestriansWalk;
}

State<Pelican>* Pelican::pedestriansWalk::timeout(Pelican* p,const
  boost::any&){
    return &p->m_pedestriansFlash;
}

State<Pelican>* Pelican::pedestriansFlash::timeout(Pelican* p,const
boost::any&){
    if (--pedestrianFlashCtr_ == 0) //done flashing?
        return &p->m_vehiclesGreen; //note the change from Miro Samek's--
                                    //we enter derived state directly

    //even counter?DONT_WALK:BLANK
    signalPedestrians((pedestrianFlashCtr_ & 1) == 0? DONT_WALK : BLANK);

    return this;
}

Pelican::Pelican(){}

void Pelican::init(){
    operational::signalVehicles(operational::RED);
    operational::signalPedestrians(operational::DONT_WALK);

    //make the initial transition. note the change from Miro Samek's--
    //we enter derived state directly
    transition(&m_vehiclesGreen);
}

void Pelican::operational::signalVehicles(enum VehiclesSignal sig) {
   ShowWindow(GetDlgItem(hWnd_, IDC_RED), sig == RED);
   ShowWindow(GetDlgItem(hWnd_, IDC_YELLOW), sig == YELLOW);
   ShowWindow(GetDlgItem(hWnd_, IDC_GREEN), sig == GREEN);
}

void Pelican::operational::signalPedestrians(enum PedestriansSignal sig) {
   ShowWindow(GetDlgItem(hWnd_, IDC_DONT_WALK), sig == DONT_WALK);
   ShowWindow(GetDlgItem(hWnd_, IDC_BLANK), sig == BLANK);
   ShowWindow(GetDlgItem(hWnd_, IDC_WALK), sig == WALK);
}

BOOL CALLBACK Pelican::dlgproc(HWND hwnd,UINT iMsg,WPARAM wParam, LPARAM lParam){
   static Pelican app;
   Pelican::Event e(0);

   switch (iMsg) {
    case WM_INITDIALOG:
        hWnd_=hwnd;
        app.init();
        SendMessage(hwnd, WM_SETICON, (WPARAM)TRUE,
(LPARAM)LoadIcon(GetModuleHandle(0), MAKEINTRESOURCE(IDI_ICON1)));
        break;
    case WM_TIMER:
        e =State<Pelican>::timeout;
        break;
    case WM_COMMAND:
        switch (LOWORD(wParam)) {
        case IDCANCEL:
            e =State<Pelican>::off;
            break;
        case IDC_PEDESTRIAN_WAITING:
            e = State<Pelican>::pedestrianWaiting;
            break;
        }
   }

   if(e){
    boost::any arg;
    if(0==app.dispatch(e,arg))
        EndDialog(hWnd_, 0);
   }
   return FALSE;
}

int main(int argc, char* argv[]){
    return DialogBox(GetModuleHandle(0), MAKEINTRESOURCE(IDD_DIALOG), 0,
      Pelican::dlgproc);
}
```