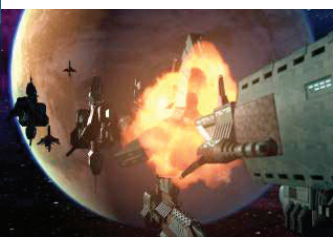
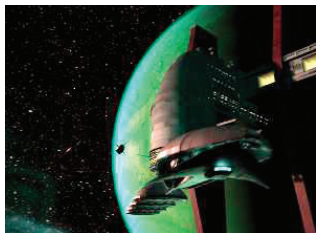
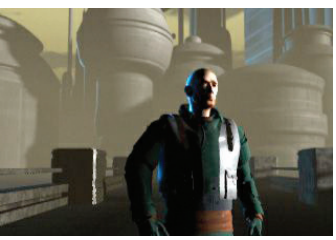


Object-oriented GAME Development




Julian Gold

Object-oriented Game Development

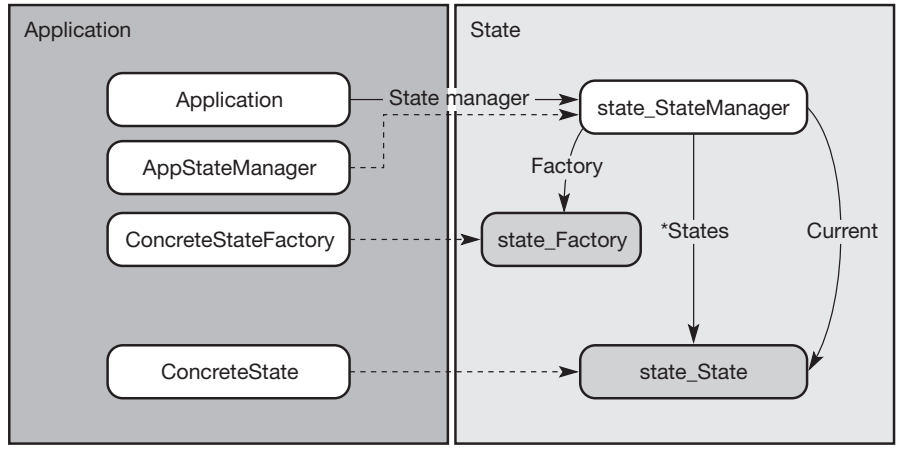


Figure 4.12 Custom state management.

```
class state_StateManager
{
public:
    state_StateManager();
    virtual ~state_StateManager();

    virtual void SwitchToState( state_State * pState )
    {
        /* As before */
    }

private:
    state_State * m_pCurrentState;
};
```

Another inspection of our class diagram might suggest that the state manager class might be a candidate for a singleton. Quite so! There is logically only one state manager – why would we need more? – but since it is hard, nay impossible, to inherit from a singleton (what with those private constructors and destructors), it is important that it is the subclass, not the base class, of `state_StateManager` that is a singleton.

In fact, the manager is not the only class in the system that is a natural singleton. The state subclasses themselves are unique, so the creation functions passed to the concrete state factory can return the singleton instance rather than `new`'ing a subclass. This hugely reduces the likelihood of memory leaking, being multiply deleted, or other related horrors.

```
// ConcreteState.hpp
#include "state_State.hpp"
```

```
class ConcreteState : public state_State
{
public:
    static ConcreteState & Instance();

    void Update( Time dt );
    void Draw( Renderer * pRenderer ) const;

private:
    // …
};

// ConcreteState.cpp
#include "ConcreteState.hpp"
#include "ConcreteStateManager.hpp"
#include "ConcreteStateFactory.hpp"

namespace
{

state_State * createState()
{
    return( &ConcreteState::Instance() );
}

bool registerState()
{
     ConcreteStateManager & aSM =
          ConcreteStateManager::Instance();
    state_Factory * pSF = aSM.GetFactory();

    return( pSF->Register( "Concrete", createState ) );
}

}

/*static*/
ConcreteState & ConcreteState::Instance()
{
    static ConcreteState anInstance;

    return( anInstance );
}
```

In engineering the state package in this fashion, we have established a paradigm: the way a system works. Take a look at how the application's main loop might be written:

```
Time t1 = GetTime();
Time dt = 0;
for(;;)
{
    ConcreteStateManager & aSM =
        ConcreteStateManager::Instance();

    state_State * pState = aSM.GetCurrentState();
    if ( pState != 0 )
    {
        pState->Update( dt );
        pState->Draw( pRenderer );
    }
    Time t2 = GetTime();
    dt = t2 - t1;
    t1 = t2;
}
```

We have a simple main loop, and a set of decoupled, largely independent states that can benefit further from inheritance to propagate common behaviours to state subclasses. This illustrates the benefits of moving to a pattern-driven, object-oriented programming methodology. Although a similar state system could, in principle, be written in C, the semantics of C++ make the code clear to understand, extend and maintain.

We'll return to the state management system later when we discuss adding actual data and graphics to the model.

Case study 4.4: graphical user interface

There isn't a game on the planet that doesn't have a user interface (UI), a method for taking changes in a controller and translating that into actions and responses in the game. This suggests that UIs are eminently abstractable, but we need to be a bit careful here because modern UIs usually come with complex graphical representations and we should be watchful to abstract only the generic behaviour common to all interface systems, lest we end up with specific behaviour at too low a level for reusability.

The graphical bit

For historical and habitual reasons, we'll assume that the basic package name for the GUI graphical bit is called 'view'. In order to keep the view package clean and simple, we need to avoid references to particular platforms and particular ways of

drawing things. We'll make an assumption first: all graphical elements are represented by 2D rectangular areas. This isn't necessarily the most general way of storing region data, and it is possible to use non-rectangular regions for a truly universal representation. Nevertheless, whatever shape we choose, it will be bounded by a rectangle. For a few shapes, the rectangle will be a less efficient bound, but for the current purpose rectangles will do fine.

We'll look at the representation of the rectangle soon. But let's define the class that drives the whole GUI system – the View. A View is a rectangular region of screen that knows how to draw itself. It also supports the notion of *hierarchy*. A View can have child Views: when you perform an operation on a View, you implicitly affect all of its children as well (Figure 4.13).

This translates naturally to code that might look something like this:

```
// File: view_View.hpp
#include <list>
#include "view_Rectangle.hpp"

class view_View;
typedef std::list<view_View *> tViewList;

class view_View : public view_Rectangle
{
public:
    view_View();
    virtual ~view_View();

    virtual void Render()const = 0;

    void AddChild( view_View * const pChild );
    void RemoveChild( view_View * const pChild );
    void RemoveAllChildren();
```

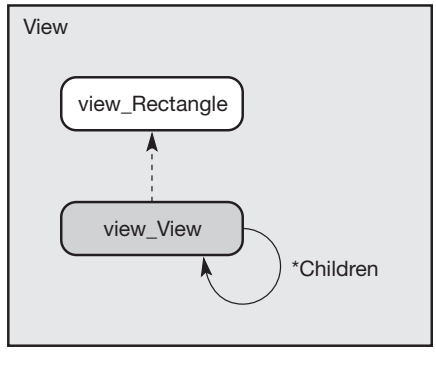


Figure 4.13
Basic view classes.

```
private:
    std::list<view_View *> m_Children;
};

// File: view_View.cpp
#include "view_View.hpp"
#include <algorithm>

// Function objects that perform per-view tasks.
struct ViewDeleter
{
    inline void operator()( view_View * pView )
    {
        delete pView;
    }
};

struct ViewRenderer
{
    inline
    void operator()( view_View const * pView ) const
    {
        pView->Render();
    }
};

view_View::view_View()
: view_Rectangle()
{
}

view_View::~view_View()
{
    // Delete all child views.
std::for_each( m_Children.begin(),
               m_Children.end(),
               ViewDeleter() );
}

/*virtual*/ void view_View::Render() const
{
    // Base class must render all child views.
    std::for_each( m_Children.begin(),
                   m_Children.end(),
                   ViewRenderer() );
}
```

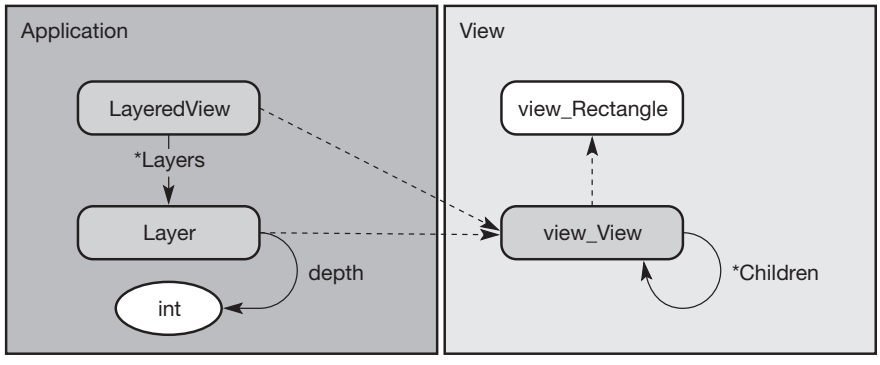


Figure 4.14
Using the `view_View` class.

Notice that the `Render()` method is pure but implemented. This keeps the `view_View` class abstract, as at this level we don't define so much what we do as how we do it. This has a subtle repercussion on how we define the base rectangle. The key question is 'How do we represent a 2D position on the view?' Consider a game that uses a type of view called a *layered view*. This defines a precise order of drawing for a series of controlled views called *layers*. Each of the layers has an associated integer depth and the layers are rendered in order lowest to highest (Figure 4.14).

(You are permitted to either shudder or marvel at the recursivity here: a layered view is a view that contains layers that are also views, ergo they can be layered views.)

Now, consider how we might define the concept of position for a View. In the base system, an *x* and a *y* ordinate suffice. However, in the layered system, the depth is required to define a position uniquely. This suggests that making the concept of location abstract is essential to a generic representation (Figure 4.15).

Now it becomes the job of the view subclasses to redefine the concept of position should it be required. Each view supports a factory method, a virtual function that manufactures the correct type of positional descriptor.

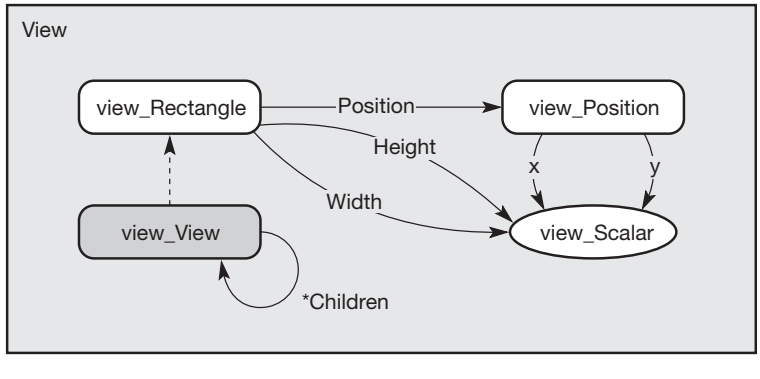


Figure 4.15
Adding position to the rectangle.