

6.6 Designing Classes

- Design of an individual class
- Questions:
 - is it one or two classes?
 - which operations?
 - how to name the operations?
- Goals
 - coherent classes / low coupling between classes
 - right size
 - understandability
 - extensibility, changeability
 - separation of concerns among operations
 - easy subclassing

Command-Query Separation

- Rule: Each operation should be either a *command* or a *query*
- In other words:
 - “Asking a question should not change the answer”
 - Functions should have no side effects
- Command
 - an operation that changes the state of an object or does input / output
 - does not return a value
 - often called “set...”
- Query
 - an operation that returns a value
 - does not change any attributes
 - often called “get...”

Command-Query Example

- The following is an operation that is both command and query

```
int n() {                // don't do this
    a = a + 1;
    return a;
}
```

- This is a problem because...
a = 1;
x = n() + n();
- has a different result than:
a = 1;
x = 2 * n ();
- And this is bad because it's very confusing!

Number of Arguments

- How many arguments should an operation have?
 - lots?
 - can do many different things
 - flexible and general
 - few?
 - easy to use
 - easy to implement, easy to change
- Rule: Arguments should only include operands, and not options.
 - operands
 - are essential to the operation
 - options
 - can be replaced by default values
 - can be modeled as attributes that can be set

Number of Arguments: Example

- Printing a document — bad design
printer.print(document, paperSize, color, resolution)
- Printing a document — good design
printer.print(document)
printer.setPaperSize(paperSize) //sets attribute paperSize
printer.setColor(color) //sets attribute color
printer.setResolution(resolution) //sets attribute resolution
- Solution:
 - several operations instead of one
 - document is an operand of print():
 - it's different for each call
 - the other arguments are options of print():
 - they change only once in a while
 - and so they should be set separately as attributes

What is a Good Class?

- ...and what should not be a class?
- A class should be named with a noun
 - Example: "class Book"
- A class should not represent a task or operation
 - Don't do this: "class FindABook"
 - Don't have classes described as "performing" something
- A class should contain several related operations
 - A class with only one operation is usually a bad thing
- A class should contain at least one attribute
 - exceptions: Java interfaces, some subclasses
- A class should represent a single abstraction

Class Size

- How many operations should a class have?
- It depends...
 - most classes shouldn't have more than 20 operations
 - some very important ones may have more
 - JComponent in Swing
- Smaller classes
 - can be understood more easily
 - can be reused more easily
- If a class has a lot of operations, check:
 - if it represents a single, coherent concept
 - otherwise: split it up
 - if all the operations work on a common set of attributes
 - otherwise: split it up
- Inheritance often causes very large classes
 - operations add up over the generations

Class Size: Adding an Operation

- When adding an operation to a class, check...
 - that it does not do the same thing as an existing operation
 - that it does not violate any assumptions made by the existing operations
 - that it is relevant to the concept represented by the class
- When an operation doesn't fit very well into your class, you might instead...
 - put it into a class that delegates to the existing class
 - if it's just a combination of existing operations
 - create a similar class
 - and factor out the common parts into a third class
 - throw it out

Method Size

- How many lines of code should a method have?
- Most methods should be small
 - Most methods should be less than 20 lines
 - Large methods should often be split into two
 - or even divided up among several classes
 - Large methods are often a symptom of incoherent classes
- One method should do one coherent task
 - ideally, should include at most one if-statement
 - ideally, should include at most one loop
- Smaller methods
 - are easier to understand
 - can be reused more easily

Immutable Objects

- Immutable objects cannot change their state
 - They are created
 - and then they never change
- Examples:
 - java.lang.String is immutable
 - java.lang.StringBuffer is mutable
 - Sets
 - immutable: you can't add an object to an existing set, you can only create a new set
 - mutable: you can add and remove objects
- Advantages of immutable objects
 - Observers don't need to be updated since immutable objects can't change
 - Can't be corrupted by other threads
 - Are required as keys in hashtables
 - because hash code changes when object changes

Immutable Objects

- How to implement immutable objects?
 - need to have separate class
 - since add, remove are not available (set example)
 - but can't inherit from each other
 - MutableSet is not a kind of ImmutableSet
 - ImmutableSet is not a kind of MutableSet
- Solution:
 - immutable class delegates to mutable one
 - ImmutableSet has a private attribute of type MutableSet, which is never changed once created
 - but does not provide any mutating operations
 - and does not give out the mutable object
 - might have to be final (i.e. cannot be subclassed)
 - otherwise subclasses might add mutating operations

Inheritance

- How to use inheritance?
- Rule: Use only if there is a "is kind of" relation
 - "A inherits from B" means: "A is a kind of B"
 - do not use for "is part of" relation
- What's the alternative to inheritance? – Delegation!
 - Instead of "A inherits from B": "A has an association to B"
 - Instead of inheriting operations, we delegate them:

```
class A {
    doSomething() {
        b.doSomething();
    }
}
```
 - often more flexible than inheritance

Inheritance

- Rule: Inheritance is good if one needs substitutability
 - A inherits from B
 - There are variables of type B
 - Instances of A are bound to these variables
 - Example:
 - Vector. add(Object o)
 - you can put Strings into the Vector too
- Rule: Don't have more than 5 levels of inheritance
 - each subclass should add or change behavior
 - don't overclassify
- Rule: a superclass needs to document what subclasses can override, and how to do it
 - otherwise subclasses will inadvertently break stuff
 - after all, subclasses can access all private features

Inheritance

- Purposes of inheritance
 - reuse an implementation
 - conform to a type
- When to use inheritance?
 - If you want to reuse an implementation only
 - use delegation
 - If you need type conformance only
 - use interface inheritance
 - If you need both type conformance and implementation reuse
 - use class inheritance

Equality Operator

- In Java:
 - boolean `Object.equals(Object)`
- It's often useful to define equality
 - Example:
 - two Strings that are not identical (are different objects at different memory locations)
 - ...can still be equal (contain the same characters in the same sequence)
- Equality used for
 - sets
 - to prevent multiple entries
 - hashtables
 - to identify keys
 - thus: if two objects are equal, they must have the same hash code

6.7 Code Reuse

- Idea: Don't invent the wheel twice
 - Reuse code written by others
- Process
 - Search for reusable code
 - Check whether it's what you need
 - Check whether it works
 - Learn how to use it
 - Integrate it into your code
- Need to be flexible
 - adapt requirements to what's out there
 - adapt design
 - often cheaper to change your mind about requirements than to redevelop code you could have reused

Code Reuse

- Advantages
 - saved effort
 - higher quality
 - many open source libraries are much better than anything you could write
 - standardization
 - other people may be using the same library
- Kinds of reusable code
 - Libraries
 - Frameworks
 - Components
 - Operating System Functions
 - Open Source Code

Reusable Code: Libraries

- Class or procedure libraries
- Well-documented set of related classes
- “Base” library
 - Java, C++ standard libraries
 - contain basic classes: String, Collection...
 - and sometimes much more
 - Java: Gui, networking,...
- Special purpose libraries
 - Algorithm libraries
 - Network protocols, database access, Gui, ...
- Problem
 - Often hard to combine several libraries
 - conflicting assumptions

Reusable Code: Frameworks

- Similar to libraries
 - sets of closely related classes
 - frameworks define overall control-flow
 - libraries don't
- Extension points
 - Where users can “plug in” their own classes
 - through inheritance
 - through implementation of empty method bodies
 - through configuration parameters
- Examples
 - Java Swing (kind of...)
 - Component frameworks
 - runtime environments for components

Reusable Code: Components

- Idea: components can easily be plugged together without having to change them
 - components define what they provide and what they require
 - connectors
 - self-description
- Component standards
 - Dot-Net
 - COM
 - Enterprise Java Beans
 - Java Beans
- A component standard includes
 - a framework (including library)
 - standard properties of components

Reusable Code: Operating Systems

- OSes often include libraries
- MS Windows
 - Win32, MFC
 - Dot-Net
- Advantages of OS functions
 - fast
 - reliable
- Disadvantages of OS functions
 - may be hard to use
 - platform-dependent

Reusable Code: Source Code

- Advantages
 - platform-independent
 - can be adapted
- Disadvantages
 - compiling often difficult
 - different language and compiler versions
 - hard to understand
 - may be incomplete

What to look for in Reusable Code?

- Requirements
 - does the right thing
- Architecture
 - does it in a way that fits your system
 - architectures are often incompatible
 - central vs. distributed
 - functional vs. imperative
 - event-based vs. procedure-call based
 - naming conflicts
 - classes with same names
- Not buggy
- Understandability
 - how difficult to learn?
- Adaptability
 - inheritance, configuration parameters...