Chapter 6: Working Classes

Class Foundations: Abstract Data Types (ADTs)

- ADT is collection of data and operations to do work on that data
- Without understanding ADTs, programmers create classes that are "classes" in name only - in reality, they are little more than convenient carrying cases for loosely related collections of data and routines.
- With an understanding of ADTs, programmers can create classes that are easier to implement initially and easier to modify over time.
- Main idea is giving yourself the ability to work in the problem domain rather than at the low-level implementation domain.

Example of the Need for an ADT

- Suppose writing a program to control text output to the screen using a variety of typefaces, point sizes, and font attributes.
- If you use an ADT, you'll have a group of font routines bundled with the data they operate on. The collection of font routines and data is an ADT
- If not using an ADT, you'll need an ad hoc approach to manipulating fonts
    - currentFont.size = 16
- If you have a collection of library routines, code may be slightly more readable
    - currentFont.size = PointsToPixels(12)
- Or could provide a more specific name for the attribute
    - currentFont.sizeInPixels = PointsToPixels(12)
- But what you cant do is have both currentFont.sizeInPixels and currentFont.sizeInPoints, because if both in play, then currentFont wont have any way to know which of the two to use.

Benefits of Using ADTs

- The problem with the ad hoc approach isn't that its bad practice. Rather, its that you can replace the approach with a better programming practice that produces these benefits:
    - You can hide implementation details
    - Changes don't affect the whole program
    - You can make the interface more informative
    - Its easier to improve performance
    - The program is more obviously correct
    - The program becomes more self-documenting
    - You don't have to pass data all over your program
    - You're able to work with real world entities rather than low level implementation structures
- currentFont.SetSizeInPoints(sizeInPoints)
- currentFont.SetSizeInPixels(sizeInPixels)
- currentFont.SetBoldOn()
- currentFont.SetBoldOff()

- currentFont.SetItalicOn()
- currentFont.SetItalicOff()
- currentFont.SetTypeFace(faceName)

More Examples of ADTs

- Light
    - turn on
    - turn off
- Blender
    - turn on
    - turn off
    - set speed
- Fuel Tank
    - fill tank
    - drain tank
    - get tank capacity
    - get tank status
- Stacks, Lists, Queues

ADTs and Classes

- One way of thinking of a class is as an abstract data type plus inheritance and polymorphism (same interface for differing underlying data types) --> __repr__, __add__, etc definitions
- Good Class Interfaces
- Good Abstraction
    - Every routine in the interface is working towards consistent end within the scope of the abstraction
- Bad Abstraction
    - Miscellaneous collection of functions
- Each class should implement one and only one ADT, or if you can't determine what ADT the class implements, it's time to reorganize the class into one or more well-defined ADTs

```
Bad
    Public:
    void addEmployee(Employee)      1 Employee
    void removeEmployee(Employee)   1

    Employee NextItemInList()       2 List Item
    Employee FirstItem()            2
    Emplyee LastItem()              2

Good
    Public:
    void addEmployee(Employee)      1 Employee
    void removeEmployee(Employee)   1
    Employee NextEmployee()         1
    Employee FirstEmployee()        1
    Employee LastEmployee()         1

    Private:
    ListContainer m_EmployeeList    2 the class used is now hidden
```

- Provide service in pairs, if you have an operation, there is usually an opposite or inverse operation
- **Make interfaces programmatic, not semantic**
    - Programmatic part consists of data types and other attributes that can be enforced by a compiler
    - Semantic part is the assumptions about how the interface will be used that cannot be enforced by a compiler (proper initialization, sequences)
        - The semantic interface should be well documented
        - Look for ways to convert semantic elements to programmatic by using Asserts or other techniques
    - **Beware of erosion of the interfaces abstraction under modification**
    - **DONT ADD PUBLIC INTERFACE ROUTINES THAT ARE OUT OF LINE WITH ORIGINAL PURPOSE AND CLASS-IFICATION**

Good Encapsulation

- Minimize accessibility
    - Don't expose member data in public
        - use getters and setters
    - Avoid putting private implementation details into a classes interface
    - Don't make assumptions about the class's users
        - should be designed and implemented to adhere to the contract implied by the class interface
    - Avoid friend classes (actual thing, not a metaphor)
    - Don't put a routine into the public interface just because it uses only public routines
    - Favor read-time convenience to write-time convenience
        - code is read far more times than it is written
    - Don't do semantic violations
    - **"It isn't abstract if you have to look at the underlying implementation to understand what's going on"**

Design and Implementation Issues

- Defining good class interfaces goes a long way towards creating a high quality program
- The internal class design and implementations are also important.

Containment - ("has a" Relationships)

- Containment is the simple idea that a class contains a primitive data element of object.\

- More is written about inheritance b/c its difficult, but Containment still slaps
- "has a"
    - employee has a phone number
    - has a name
    - has a tax ID
- Implement "has a" through private inheritance as a last resort
- Limit yourself to 7+-2 data members

Inheritance - ("is a" Relationships)

- Inheritance is the idea that one class is a specialization of another class.
- The purpose of inheritance is to create simpler code by defining a base class that specifies common elements of two or more derived classes
- The common elements can be
    - routine interfaces
    - implementations
    - data members
    - data types

Inheritance as a Whole

- When deciding to use, need to make several decisions:
    - For each member routine, will the routine be visible to derived classes? Will it have a default implementation? Will the default implementation be overridable?
    - For each data member (including variables, named constants, enumerations, and so on), will the data member be visible to derived classes?
- How to make these decisions:
    - Implement "is a" through public interface
        - if the derived class isn't going to adhere COMPLETELY to the same interface contract defined by the base class, inheritance is not the right technique
    - "Design and document for inheritance, or prohibit it"

Inherited routines

- Abstract overridable routine: means the derived class inherits from the routines interface, but not its implementation
- Overridable routine: means that the derived class inherits the routines interface and a default implementation, and it is allowed to the default implementation
- Non-overridable routine: means that the derived class inherits the routines interface and its default implementation and is not allowed to override the routines implementation
- **"don't reuse names of non-overridable base-class routines in derived classes"**

Other rules of thumb:

- Move common interfaces, data and behavior as high as possible in the inheritance tree. The higher you move interfaces, data and behavior, the more easily derived classes can use them

- Be suspicious of classes of which there is only one instance. A single instance might indicate that the design confuses objects with classes. Can the variation of the derived class be represented in data rather than as a distinct class?
- Be suspicious of base classes of which there is only one derived class
- Be suspicious of classes that override a routine and do nothing inside the derived routine
- **Fix the base class to handle variations in derived**

Multiple Inheritance

- **Is a thing, but gets overly complex easily**
- **(very short section on this saying to stay away from it, lmao)**

Member Functions and Data

- Guidelines:
  - Keep number of routines in a class as small as possible
    - higher number of routines per class are associated with higher fault rates
  - Disallow implicitly generated member functions and operators you don't want
  - Minimize the number of different routines called by a class
    - Higher fault rates correlate to the total number of routines called from within a class
    - The more classes a class uses, the higher its fault rates
  - Minimize indirect routine calls to other classes
    - account.ContactPerson().DaytimeContactInfo().PhoneNumber() = BAD
- Basically:
  - Minimize the number of kinds of object instantiated
  - Min number of different direct routine calls on instantiated objects
  - Min num of routine calls on objects returned by other instantiated obj

Constructors

- Guidelines:
  - Initialize all member data in all constructors
  - Enforce singleton property by using a private constructor
  - Prefer deep copies to shallow copies until proven otherwise