Xcpp Mixins

David Barrett-Lennard
Cedanet Pty Ltd
Perth, Western Australia
david.barrettlennard@cedanet.com.au

March 24, 2010

Abstract

A mixin is a fragment of a class that is intended to be composed with other classes or mixins. Xcpp (a front end providing extensions to C++) supports a specialised syntax for template mixins. This paper presents some motivating examples and describes the syntax.

1 Overview

Mixins have been described in the literature as a remarkable means to achieve code reuse (See for example [1] and [2]). A mixin is a capability that can be easily added to one or more classes. A mixin itself is never intended to be stand alone - i.e. to be instantiated in isolation. Rather it is only an adornment to be applied to some existing class.

Typically a mixin is highly reusable, and may be mixed into many different concrete classes. Therefore it can be regarded as a powerful means of achieving code reuse. There are different ways that the mixin concept has been implemented in C++. One way is to use multiple inheritance. However, there are many advantages to using single inheritance chains of mixins involving template classes that are parameterised on the base class.

Java and C# don't support a general enough form of genericity to make the technique possible in those languages [3].

The Xcpp front end provides specialised support for mixins, and involves the `$mixin` keyword. A detailed description of this feature is the main focus of this article.

2 C++ parameterised template mixins

In this section the technique of using mixins is illustrated in standard C++ in the GUI controls application domain. For simplicity it is assumed the GUI elements appear in a rectangular region and the examples only show how mixins can be used to help write `GetWidth()` and `GetHeight()` methods. In a more complete example, methods to perform drawing (e.g. by issuing OpenGL commands), hit testing and to process mouse events would be defined as well.

2.1 Some example mixins

The mixin `Rotate90` applies a rotation of 90 degrees to its (unspecified) base class. Note that when a rectangle is rotated by 90 degrees its width becomes its height and vice versa.

```
template <class Base>
struct Rotate90 : public Base
{
  int GetWidth() const
  {
    return Base::GetHeight();
  }
  int GetHeight() const
  {
    return Base::GetWidth();
  }
};
```

In a more complete example, methods would also be implemented to apply a rotation transformation before calling the base class draw method (this for example would simply involve a call to `glRotate` in OpenGL), and to rotate (x,y) positions passed into hit testing or mouse event methods. The upshot is that one can rotate *any* GUI control by 90 degrees, and it works exactly as expected. For example a horizontal slider control becomes a vertical slider, and even allows the mouse to be used to drag the slider thumb in a vertical direction.

The following mixin scales the x coordinate by `scalex`, and the y coordinate by `scaley`.

```
template <class Base, int scalex, int scaley>
struct Scale : public Base
{
  int GetWidth() const
  {
    return scalex * Base::GetWidth();
  }
  int GetHeight() const
  {
    return scaley * Base::GetHeight();
  }
};
```

The following mixin applies a border (i.e. left, right, top and bottom margins) around its base class.

```
template <class Base, int border>
struct Border : public Base
{
  int GetWidth() const
  {
    return 2*border + Base::GetWidth();
  }
  int GetHeight() const
  {
    return 2*border + Base::GetHeight();
  }
};
```

2.2 Using the mixins

In order to use the mixins, a concrete class is needed that can be fed into the base of the *mixin chain*. For this purpose, consider a rather conventional C++ class named `UnitSquare`.

```
struct UnitSquare
{
  int GetWidth() const { return 1; }
  int GetHeight() const { return 1; }
  };
```

Mixins are applied in a linear chain involving single inheritance. Interestingly the same mixin can usefully appear more than once in the chain. Most generally the order in which the mixins are applied is significant. For example applying a border before it is scaled, means that the margins are scaled as well.

```
  struct X : Scale< Border< Rotate90<
      Scale<UnitSquare,2,1> >,1 >,10,3 >
  {
  };
```

It turns out that `X::GetWidth()` returns `30`, and `X::GetHeight()` returns `12`. Modern C++ compilers are rather good at inlining, so typically the release build is *exactly the same* as if the following had been entered by the programmer:

```
  struct X
  {
    int GetWidth() const { return 30; }
    int GetHeight() const { return 12; }
  };
```

2.3 Advantages

This example only shows the tip of the iceberg. A more complete example would provide a few dozen mixins, and support drawing, mouse events etc. Mixin classes remain relatively simple because they represent simple orthogonal concepts. In combination they make the compiler generate fast, efficient, non trivial code - the kind of code done manually by the human with conventional GUI programming.

OO programs often exploit dynamic polymorphism to achieve code reuse. For example the decorator design pattern ([4]) is often used. A decorator object could apply a rotation of 90 degrees to the GUI control that it decorates. This approach is different to mixins in the following respects:

- A run time decorator class requires a member variable which is a pointer to the object being decorated. The programmer must write the code to initialise this member. E.g. it could be provided in a constructor or an initialisation method;
- Dynamic polymorphism is required so the same decorator can be applied to many different kinds of objects at run time;
- Compile time assembly instead becomes run time assembly. This can be inconvenient to express in procedural code. Code that creates and wires up objects at run time is typically more verbose than using mixin chains, where binding of method calls through the chain is implicit;
- Very fine grained run time assembly can lead to objects that seem mysterious - such as an object that rotates another object;
- Run time assembly can exacerbate the problem of supporting persistence (i.e. assuming the assemblies themselves are to be made persistent). There are great performance overheads for persisting fine grained graphs or trees of objects, and perhaps even more significantly it greatly complicates the problem of schema evolution;
- There is inevitably the overhead of indexing into a vtable for each method call, and more significantly it defeats the inlining capabilities of the compiler; and
- There are more objects to be heap allocated.

Run time assembly is very powerful for allowing end users to compose complex systems from simple parts. Therefore both approaches can be important. An effective strategy, that gives the best of both worlds is to use mixins as a basis for writing components that support run time assembly. More specifically, a valuable technique is to write a delegator base class associated with the abstract base class. That way all the compile time mixins can be easily converted into run time decorators. For example:

```
  // Abstract Base Class uses virtual methods
  // to allow for dynamic polymorphism.
  struct GuiControl
  {
    virtual int GetWidth() const = 0;
    virtual int GetHeight() const = 0;
  };

  // General purpose delegator forwards on
  // method calls to its delegate
  struct GuiControlDelegator :
    public GuiControl
  {
    int GetWidth() const
    {
      return delegate->GetWidth();
    }
    int GetHeight() const
    {
      return delegate->GetHeight();
    }
    GuiControl* delegate;
  };

  // Applying the mixin to the delegator
  // gives a run time decorator
  struct Rotate90Decorator :
    Rotate90<GuiControlDecorator>
  {
  };
```

Note that as a result of inlining the run time decorator is just as efficient as a version coded directly without mixins.

3 Using Xcpp for mixins

The above example shows that in straight C++ long mixin chains are rather awkward syntactically. The Xcpp front end provides a much more convenient syntax. The above example can be encoded as follows

```
  $mixin Rotate90
  {
    int GetWidth() const
    {
      return $base::GetHeight();
    }
    int GetHeight() const
    {
      return $base::GetWidth();
    }
  };
  $mixin Scale<int scalex, int scaley>
  {
    int GetWidth() const
    {
      return scalex * $base::GetWidth();
    }
    int GetHeight() const
    {
      return scaley * $base::GetHeight();
    }
  };
  $mixin Border<int border>
  {
    int GetWidth() const
    {
      return 2*border + $base::GetWidth();
    }
    int GetHeight() const
    {
      return 2*border + $base::GetHeight();
    }
  };
  $mixin UnitSquare
  {
    int GetWidth() const { return 1; }
    int GetHeight() const { return 1; }
  };
  $struct X
    mixin
    [
      UnitSquare
      Scale<2,1>
      Rotate90
      Border<1>
      Scale<10,3>
    ]
  {
  };
```

The mixin chain is enclosed in square brackets. It is interpreted as follows: we start with a unit square then apply `Scale<2,1>` to scale it horizontally. Next we rotate by 90 degrees, apply a border then finally scale it vertically and horizontally.

4 Parameterised mixins and model-view-controller

Ceda makes heavy use of the model-view-controller pattern, but in a very unconventional manner. The idea is to use parameterised mixins to write a single class that internally combines the model, view and controller into a *single object*.

The basic pattern is illustrated with the following code

```
  $class MyMVC isa ceda::IView
    model
    {
      // model variables go here
  }
    mixin
    [
      MyViewMixin
      MyControllerMixin
    ]
  {
  };
```

`MyMVC` is a class that supports persistence. It has a single model which supports schema evolution and this feeds into the start of the mixin chain. As a result all mixins have read and write access to the model variables. Note that this access is through the appropriate read and write barriers. Therefore when the view mixin(s) read the model they automatically establish dependencies for the Dependency Graph System. Also when the controller manipulates the model variables, operations are automatically generated against the model. These operations allow for interactive and non-interactive collaboration amongst multiple users, configuration management etc. Note as well then whenever the model variables are changed (either locally through the controller, or remotely due to the execution of operations received from other computers), the dependent caches will automatically be marked as dirty. Therefore there is no need for the programmer to be concerned with the observer pattern between model and view.

5 Anonymous mixins

When writing a mixin the objective is generally to make it maximally reusable. Often the best way to achieve that aim is to avoid any state (i.e. member variables)! Consider a mixin that is responsible for drawing a slider

```
  $mixin DrawSliderMixin
  {
    void Draw() const
    {
      float pos = GetThumbPos();
      int height = GetHeight();
      int width = GetWidth();

      // Draw the slider in a rectangle of
      // dimensions 'width', 'height'
      // with the thumb at position 'pos'
      ...
    }
  };
```

Notice that this mixin assumes that the base class has implemented the following functions

```
  int GetWidth() const;
  int GetHeight() const;
  float GetThumbPos() const;
```

We have already seen how mixins can help implement `GetWidth()` and `GetHeight()`. For example

```
  $mixin DefaultSliderDimensionsMixin
  {
    int GetWidth() const { return 128; }
    int GetHeight() const { return 32; }
  };
```

Now consider the following class

```
  $class X isa ceda::IView
    model
    {
        int m_x;
    }
    mixin
    [
      DefaultSliderDimensionsMixin

      // Anonymous mixin
      {
        float GetThumbPos() const
        {
          return (float) m_x/100;
        }
      }

      DrawSliderMixin
    ]
  {
  };
```

This makes use of an *anonymous* mixin within the mixin chain. The anonymous mixin implements the method `GetThumbPos()` that is expected by `DrawSliderMixin`. The result is that the slider position is determined by the data model variable `m_x`.

References

[1] G. Bracha and W. Cook, *Mixin-Based Inheritance*, Proceedings of the 8th Conference on Object-Oriented Programming, Systems, Languages, and Applications, European Conference on Object-Oriented Programming, 1990.

[2] Ulrich W. Eisenecker, *Mixin-Based Programming in C++*, Dr. Dobb's Journal, January 2001, `http://www.ddj.com/cpp/184404445`.

[3] Bruce Eckel, *Mixins: Something Else You Can't Do With Java Generics?*, October 19, 2005, `http://www.artima.com/weblogs/viewpost.jsp?thread=132988`.

[4] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995, ISBN-13: 9780201633610.

[5] ISO/IEC 14977:1996(E). *Information technology - Syntactic metalanguage - Extended BNF*, First edition 1996-12-15. `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`.