An Introduction to M Classes

This article provides a comprehensive introduction to Symbian M (mixin) classes

Introduction

'I want to have this sort of thing so that I'll be able to tell that this event had taken place in my object. I'm not sure how the rest of that thing will look and what it will do with that information, but I need to inform it by calling `MyThingJustHappened()`. I need to enforce the syntax of such a call but I don't really care about the rest.'

How many times have you had this problem?

In such a situation, all you need to do is define an M class!

Symbian Platform standards define certain class types in order to group classes with similar properties and behavior; these are C classes, R classes, T classes and M classes1. M classes have their prefix originating from 'mixin,' which is an early object-oriented term that defines interface classes. (Actually, legend has it that the term 'mixin' was borrowed from an ice cream parlor near the Massachusetts Institute of Technology, where your ice cream came with a base 'flavor' and you could then 'mix in' other flavors.)

M classes define abstract protocols or interfaces which declare pure virtual functions. Actual implementations of those interfaces are provided by concrete classes deriving from M classes. When referring to multiple inheritance, it implies inheriting from a main flavor base class with a combination of additional mixin classes which extend its functionality. This is the only form of multiple inheritance which should be used in Symbian Platform development; inheriting from more than one C class will inevitably introduce unmanageable complexity and ambiguity.

All the Symbian Platform interface classes have their name preceded by an M so that they are easily recognized.

An M class has similar characteristics to a Java interface. However, since it's C++, no restrictions can be enforced and it is possible to break all the rules; for example, you can provide full method implementations in your M class, derive from a C class, etc.

Anatomy of an M Class

Although opinions differ slightly as to what an M class should be, the common rules are as follows:

- M classes should have the 'M' prefix.
- M classes should have no member data.
- M classes should not have a constructor or a destructor: no member data means that there is nothing to construct/delete (although there are some exceptions that we'll discuss later).
- M classes should usually define a pure abstract interface. However, there might be cases where it may be appropriate to implement some virtual functions if some behavior is shared.
- M classes should not contain overloaded operators such as =

Some Warnings

Use with C classes

It is good practice to use the correct order when creating a C class that also derives from an M class. Try to ensure that `CBase` or a `CBase`-derived class is the first base class. This will emphasize the primary inheritance tree. If an M class is the first base class, you may have problems when using the cleanup stack; if you use one of the `Pop()` methods which takes a parameter to pop your object, the expected object will be different to that found at the top of the stack (see the accompanying C class article for full details). Although the C++ standards do not make it compulsory that object layout follows the order in which base classes are specified, in practice this is the case for most current compilers, including those used for Symbian Platform.

Don't mix in an already-mixed in mixin!

No M class needs to be mixed in more than once in any class, either as a direct base or as a base of any of its primary base classes. This reflects the fact that if the base class already supports the protocol defined by the M class, there is nothing to gain from mixing in the M class again. In fact, due to issues with ambiguity, your compiler will complain if you attempt to use a mixin twice.

Name clash

It is easy to imagine the problem of a name clash occurring when using M classes; for example, you may wish to implement two separate interfaces which happen to contain methods with the same name, such as `Count()`:

```
Class CSchool : public CBase, public MSchoolPupils, public MSchoolEmployees
     {
public:
     // from MSchoolPupils
     TInt Count();
     // from MSchoolEmployees
     TInt Count();
     };
```

A simple naming convention avoids this problem: just add the name of the interface (or part of it) to the beginning of its methods:

```
Class CSchool : public CBase, public MSchoolPupils, public MSchoolEmployees
     {
public:
      // from MSchoolPupils
     TInt MPupilsCount();
     // from MSchoolEmployees
     TInt MEmployeesCount();
     };
```

A real world example of this technique can be found in `MDesC16Array`, which declares the methods `MdcaCount()` and `MdcaPoint()`.

Cleanup

Although in most cases the M class interface is passed through references, there are some cases that require a pointer to be passed. Remembering that having a reference to an object usually means it is owned elsewhere, when you have a pointer to an object there may be a need for a cleanup mechanism. A destructor places a constraint on how the M class is mixed in, forcing it to be implemented only by a `CBase`-derived class. This is because a destructor means that `delete` will be called, which in turn demands that the object cannot reside on the stack, and must always be heap-based. Therefore, an implementing class generally only derives from `CBase`, since T classes and R classes rarely possess destructors.

If all implementers of your interface class are guaranteed to be C classes (i.e., they all derive from `CBase`), then you can provide a virtual destructor in the M class. This allows the owner of the M class pointer to call delete on it. However, cleanup need not be limited to using a destructor. You may instead create a pure virtual `Release()` method.2 For a C class, the method can simply call delete. This is a more flexible interface: we are no longer restricting our implementers to be `CBase`-derived. Also, it enables the client to make use of the `CleanupReleasePushL()` function. `CleanupReleasePushL()` is a cleanup method that calls `Release()` on the object in question when the cleanup stack tries to delete/destroy the item.

Note: `Release()` is a useful method to implement when an object is shared and reference counting is taking place. In this context it causes the number of references to the object to be decremented.

Observers: a Basic Usage Pattern

Overview

The observer is a pattern used to support a one-to-many relationship among collaborating objects. An object known as the ‘subject’ notifies all registered observers when some event has taken place.

For example, let’s declare an observer class:

```
class MMyObserver
     {
```

```
public:
     virtual void SomethingWentWrong(TInt aErrorCode) = 0;
     };
```

Next, we can declare our subject, which will notify the observer if its calculations go wrong. In order to allow the subject to notify the observer, we'll pass the observer as a reference:

```
class CMySubject : public CBase
     {
public:
     ~CMySubject();
     static CMySubject* NewL(MMyObserver& aObserver);
     void DoCalculation();

private:
     CMySubject(MMyObserver& aObserver);

     void ConstructL();
private:
     MMyObserver& iObserver;
     };
```

Our subject is supposed to notify the observer if the result of DoCalculation() meets certain requirements:

```
void CMySubject::DoCalculation()
     {
     // make some calculations and store outcome in result
     if(result < 0)
     {
     iObserver.SomethingWentWrong(errorCode);
     }
}
```

The beauty of it all is that our observer can be, for example, a UI:

```
class CMyDialogObserver : public CAknDialog, public MMyObserver
     {
public:
     static void ShowL();
     // ...
private:
        void PreLayoutDynInitL();
     // from MMyObserver:
        void SomethingWentWrong(TInt aErrorCode);
private:
     CMySubject * iMySubject;
     };
```

During the dialog's construction, we create our subject and tell it that our dialog is the observer:

```
void CMyDialogObserver::PreLayoutDynInitL()
     {
     iMySubject = CMySubject::NewL(*this);
     // ...
```

```
    }
```

Now we need to implement the observer's method:

```
void CMyDialogObserver::SomethingWentWrong(TInt aErrorCode)
    {
    // for example, modify the display to indicate the change
    }
```

In Figure 1, the concrete observer implements the M class interface. The observer uses the`CMySubject` constructor to register itself with the subject. `CMySubject` calls `iObserver.SomeThingWentWrong()` to notify the observer of a change.

Unit testing

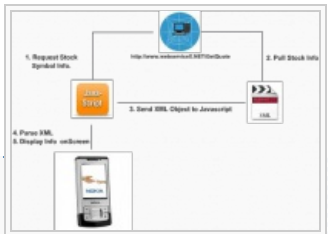
Figure 1

But it's not over. We can write unit tests (using SymbianOSUnit, for example) and fake the UI. First the fake object:

```
class TFakeUI : public MMyObserver
    {
public:
    inline TFakeUI() {iErrorCode = 0;}
private:
    // from MMyObserver:
    inline void SomethingWentWrong(TInt aErrorCode) {iErrorCode = aErrorCode;}
public:
    TInt iErrorCode;
}
```

Now an actual test:

```
void CMyTests::testMySubjectL()
    {
    TFakeUI fakeUI;
    CMySubject* mySubject = CMySubject::NewL(fakeUI);
    CleanupStack::PushL(mySubject);
    TS_ASSERT_EQUALS(0, fakeUI.iErrorCode);
    mySubject.DoCalculation();
    TS_ASSERT_EQUALS(-50, fakeUI. iErrorCode);
    CleanupStack::PopAndDestroy(mySubject);
    }
```

The test asserts that the fake UI should get called and that the value passed to the observer equals -50.

Alternative to observers

There is an alternative to the observer pattern in Symbian C++, `TCallBack`, which encapsulates a general callback function. Basically, one class requests another class to perform a task. When the class has finished the task, it calls back the requesting class.

Exceptions to the Common Usage Pattern

Even though the rules about how M classes are defined and used apply in general, there are sometimes exceptions.

Functions

We mentioned at the start of this paper that M classes should usually define a pure abstract interface. However, there are

occasions when providing some implementation is desirable. For example, if you know that all the implementers of the interface will share some common behavior, you can implement this in the interface:

```
class MRectShape
     {
public:
     // area functionality will be the same for all implementors
     TReal AreaOfRect(TReal aWidth, TReal aHeight) {return aWidth * aHeight;}
     }
```

Bear in mind that the implementation is limited in what it can actually do, as the M class can have no member data.

Destructors

As mentioned in the cleanup section (Section 3.4), it may be necessary to provide a virtual destructor. Both `MDesCArray` and `MListBoxModel` are examples of M classes with virtual destructors and are discussed further in Section 6.

Some Common M Classes

MDesCArray

`MDesCArray` is an interface class for descriptor arrays. It should be inherited by classes which implement this protocol, such as `CDesCArray` and the concrete class `CPtrCArray`.

The interface defines three things:

- a virtual destructor so that the array can be deleted through the interface pointer
- a function to get the item count:

```
virtual TInt MdcaCount() const = 0;C/code>
*a function to get a particular item in the array:
<code>virtual TPtrC MdcaPoint( TInt aIndex ) const = 0;
```

What makes this class so powerful is that it can be used to make a data source out of any class. The base class for all Symbian list boxes,`CEikListBox` (and more specifically its model), uses `MDesCArray` to access the list box lines. This means that any class that implements the interface can be given to any list box as a data source as long as the descriptors it returns conform to the format of the list box.

MListBoxModel

This mixin protocol is implemented by all list box models. List box models provide information needed to display the required data in a list box. The `MListBoxModel` provides the bare bones of what a list box requires. Apart from a virtual destructor, it contains only two methods:

- `NumberOfItems()`, which returns the number of items in the model
- `MatchableTextArray()`, which returns an array of strings used by the list box for matching user key presses incrementally.

The class is defined as:

```
class MListBoxModel
     {
public:
     IMPORT_C virtual ~MListBoxModel();
     virtual TInt NumberOfItems() const = 0;
     virtual const MDesCArray* MatchableTextArray() const = 0;
     };
```

`MListBoxModel` is a completely generic class. A more specialized class is `MTextListBoxModel`. This interface, which inherits from `MListBoxModel`, represents a text-based list model. It adds an abstract method, `ItemText()`, which returns the item content as a

string:

```
class MTextListBoxModel : public MListBoxModel
     {
public:
     IMPORT_C ~MTextListBoxModel();
     virtual TPtrC ItemText(TInt aItemIndex) const = 0;
     };
```

Symbian provides a default `MTextListBoxModel` implementation: `CTextListBoxModel`.

MHTTPDataSupplier

`MHTTPDataSupplier` is used to deliver the HTTP response data to the client and is also used by the client to supply request body data to HTTP in POST transactions. Data is supplied in a number of parts and when a part becomes available it can be retrieved with `GetNextDataPart()`. The returned descriptor will remain valid until `ReleaseData()` is called. Other methods include:

- `OverallDataSize()` – this returns the size of the data being supplied. If the data is in several parts, it will return the sum of the size of the data parts.
- `Reset()` – this resets the data supplier, returning it to the first part of the data.

MGraphicsDeviceMap

`MGraphicsDeviceMap` defines the size-dependent functions in a graphics device. It provides methods to handle twips to pixel conversion, such as:

- ```
  TPoint}} PixelsToTwips(const TPoint& aPixelPoint) const;
  ```
- ```
  TPoint TwipsToPixels(const TPoint& aTwipPoint) const;
  ```

and will locate the device font closest to the device independent specification:

- ```
  virtual TInt GetNearestFontInTwips(CFont*& aFont, const TFontSpec& aFontSpec) = 0;
  ```
- ```
  virtual void ReleaseFont(CFont* aFont) = 0;
  ```

It is used as the base class for TZoomFactor which implements the twips/pixel conversion and facilitates zooming.

MHTTPSessionEventCallback

`MHTTPSessionEventCallback` receives and reports HTTP session events, defined in `THTTPSessionEvent`. These include:

- `EConnectedOK`
- `EDisconnected`
- `ERedirected`
- `EAuthenticationFailure`.

MEikListBoxObserver

`MEikListBoxObservers` are used to receive events from list boxes. Each `MEikListBoxObserver` can observe a number of list boxes. Events are passed to the observer via the `HandleListBoxEventL()` method. The events which can be received include:

- `EEventEnterKeyPressed`
- `EEventItemClicked`
- `EEventItemDoubleClicked`

MCameraObserver2

In order to make use of the camera API, a class must implement the `MCameraObserver2` interface. This class is notified when the camera is ready for use. Its methods include:

- `HandleEvent()`, which receives the event as a `TECAMEvent`.
- `ViewFinderReady()`, which takes a reference to a camera buffer and an error code, `KErrNone` if successful. This method is called periodically in response to `CCamera::StartViewFinder()`.
- `ImageBufferReady()` and `VideoBufferReady()`. These methods notify the client of a new captured image or video respectively. Both take a reference to a camera buffer and an error code, `KErrNone` if successful.

MContactDbObserver

`MContactDbObserver` handles changes to a contact database. Its single method is the pure virtual `HandleDatabaseEventL()`, which takes a `TContactDbObserverEventType`, such as:

- `EContactDbObserverEventContactChanged`
- `EContactDbObserverEventContactDeleted`
- `EContactDbObserverEventContactAdded`.

`HandleDatabaseEventL()` tests for the type of event and either handles it or ignores it.

MMdaAudioOutputStreamCallback

This interface is used to monitor the progress of audio output streaming (similarly, `MMdaAudioInputStreamCallback` monitors audio input streaming). It includes the pure virtual methods:

- `MaoscOpenComplete()`, which indicates that the audio stream is ready for use.
- `MaoscBufferCopied()`, which is called either when the buffer has been copied to the lower layers of the multimedia framework, an error has occurred, or when the client stopped the stream playing before the buffer has been fully copied.
- `MaoscPlayComplete()`, which is called when playback terminates as a result of `CMdaAudioOutputStream::Stop()`. It takes an error code such as `KErrUnderFlow` or `KErrNone` if successful.

For more details on all of the above classes, please see the latest SDK.

Summary

M classes define interfaces which are implemented by derived classes and provide the only approved form of multiple inheritance in Symbian C++. Prefixed with ‘M,’ they are easy to recognize and contain no member data. For more information, see the references in the following section.

References

- Stichbury, (2004) Symbian OS Explained: Effective C++ Programming for Smartphones, pages 9-10. John Wiley & Sons, Ltd.
- Gamma, Helm, Johnson & Vlissides, (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.
- Symbian OS Unit is a port of the popular C++ unit testing framework CxxUnit.
- Issott, (2008) Common Design Patterns for Symbian OS]], pages 93-103. John Wiley & Sons, Ltd.
- Symbian Reference in the Nokia Library, the complete Symbian developer help resource.

Company Profile

Penrillian makes mobile software work. We are experts in developing and porting applications across all the major mobile software platforms, turning great ideas in to real products, quickly and efficiently. Our diverse customer base includes global service providers and software companies such as Handmark, Sybase, T Mobile and Vodafone. They choose us because we are agile enough to respond to changing demands and always deliver, on time and on budget.

Our core competencies include: Symbian OS, S60, UIQ, Java, Windows Mobile, secure applications, connectivity, RFID and user interface design. We maintain close links with the major influencers and knowledge centres in the mobile software industry and we’re pleased to be a Symbian Platinum Partner, UIQ Alliance Partner and a member of Nokia Developer PRO.

To find out more about our consultancy, software porting and application development services go to www.penrillian.com.

Author Biographies

Barry Drinkwater Barry Drinkwater is a software developer at Penrillian who graduated with first class honors in Computing. He started his career in the software industry as a Software Test Engineer before progressing to his current position as Software Developer. He has been involved in numerous Symbian projects, such as a Bluetooth locator application, a live video streaming application and a network monitoring service.

Sebastian Jakubowski Sebastian Jakubowski is Penrillian's Senior Software Developer and specializes in Symbian development. Sebastian graduated as Master of Science in Software Engineering. He has over 5 years commercial experience in software, starting as a warehousing software developer and gaming engine developer, before moving into mobile development.