Outline

1. Introduction
2. Class Design
   - Principles
   - Goals
   - Encapsulation
3. Relationships Between Objects
   - Has-A
   - Is-A
   - Is-Implemented-In-Terms-Of
   - Protected Inheritance
   - Summary
4. Slicing
5. Abstract Non-Leaf Classes
6. Bibliography
Object-based programming groups related data into objects and organizes member functions around that data.

Polymorphism is the key to moving from object-based programming to object-oriented programming.

- Greek: Poly- = multiple, -morph = form
- Polymorphism is the mechanism whereby objects take on multiple forms.
- Objects with multiple forms facilitate code reuse and extension.
SOLID Object-Oriented Design Principles

**Single Responsibility** Each class should have a single, well-defined responsibility.

**Open/Closed** Classes should be open for extension, but closed for modification.

**Liskov Substitution** A derived class should *always* be substitutable for its base class

**Interface Segregation** Clients shouldn’t have to depend on interfaces they don’t use.

**Dependency Inversion** Details should depend on abstractions, not vice-versa.

[Meyer, 1988], [Martin, 2002]
Goals of Class Design

Classes should:
- be complete and minimal. [Meyers, 1998]
- have interfaces that are as small as possible, but no smaller.
- reuse existing code whenever possible.
- hide implementation details from callers whenever possible.
- maximize encapsulation. [Meyers, 2000]
You can compare the relative encapsulation of two implementations by determining if changes to one implementation might lead to more or less broken code than the other implementation.

More encapsulation means more practical flexibility. [Meyers, 2000]
Encapsulation Example

Structure - Public Data

```cpp
struct Direction
{
    double mX;
    double mY;
};
```
Encapsulatioon Example

class Direction {
public:
    Direction(double x, double y) : mX(x), mY(y) {};
    double getX() const { return mX; }
    void setX(double x) { mX = x; }
    double getY() const { return mY; }
    void setY(double y) { mY = y; }
    void scale( double scale_factor)
    {
        mX *= scale_factor;
        mY *= scale_factor;
    }
private:
    double mX;
    double mY;
};
Encapsulation Example

Use of Accessors Internally

class Direction {
public:
    Direction(double x, double y) : mX(x), mY(y) {};
    double getX() const { return mX; }
    void setX(double x) { mX = x; }
    double getY() const { return mY; }
    void setY(double y) { mY = y; }
    void scale(double scale_factor) {
        setX(getX() * scale_factor);
        setY(getY() * scale_factor);
    }
private:
    double mX;
    double mY;
};
Encapsulation Example

Non-Friend, Non-Member Function

class Direction {
public:
    Direction(double x, double y) : mX(x), mY(y) {}
    double getX() const { return mX; }  
    void setX(double x) { mX = x; }
    double getY() const { return mY; }  
    void setY(double y) { mY = y; }
private:
    double mX;
    double mY;
};

void scale(Direction& v, double scale_factor) {
    v.setX(v.getX() * scale_factor);
    v.setY(v.getY() * scale_factor);
}
As developers, we are sometimes tempted to make a helper function a static member of a class because it “belongs” to a class.

**Problem:** Making a function a static member grants it access to member data and methods it may not need, thereby decreasing encapsulation.

**Guideline**
Don’t make a function a static member unless it needs access to non-public methods.
Composition vs. Inheritance

**Composition** A new object is made up of one or more objects that contain some functionality that is useful to the new class. The contained object’s *implementation* is reused, but its *interface* is not. This is also referred to as containment, aggregation, or layering.

**Inheritance** A new object is derived from an existing object. The parent object’s *interface* is reused, and its *implementation* may or may not be reused.
Has-A relationships are captured with composition. The new class is composed of member variables that contain functionality the class needs but doesn’t want to reimplement. This has the following consequences:

- The implementation of the existing class is reused but its interface isn’t propagated to reusing code.
- The implementation of the new class is hidden from client code.
- The implementation of the new class can be changed later without having to modify users of the class.

**Example**

‘Color’ contained by ‘Shape’
Is-A Relationships

Shape:
- Point location pt
  + virtual void draw() const = 0;
  + virtual void rotate(double angle);
  + void move(const Direction& move_dir);

Triangle:
- Direction to_pt_1
  - Direction to_pt_2
  + virtual void draw() const;
  + virtual void rotate(double angle);

Circle:
- double radius
  + virtual void draw() const;
  + virtual void rotate(double angle);

Ellipse:
- Direction to_other_focus
  + virtual void draw() const;
  + virtual void rotate(double angle);
Virtualness Defines What Gets Inherited

Pure virtual

Only the *interface* is inherited. An implementation must be provided.

- virtual void func() = 0;

Simple virtual

*Interface* and *default implementation* is inherited. The derived class can override the base class implementation. Remember, “Require no more, promise no less”. Consider using the Non-Virtual Interface (NVI) pattern.

- virtual void func();

Non-virtual

Interface and mandatory implementation is inherited. This represents an *invariant over specialization*. [Meyers, 1998]

- void func();
The problem with simple virtual functions is that they do two things at once:

- Specify Interface
- Customize Implementation

The NVI idiom separates these two roles by providing a public, non-virtual interface function and a non-public, virtual implementation function. [Sutter, 2005] [Sutter, 2001]

Guideline

Prefer non-virtual public functions and non-public virtual functions.
Non-Virtual Interface (NVI) Idiom

Base Class With Public Virtual Function

class Widget
{
    public:
        virtual int Process(const Gadget& gadget);
        // ...
}
Non-Virtual Interface (NVI) Idiom

NVI Base Class

```cpp
class Widget
{
public:
    //public, non-virtual interface
    int Process( const Gadget& gadget ); //calls DoProcess(const Gadget& gadget)
    //...

private:
    //virtual, non-public implementation
    virtual int DoProcess( const Gadget& gadget );
    //...
};
```
Redefining Non-virtual Member Functions

Don’t do it!

Why?
Redefining Non-virtual Member Functions

Ruins ‘invariant over specialization’ characteristic

*If Derived really must reimplement doSomething(), and if every Base object really must use Base’s implementation of doSomething(), then Derived really isn’t a Base.*

*If Derived really has to inherit publicly from Base and Derived really has to reimplement doSomething(), then doSomething() really isn’t an invariant over specialization.*

*If Derived really is a Base and if doSomething() really represents an invariant over specialization, then there really isn’t a good reason for Derived to reimplement doSomething() and shouldn’t try to do so. Something has to give.*

[Meyers, 1998]

Name Hiding

*Anytime you redefine an overloaded function name from the base class, all the other versions are automatically hidden in the new class.* [Eckel, 2000]
Never use public inheritance to capture Is-Almost-A relationships. Is-Almost-A is like being “kind of pregnant”.

Prefer containment to inheritance. Containment:
- provides greater relative encapsulation.
- reuses only functionality you need from the contained class.
- avoids referencing functionality in other parts of the class that you don’t need.
This is similar to Has-A, so most of the time, you’ll want to use composition.

Occasionally, private inheritance should be used instead.

With private inheritance, all public and protected members of the base class become private in the derived class. The interface to the base class is hidden from callers of the derived class. Instances of the derived class can’t be treated as instances of the base class.

Use private inheritance if you need:
- access to protected member functions.
- to override virtual member functions.
What does protected inheritance mean?

Protected inheritance means Implemented-In-Terms-Of to other classes, but Is-A for derived classes and friends. [Eckel, 2000]

Controlled polymorphism means Is-A, but in certain code only. [Sutter, 1999]

Sound complicated? It is. If your design requires this, it is probably too complex. Protected inheritance has been included in the standard primarily for completeness.
What Do You Want To Reuse?

**Implementation Only**
- Composition
- Private inheritance when you need to access protected members or override virtual functions.

**Interface Only**
- Public inheritance and pure virtual functions.

**Interface and Default Implementation**
- Public inheritance and simple virtual functions.
- Consider using the NVI idiom.

**Interface and Mandatory Implementation**
- Public inheritance and nonvirtual functions.
Slicing occurs when polymorphic objects are manipulated *by value* rather than *by reference*.
Abstract Non-Leaf Classes

If you find yourself wanting to derive from a leaf class, separate out the interface into an abstract base class and derive both the old and new classes from the abstract class.

If the interface is worth deriving from, it’s worth abstracting. [Meyers, 1996]

Going to the extra work of abstracting out the interface keeps derived classes from inheriting too much from the base class.

Guideline

Derive only from abstract classes.
Bibliography


