Abstract Classes and Pure Virtual Functions

In this article we will be discussing abstract classes, and how they are implemented in C++. The first part of the article will define abstract classes and describe how to use them as tools for *Object Oriented Design.* The second half of the article will concern itself with their implementation in C++, and some of the quirks and pitfalls that surround their use.

While it is true that all objects are represented by a class, the converse is not true. All classes do not necessarily represent objects. It is possible, and often desirable, for a class to be insufficient to completely represent an object. Such classes are called abstract classes.

The illustrations used in this article conform to the "Booch Notation" for Object Oriented Design. This is a rich and expressive notation which is very useful for presenting OOD concepts. Where necessary I will digress to explain some of this notation.

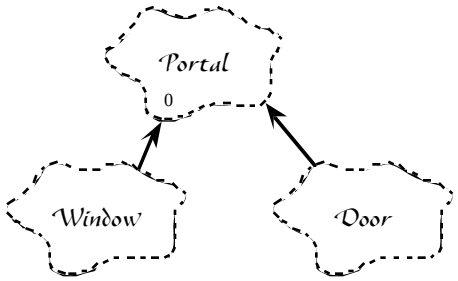
What is an Abstract Class?

Simply stated, an abstract class is a class which does not fully represent an object. Instead, it represents a broad range of different classes of objects. However, this representation extends only to the features that those classes of objects have in common. Thus, an abstract class provides only a partial description of its objects.

Because abstract classes do not fully represent an object, they cannot be instantiated. At first it may seem a little odd that a class is incapable of having any instances. But everyday life is full of such classes. We may describe a particular animal as belonging to the class of all Mammals. However, we will never see an instance of the class Mammal! At least not a pure instance. Every animal belonging to the class Mammal must also belong to a class which is subordinate to Mammal such as Mouse, Dog, Human, or Platypus. This is because the class Mammal does not fully represent any animal. In Object Oriented Design, we will never see an instance of an abstract class, unless it is also an instance of a subordinate class. This is because the abstract class does not fully represent any object. As Booch says: "An abstract class is written with the expectation that its subclasses will add to its structure and behavior..."[1]

The icon above represents a class. Note the '0' in the lower left corner. This represents the *cardinality* of the class, or the number of instances that the class can support. Since this class can support '0' instances, its is abstract.

According to Rumbaugh: "Abstract classes organize features common to several classes."[2] What are the "common features" mentioned by Rumbaugh? They are features of the class's interface. For example, consider the classes *Window* and *Door*. At first glance these classes may not seem to have anything to do with each other. But both share some interesting common features. They are both holes in a wall. They both have particular locations and sizes with respect to the wall. They both may exists in one of three states: {OPEN, CLOSED, LOCKED}. Finally, they can be sent similar messages: {*open, close, lock*}. Thus, we can describe both classes as inheriting from a common abstract base, *Portal*, which contains all the common features of their interface.



STEREOTYPES AND POLYMORPHISM. Notice how this allows *Window* and *Door* objects to be stereotyped. They can both be referred to as *Portal*s. While this is an incomplete description of either a *Door* or a *Window*, it is nonetheless accurate and useful.

This ability to stereotype an object is a powerful design tool. It allows us to bundle all the common aspects of a set of objects together into an abstract class. As Lippman says: "[An abstract class] provides a common public interface for the entire class hierarchy."[3] This common interface allows us to treat all such objects according to the stereotype. While such treatment may not be socially acceptable when dealing with humans, it provides for great efficiencies when dealing with software objects.

For example, consider the *Door* and *Window* classes when they do not inherit from a common base:

```
class Window
{
  public:
    enum WindowState {open, closed, locked};
    void Open();
    void Close();
    void Lock();
    void GetState() const;
    int GetXPos() const;
    int GetYPos() const;
    int GetHeight() const;
```

```
    int GetWidth() const;
  private:
    WindowState itsState;
    int itsXPos, itsYPos, itsWidth, itsHeight;
};

class Door
{
  public:
    enum DoorState {open, closed, locked};
    void Open();
    void Close();
    void Lock();
    void GetState() const;
    int GetXPos() const;
    int GetYPos() const;
    int GetHeight() const;
    int GetWidth() const;
  private:
    DoorState itsState;
    int itsXPos, itsYPos, itsWidth, itsHeight;
};
```

These two classes are horribly redundant. They cry for some form of unification. That unification is supplied by creating the abstract base class.

```
class Portal
{
  public:
    enum PortalState {open, closed, locked};
    virtual void Open() = 0;
    virtual void Close() = 0;
    virtual void Lock() = 0;
    void GetState() const;
    int GetXPos() const;
    int GetYPos() const;
    int GetHeight() const;
    int GetWidth() const;
  private:
    PortalState itsState;
    int itsXPos, itsYPos, itsWidth, itsHeight;
};

class Window : public Portal
{
  public:
    virtual void open();
    virtual void close();
    virtual void lock();
};

class Door : public Portal
{
  public:
    virtual void open();
    virtual void close();
    virtual void lock();
};
```

The efficiencies that we gained are obvious. Several of the `Portal` member functions can be reused. The declarations for *Door* and *Window* are terse and understandable in terms of the functionality of *Portal*. Moreover, both *Window* and *Door* can be stereotyped as a *Portal* when convenient. For example:

```
void TornadoWarning(List<Portal*> thePortals)
{
    ListIterator<Portal*> i(thePortals);
    for (; !i.Done(); i++)
    {
        (*i)->Open();
    }
}
```

Forgive the obvious license with the semantics of the `ListIterator`. Clearly this method for opening all *Doors* and *Windows* is superior to handling each type separately. As the application matures, more types of *Portals* will probably be added. However, since the `TornadoWarning` function deals with *all* species of `Portals`, it will not have to change. Thus, the polymorphic behavior of the abstract class `Portal` provides a more maintainable and robust design.

It should be stressed that forcing *Door* and *Window* to inherit from *Portal* is not necessary to the proper functioning of the application. The application could be designed without the abstract class. However, creating the abstract *Portal* class results in a superior design which promotes polymorphism, and code reuse.

Designing Applications with Abstract Classes

During the first stages of a design, we usually have a good idea of the concrete classes that we need. As the design is refined, we should begin to find common features amongst some of these classes. These common features are not always obvious. They may come from different parts of the design, and may have different names and configurations. It sometimes takes a careful eye to spot them.

For example, Jim, Bill and Bob are working on the design of the software that will control a car crushing machine. Jim is responsible for the control panel, Bill for the hydraulics control and Bob for the servo motors.

Jim has designed an *Indicator* class for the control panel. It looks like this:

```
class Indicator
{
  public:
    enum IndicatorState {on,off};
    Indicator(const IndicatorAddress&)
    void TurnOn();
    void TurnOff();
    IndicatorState GetState() const;
};
```

Bill has designed a *Valve* class for controlling the hydraulics. It looks like this:

```
class Valve
{
  public:
    enum ValveState {open, closed};
    Valve(const ValveAddress&)
    void Open();
    void Close();
    int IsValveOpen() const;
    double GetFlowRate() const;
};
```

Bob has designed a *Motor* class for controlling the servos. It has the following interface:

```
class Motor
{
  public:
    enum MotorState {running, stopped};
    Motor(const MotorAddress&)
    void SetState(const MotorState);
    MotorState GetState() const;
    void SetSpeed(int);
    int GetSpeed() const;
};
```

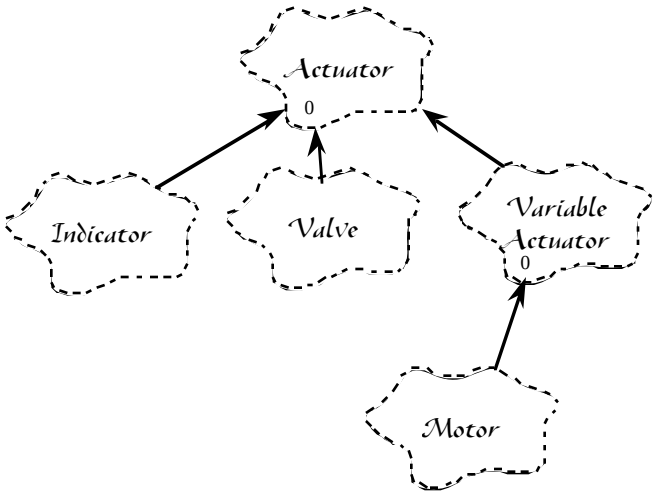
In this simple example, the common features aren't too hard to find. Each of these classes has a binary state, methods to alter that state, and methods to interrogate that state. The forms and names of these methods are dissimilar, but their functions are common. Thus we can create an abstract base class *Actuator* which defines the common points of each:

```
class Actuator
{
  public:
    enum ActuatorState {on, off};
    Actuator();
    virtual void Activate() = 0;
    virtual void Deactivate() = 0;
    ActuatorState GetState() const;
};

class Indicator : public Actuator
{
    Indicator(const IndicatorAddress&);
    virtual void Activate();
    virtual void Deactivate();
};

class Valve : public Actuator
{
    Valve(const ValveAddress&);
    virtual void Activate();
    virtual void Deactivate();
    double GetFlowRate() const;
};

class Motor : public Actuator
{
  public:
    Motor(const MotorAddress&);
    virtual void Activate();
    virtual void Deactivate();
    void SetSpeed(int);
    int GetSpeed() const;
};
```

In a more complex application, commonality can be much harder to detect. The names and forms of the methods can disguise the intrinsic similarities. Thus, care should be exercised in the search for commonality. The effort spent in the search will be paid back with a more maintainable design which supports a higher degree of code reuse.

FACTORING. In the `Actuator` example, we found the common features of the concrete classes, and promoted them to the abstract base. Rebecca Wirfs-Brock, et. al. call this *factoring*[4] They go on to state a profound principles of Object Oriented Design: *"Factor common responsibilities as high as possible."* The higher, in the inheritance hierarchy, common features can be factored, the more chances for re-use and polymorphism are engendered.

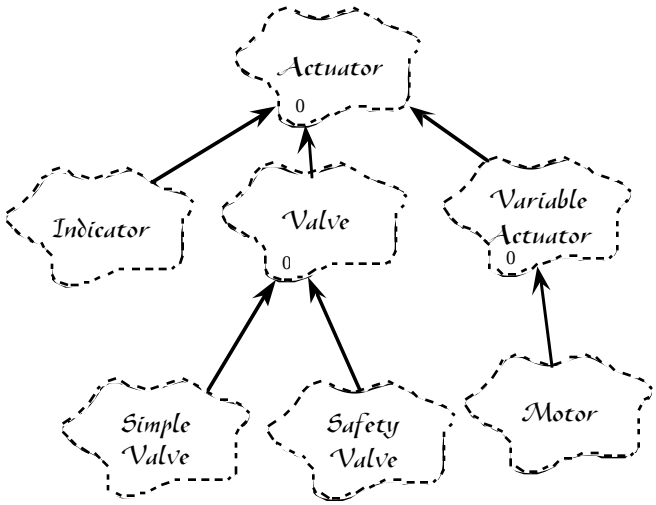
Have we factored the features in the example high enough? Our example didn't have very many features, but there is still room for some more factoring. It seems unlikely that the *Motor* class will be the only class requiring a *speed* setting. So we might want to create an abstract subclass of *Actuator* with methods for handling speed. However, speed is just a variable quantity. We can generalize it by creating the abstract class *VariableActuator*:



This provides us with the opportunity to derive other variable controlled actuators from the common base. For example, we could create a variable brightness lamp, or a variable speed fan.

EVOLUTION OF CONCRETE CLASSES. As the iterative process of design continues to add more and more detail to the application model, classes which had been concrete will tend to become abstract. As we incorporate more of the details of the application into the design, classes which had been specific become the generalities for the new details.

For example, as the car crusher design proceeds, Bill discovers that he needs a safety valve which opens automatically when a pressure limit is exceeded. Bill could simply derive *SafetyValve* from the current *Valve* class. However, this assumes that the the *SafetyValve* is going to share all the features of the simple valve. This may not be the case. Bill might be better off creating an abstract class to represent all valves, and then deriving *SimpleValve* and *SafetyValve* from that common base.



By taking this tack, Bill has created a generalization which fits not only his existing concrete valve classes, but any new valve class that may be needed in the future.

This process of factoring common elements, higher and higher into the inheritance hierarchy as the design progresses, is typical. The more details that are added to the design, the more abstract classes are created to deal with the common generalities.

PURE INTERFACES. Note that abstract classes define features which are common to the interfaces of their derived classes. But what implementation should be provided by the abstract class? For example, what is the implementation for `Actuator::Activate()`. There is no sensible implementation! It is only the derived classes that know how to deal with the `Activate` method. `Indicator::Activate()` turns on a real indicator lamp. `SimpleValve::Activate()` causes a real life valve to open. `Motor:Activate()` turns on a real motor. But `Actuator::Activate()` can't do anything because it doesn't have anything "real" to interface to. Thus, `Actuator::Activate()` is a *pure interface* devoid of any implementation.

It is the *pure interfaces* within an abstract class which define the common features encapsulated within it. They are the basis for the polymorphic behavior of abstract classes. The *impure interfaces*, those interfaces which have implementations, contain the code which is re-used by all the derived classes.

Pure Virtual Functions

In C++, pure interfaces are created by declaring *pure virtual functions*. A pure virtual function is declared as follows:

```
virtual void Activate() = 0;
```

The "`= 0`" on the end of the declaration is supposed to represent the fact that the function has no implementation.

However, as we will see, implementations *are* sometimes provided.

In C++, a class with a pure virtual function is an abstract class. The language provides special semantics for abstract classes. It enforces the constraint that abstract classes cannot have any instances. Thus the compiler will not allow an instance of an abstract class to be created. If you attempt to declare or create one, the compiler will complain:

```
Actuator myActuator; // error
Actuator* myActuator = new Actuator; // error;
```

This constraint does not, however, prevent you from declaring pointers and references to abstract classes. The compiler allows such constructs, and will allow them to refer to instances of concrete classes which have the abstract class somewhere in their inheritance hierarchy.

```
Actuator* myActuator = new Indicator;
Actuator& anActuator = *(new Indicator);
```

Such pointers and references are very useful for taking advantage of the polymorphic attributes of abstract classes. The pure virtual functions of the abstract class are bound to the implementations defined in the instances to which they refer.

```
myActuator->Activate(); // Turn on indicator.
anActuator.Deactivate();// Turn off indicator.
```

Including a pure virtual function in a class is the *only* way to tell the compiler that the class is abstract. This is sometimes considered to be a limitation. It has been suggested that an 'abstract' keyword be added to the language so that classes could explicitly be declared abstract.

```
abstract class Actuator; // suggested syntax.
```

However, in my opinion, this would create a redundancy in the language. A truly abstract class *must* contain a pure interface; otherwise, it would be instantiable.

INHERITING PURE VIRTUAL FUNCTIONS. Pure virtual functions behave differently depending upon the version of the compiler you are using. Cfront 2.0 did not allow pure virtual functions to be inherited by derived classes. If the base class had a pure virtual function, then that function *had to* be declared in the derived classes, either in pure form or in impure form. Thus:

```
class Valve : public Actuator
{
  public:
    virtual void Activate() = 0;
    virtual void Deactivate() = 0;
};
```

This restriction was relaxed in the 2.1 version of the compiler, so pure virtual function *can* be inherited without specifically being declared. If they are not declared in the derived class, they are inherited in *pure* form, making the derived class abstract.

```
class Valve : public Actuator
{
    // This class is abstract.
    // It inherits the pure virtual functions
    // Activate() and Deactivate()
};
```

INSTANCES OF ABSTRACT CLASSES. Although it is impossible to explicitly create an instance of an abstract class, such instances can temporarily exist during construction or destruction. Instances under construction are only as complete as the currently executing constructor has made them. Instances under destruction are only as complete as the executing destructor has left them. Because these instances are incomplete, the language prevents the virtual mechanism from calling any virtual functions in classes which either have not been constructed yet, or have been destructed already. What might happen if a pure virtual function of such an incomplete object were called?

The language specification (Arm 10.3) says that calling a pure virtual function from a constructor of destructor is undefined. This means that if `f()` is defined as pure virtual in class `C`, then calling `f()` from a constructor or destructor of `C` is an error.

For exmaple, let us assume that the designer of `Actuator` wanted to initialize the instance in the `deactivated` state. To do this he called `Deactivate()` in the constructor:

```
Actuator::Actuator()
{
    Deactivate();
}
```

But `Deactivate()` is defined as pure virtual in `Actuator`. So this is an error. In fact, it is an error that compilers can easilty detect, so most will issue some kind of error message. However, it is unreasonable to expect a compiler to detect all the myriad ways in which pure virtual functions can be invoked from constructors or destructors. The error can be hidden by a slight, and very reasonable modification to the example above.

```
class Actuator
```

```
{
  public:
    enum ActuatorState {on, off};
    Actuator();
    virtual void Activate() = 0;
    virtual void Deactivate() = 0;
    ActuatorState GetState() const;
  private:
    void Init();
};

Actuator::Actuator()
{
    Init();
}

void Actuator::Init()
{
    Deactivate();
}
```

Here, the designer has, quite reasonably, created a private `Init()` function which will be called by the constructor. `Init` calls `Deactivate` in order to initialize the device in the "deactivated" state.

The same bug still exists. `Deactivate()` will still be called within the context of the `Actuator` constructor. But this time, the compiler will probably not complain. Thus, when an `Indicator` is created, the call to `Deactivate` will attempt to invoke a non-existent implementation, and will very likely crash.

PURE VIRTUAL IMPLEMENTATIONS Sometimes it is convenient for pure virtual functions to have implementations. C++ allows this. The implementation is coded in exactly the same way as the implementation of any other member function:

```
void Actuator::Activate()
{
    /* Do something clever here */
}
```

There are not very many good reasons to supply implementations for pure virtual function. After all, a pure virtual function represents a pure interface whose implementation would make no sense in the context of the abstract class. But sometimes there are exceptions.

For example, assume we have a virtual function `IsValid` which checks the instance to see if it is in a valid state. The general form of such a function will be:

```
class MyClass : public MyBase
{
  typedef MyBase superclass;
  public:
    int IsValid() const;
    /* ... */
};

int MyClass::IsValid() const
{
    int retval = 0;
    if (superclass::IsValid())
    {
           if (/* I am valid */)
           {
                  retval = 1;
           }
    }
    return retval;
}
```

Note the call to the super class. When `IsValid` is called for an instance, it checks to see if the portion of the instance which is described by its base class (superclass) is valid. If so, it checks its own parts and returns 1 if they are ok, and 0 if they are not. Each `IsValid` function in each of the base classes repeats this procedure. Thus, when `IsValid` is called for an instance, the call is passed all the way up the inheritance hierarchy.

Now, lets presume that the base most class of the hierarchy is as follows:

```
class Validatable
{
  protected:
    virtual int IsValid() = 0;
};
```

In other words, the abstract base class describes the set of all classes which have `IsValid` functions.

Can the immediate derivatives of `Validatable` obey the `IsValid` protocol by passing the call up to their superclass (i.e. `Validatable`)? This would be very desirable since we don't want to have one convention for immediate derivatives of `Validatable`, and another for its indirect derivatives. If at all possible, we want all the derivatives of `Validatable` to use the same form for `IsValid()`.

Fotunately, the ARM(10.3) allows implementations of pure virtual function to be called explicitly by using their qualified name.

```
if (Validatable::IsValid()) ...
```

Thus, if we supply the following implementation for the

pure virtual function `Validatable::IsValid()`, then the immediate derivatives of `Validatable` can obey the superclass protocol.

```
int Validatable::IsValid()
{
    return 1;
}
```

PURE VIRTUAL DESTRUCTORS. There is one aspect of pure virtual functions which the language specification does not define very well. This is the behavior of *pure virtual destructors*. Pure virtual destructors are an oddity. They are, I suspect, an accident of syntax rather than a designed feature. They look like this:

```
class OddClass
{
  public:
    virtual ~OddClass()=0; // valid but strange.
};
```

A destructor is not a normal function, it cannot be inherited (ARM 12.4), it cannot be overridden and it cannot be hidden. A virtual destructor is not a normal virtual function. It does not share the same name as the base class destructor, and it is not inherited. A pure virtual function is *meant* to be inherited, its is the interface for a feature which is to be defined in a derived class. So what is a pure virtual destructor? It is not an interface for a destructor which is to be defined by a derived class, because destructors cannot be inherited. It is not a function without an implementation, becauseit will be called as the destructor for the class, and so it must be implemented. The only guaranteed feature of a pure virtual destructor is that it makes the class that contains it abstract.

Using Abstract Classes

Although programs are not allowed to instantiate abstract classes, in every other way they can be used to manipulate normal objects. Classes may contain pointers or references to abstract classes.

```
class ActuatorTimer
{
  public:
    ActuatorTimer(Actuator&);
    void SetTimer(int);
    int GetTimer() const;
    void Start();
  private:
    Actuator& itsActuator;
};
```

This declaration shows the interface of a class which will activate an actuator for a specified period of time, and then turn it off again. Any derivative of `Actuator` can be used as an argument to the `ActuatorTimer` constructor. The specified `Actuator` will be polymorphically activated and deactivated by the `ActuatorTimer` class.

Summary

Abstract classes provide a powerful design technique which promotes code re-use and polymorphism. According to Coplien: “The power of the object paradigm in supporting reuse lies in abstract base classes.”[5] Abstract classes are produced by factoring out the common features of the concrete classes of the application. Although such factorings are sometimes hard to find, the effort put into finding them is usually well worth the benefits of the extra maintainability and reusability. In general, common features should be factored out and moved as high as possible in the inheritance structure.

In C++, pure virtual function are used to specify the pure interfaces of abstract classes. An abstract class in C++ must have at least one pure virtual function. Although pure virtual functions typically have no implementation, C++ allows implementations to be given to them. The user must take care not to invoke pure virtual functions in the constructors or destructors of an abstract class.

Although abstract classes cannot be instantiated, they can be used in every other way to represent normal objects. They can be contained or passed by reference, and their interface can be used to invoke their intrinsically polymorphic behavior.

References

[1] Booch, *Object Oriented Design with Applications*, Benjamin/Cummings, 1991

[2] Rumbaugh, et. al. *Object-Oriented Modeling and Design*, Prentice-Hall,1991

[3] Lippman, *C++ Primer*, second edition, Addison-Wesley, 1991

[4] Wirfs-Brock, et. al. *Designing Object-Oriented Software*, Prentice-Hall, 1990

[5] Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992