Class Design

When designing classes, there are three different design errors you can fall into:

- **The Data Warehouse Trap**: An object is ***not*** a repository for data that the rest of your program will use. Your objects should contain both data and the methods that work on that data.

  If you find yourself with classes that have lots of data elements and almost every element has a **get()** or **set()** method, you may be falling into this trap.

  Objects should ***do*** something, not just sit there!

- **The Spectral Object Trap**: An object is ***not*** just a collection of methods that you pass data to. If a class has no data elements, and if most of the methods in the class are **static**, then what you have is a procedural ***module***.

  An object contains both data and the operations that are performed on the data. Objects with no data are ghosts (specters)!

- **The Multiple Personality Trap**: An object should model ***one*** abstraction. It should be one kind of thing. Every class should be highly ***cohesive***, meaning that every data element and every method should contribute to that one abstraction.

Class Interfaces

In evaluating the public interface of your class, ask yourself the following questions:

1. **Does the interface represent, or model, a single abstraction?**

   Classes are not collections of autonomous methods. Make sure each method contributes to the intended behavior of your object.

2. **Does each method represent, or model, a single operation? That is, are the methods themselves internally cohesive?**

3. **Does the class do everything it needs to do, and no more?**

   If programs that use your objects rely primarily on get() and set() methods, this may be a sign that your interface is not complete, that client objects are having to process your object's data.

   If there are methods that are not used, don't clutter your interface with them.

4. **Is your interface easy to use?**

   Your interface should be designed around the problem your class solves, not the details of your implementation.

   Methods that require long argument lists or complex calculations clamor for simplification. Ex: input in Hex, or an addresses in decimal. Ease of use must be evaluated from the user's perspective.

5. **Use a checklist of common operations that all classes should understand.**

   Use your checklist to make sure that you have considered how objects are constructed, copied (or assigned), and destroyed.

   Make sure you've dealt with the difference between identity and equality if ordering or sorting objects is important. When you design your class, consider including methods to test, debug, and print the state of your objects.

Class Design Guidelines

1. The only members of the public interface of a class should be the operators of the class.

   Information hiding required reinforces the development of representation independent designs.

2. An instance of a class A should not send a message directly to a component of B.

   Encapsulation: prohibits accessing class instances used as part of the representation of this class.

This first two rules enforce the idea that a class is characterized by its set of operations and not by its representation.

3. An operator should be public if and only if it is to be available to users of instances of the class.

   All other should be private or protected.

4. Each operator that belongs to a class either accesses or modifies some data of the class.

   Requires that to belong to the class, each operator must represent a behavior of the concept being modeled by the class.

The first four guidelines address the form and use of the class interface; and give the designer directions for developing and separating the class interface and representation.

5. A class should be dependent on as few other classes as possible.

   Constrains the designer to link a class with as few other classes as possible.

   If a class being designed will need many of the services of another class, perhaps that functionality should be part of the new class.

6. The interaction between two classes should be explicit.

   Intended to reduce, and perhaps, eliminate global information. All that is needed should be passed as parameter. Also helps design standard interfaces.

7. Each subclass should be developed as a specialization of the superclass with the interface of the superclass becoming a part of the public interface of the subclass.

   Prohibits the use of inheritance to develop the representation of a new class rather than its interface (private inheritance in C++). Consider using composition.

8. The root class of an inheritance structure should be an abstract model of the target concept.

   Encourages designers to develop inheritance structures of classes which are specializations of an abstraction. Leads to more reasonable subclasses and to clear-cut differences between subclasses.

The second set considers the relationship of the class to other classes

Designing Reusable Classes

The design rules given are a way of converting specific solutions into reusable abstractions, not a way of deducing abstractions from first principles.

1. Recursion Introduction

If one class communicates with a number of other classes, its interface to each of them should be the same.

2. Eliminate Case Analysis

It is almost always a mistake to check the class of an object.

```
if (anObj.class == ThisClass)
        anObj.foo()
else
        anObj.fee()
```

This could be replaced with a message to the object whose class is being checked (anObj). Methods will have to be created in the various possible classes of the object to respond to the message, and each method will contain one of the cases that is being replaced.

3. Reduce the Number of Arguments

Messages that have a dozen or more arguments are hard to read (except constructors). Messages with smaller number of arguments are more likely to be similar to some other message.

The number of arguments can be reduced by breaking a message into several smaller messages or by creating a new class that represents a group of arguments.

4. Reduce the Size of Methods

It is easier to subclass a class with small methods, since its behavior can be changed by redefining a few small methods instead of modifying a few large methods.

5. Hierarchies should be Deep and Narrow

6. The Top of the Hierarchy should be Abstract

7. Subclasses should be specializations

8. Split Large Classes

Large classes should be viewed with suspicion and held to be guilty of poor design until proven innocent.

9. Factor Implementation differences into subcomponents

If some subclasses implement a method one way and others implement it another way then the implementation of that method is independent of the superclass. It is likely that it is not an integral part of the subclasses and should be split off into the class of a component.

10. Separate Methods that do not Communicate

A class should almost always be split when half of its methods access half of its instance variables and the other half of its methods access the other half of its variables.

Restructure a Hierarchy

With extensive expansion, the inheritance hierarchy may become less suitable and a restructuring may be necessary.

Example: Assume with have a class Person and subclasses Male and Female that inherit some common attributes from Person; but redefine the functions jump(), and dance().

i) Assume we add two individuals: James, Peter

ii) Describe the them as old men having different characteristics (dance style)

How will the new class be related to the existing ones ?

1. OldMale as subclass of Person and define the differences.

2. OldMale inherits from Male and redefine the dance method inherited.

3. Create another class YoungMale that also inherits from Male and define the dance functions at that level.

To determine the appropriate restructuring alternative consider the following factors:

- How the classes are to be used
- How will further modifications be handled
- What do we really want to model
- How important is a clear understanding
- How difficult is it to restructure the classes
- What is the price of restructuring
- How does it affect the applications

Building Good Hierarchies

1. Model a "Kind_Of" / "Is_A" hierarchy: subclass should support all the responsibilities defined in their superclass, and possibly more.

When a subclass correctly supports all the responsibilities defined in their superclass, its responsibilities will completely encompass those of its superclass.

When the subclass includes only part of the responsibilities, create an abstract class with all the responsibilities that are common.

2. Factor common responsibilities as high as possible: If a set of classes all support a common responsibility, then they should inherit it form a common superclass (probably abstract).

DrawingElement

Text **Line** **Ellipse** **Rect** **Group**

Ellipses, Rects, Line are responsible for maintaining two attributes: **width**, **color** of the line by which they are drawn. Since the others do not support that responsibility those attributes cannot be added to the superclass.

DrawingElement

Text **LinearElement** **Group**

Line **Ellipse** **Rectangle**

Rectangle and Ellipses share a responsibility associated with the ability to be **filled** with a color ?

Add polygon ?

The main benefit of carefully designing with Abstract Classes evidences itself when you wish to add new functionality to the existing application.

More behavior can be reused with abstract classes. Using ACs means you have factored out as much common behavior as you could.

3. Make sure that Abstract classes do not inherit from concrete classes: abstract classes support their responsibilities in implementation independent ways. They should never inherit form a concrete class, which may specifically depend upon implementation.

4. Eliminate classes that do not add functionality.

Example: Assume that we have a Class defined to represent Birds. Among its behavior we have a method **flyingRange()** because it was important in the original specs of the problem.

Suppose that we decide to add a class to represent the Penguins. What would the best solution for this extension ?

Composition vs. Inheritance

Example: Suppose we have a class to represent a **Course**. The attributes includes the **number** and the **title**, and the behavior includes a method **print** all course information.

The attributes does not include the **time** of the course and the **teacher** of the course because no all existing courses are scheduled to be taught.

How do we define a class to represent a course that is scheduled to be taught ?

Example: Design a Symbol Table for string values. Since the class Dictionary already exists we can either make the ST a subclass of D or let the ST contain a D. Advantages and disadvantages:

Inh: is shorter and may be easier to develop (good for prototyping)

Inh: does not prevent users from manipulating a Symbol Table as though it were a Dictionary

Comp: the fact that a Dict. is used is merely an implementation detail, it would be easy to reimplement the ST with a different technique with minimal impact on the users of the ST.

Comp: defines a clear interface to the ST, only the operations explicitly provided are permitted

Inh: may provide free useful inherited operations.

Inh: may be more difficult to understand because some methods may not be proper for the ST.

Example: Suppose that and existing class provides 80% of the functionality needed for a new requirement and that this new functionality can be added with a few new methods.

Should the programmer simply change the base class to provide the new functionality **or** create a new class and used inheritance to gain access to the existing structure ???

If the existing class is widely used by other applications then its probably best to leave the class alone. Especially true if new data is added. There is always a risk that new changes may introduce errors.

If the changes are totally transparent (replacing an algorithm with a more efficient one) then a single change can have widespread, and probably welcomed, advantages.

Example: A digital timer consists of two display panes, one for hours and one for minutes.

Each display will hold a value between 0 and some preset upper limit.

Users must be able to initialize the timer by initializing each display's value to 0.

Users should also be able to :

- Increment the minutes and hours
- Set the value of each display
- Ask the timer to show its value by showing the value of each display.