Problems in Software Design

Overview

- Sample of software design issues
- Class hierarchies
- Protected variations
- Design patterns
- Many more ...

Class Hierarchies

- Superclass-subclass relationships
- Domain model: relationship between concepts
  - "An instance of X is also an instance of Y"
- Design model: relationships between software entities (classes)
  - Implemented in programming languages through inheritance
- How do we ensure correctness of inheritance hierarchies?

Inheritance

- Java: class B extends A { ... }
  - Single inheritance (one superclass)
- C++: class B : public A { ... }
  - Multiple inheritance (many superclasses)
- Every member of A is inherited by B
  - if a field f is defined in A, every object of class B has an f field
  - if a method m is defined in A, B inherits m
- B may declare new members
- B may override inherited methods

Example

class Rectangle {
    private double h,w;
    public Rectangle(double h,double w) { ... }
    public double getHeight() { return h; }
    public double getWidth() { return w; }
    public double area() { ... }
}

class SwissRectangle extends Rectangle {
    private int hole_size;
    public SwissRectangle(double h,double w, int hs) { ... }
    public void shrinkHole() { hole_size--; }
    public double area() { ... }
}

Example

- A SwissRectangle object has
  - Fields h, w, hole_size
  - Methods getHeight, getWidth, shrinkHole, area
  - area() is overridden in the subclass
- Java: if x is of type Rectangle, it may point to instances of Rectangle or instances of SwissRectangle
- C++: if x is of type Rectangle*, it may point to instances of Rectangle or instances of SwissRectangle
**Polymorphism**
- A variable may refer to (point to) objects of different classes
- Calls through this variable may invoke different methods
- Call: `x.area()` in Java, `x->area()` in C++
  - If `x` points to a Rectangle object, this call invokes method `area()` in Rectangle
  - If `x` points to a SwissRectangle object, invokes the overriding method `area()`
- C++: `area()` must be declared "virtual"
- Java: all methods are "virtual" by default

**Calls to Abstract Methods**
- Java: abstract class `X`:
  ```java
  abstract public void m();
  ```
- Method `foo(X x) { x.m(); }`
- C++: class `X`:
  ```cpp
  public: virtual void m() = 0;
  ```
- No keyword "abstract" in C++
- Method `foo(X* x) { x->m(); }`
- Invokes a non-abstract method in some subclass of `X`

**Liskov Substitution Principle (LSP)**
- A subclass must be substitutable for its superclasses
  - Named after Barbara Liskov (MIT)
- E.g.: class `A`, subclass `B`
  - In class `X`: method `m(A a) { ... }` (Java)
  - In class `X`: method `m(A* a) { ... }` (C++)
- If `m` behaves correctly when given an `A` object, it should also behave correctly when given a `B` object
  - Without `m` knowing about the existence of `B`

**Liskov Substitution Principle (LSP)**
- It should be possible to substitute a `B` object for the `A` object
  - Protection against variability of subclasses
- Polymorphism: if we add a new subclass of `A`, we don’t need to recompile `m`
  - Programming language mechanism
- LSP is stronger: if we add a new subclass of `A`, `m` will still be correct

**Classis Example**
```java
class Rectangle {
    protected double h, w;
    protected Point top_left;
    public void setHeight(double x) { h = x; }
    public void setWidth(double x) { w = x; }
    public double getHeight() { return h; }
    public double getWidth() { return w; }
    public double area() { return h*w; }
}
```
- This Rectangle is different from the one in the previous example
- Suppose we have written a lot of code that uses Rectangle (e.g. graphics code)

**Adding a Square**
- The customer decides that we need a new class `Square`
- A square is a kind of rectangle, right?
- class `Square` extends `Rectangle`
- With polymorphism: don’t have to recompile existing code
  - e.g., `void m(Rectangle x) { ... }` does not have to be recompiled
### Problems
- Square doesn’t really need both w and h
  - Wasted memory (relatively minor issue)
- Square inherits setHeight and setWidth: incorrect behavior

```java
Square s = new Square();
Rectangle r = s;
...r.setHeight(5);
r.setWidth(10);
???
```

### A Solution
- “Guards” in client code
  - Before changing the size of a Rectangle, check if it is a Square

```java
Rectangle r;
...r.setHeight(5);
if (r instanceof Square) r.setWidth(5);
```
- For client code: increases coupling, reduces cohesion, makes it fragile
  - Bad idea

### Another Solution
- Override setHeight and setWidth in Square

```java
class Square extends Rectangle {
    public void setHeight(double x) {
        h = x; w = x;
    }
    public void setWidth(double x) {
        h = x; w = x;
    }
}
```

### More Problems
- When we only had Rectangle objects, this was valid code
- But now it may break. Who’s to blame?
- The programmer of m is justified in writing this code
- There is something wrong with Square ...

### Back to LSP
- If m was written correctly with respect to a superclass, it should also be correct for the subclass
- Square violates LSP

**Postcondition for Rectangle.setHeight:**
- \( h = x \) and \( w = w_{old} \)

**Postcondition for Square.setHeight:**
- \( h = x \) and \( w = x \)

The subclass postcondition does not imply the superclass postcondition

### LSP is about Behavior
- The behavior of Square.setHeight does not conform to the behavior of method Rectangle.setHeight
- Inheritance should be more than just "is-a-kind-of" relationship
- Should guarantee conformance of behavior
- Often used term: behavioral subtyping
Using the LSP

- Warning flag
  - It may be OK to violate it, but the violation should be examined carefully
- Depends on the clients of the hierarchy
  - e.g. what if we have a program in which height and width are never changed?
  - "Square is not a subclass of Rectangle" has been the source of many debates in the OO community

Ensuring LSP

- One way: consider preconditions and postconditions
  - Whenever a subclass inherits or overrides an operation from a superclass
  - Precondition for the superclass should imply the precondition for the subclass
  - Postcondition for the subclass should imply the postcondition for the superclass

A Simple Example

- class Employee, with subclass Manager
- Operation double calcBonus(int eval)
  - Parameter: performance evaluation
  - Calculates a bonus percentage
  - Defined in Employee; overridden in Manager
- The operation in the subclass
  - should not expect something more restrictive
  - should not produce something less restrictive

Preconditions and Postconditions

- Employee.calcBonus:
  - Precondition: \(0 \leq \text{eval} \leq 5\)
  - Postcondition \(0\% \leq \text{bonus} \leq 20\%\)
- \(0 \leq \text{eval} \leq 5\) should imply the precondition for Manager.calcBonus
  - e.g. should not be \(1 \leq \text{eval} \leq 3\)
- Postcondition for Manager.calcBonus should imply \(0\% \leq \text{bonus} \leq 20\%\)
  - e.g. should not be \(0\% \leq \text{bonus} \leq 30\%\)

Another Example

- class Polygon
  - public void move(int x, int y) { ... }
  - public void addVertex(Point p) { ... }
  - public int numVert() { return vertices.size(); }
  - private Set vertices;
- class Triangle extends Polygon
  - public void addVertex(Point p) { }
- Need to override addVertex to do nothing
- Postcondition for addVertex
  - in Polygon: \(|\text{vertices}| = 1 + |\text{vertices}_{\text{old}}|\)
  - in Triangle: \(|\text{vertices}| = |\text{vertices}_{\text{old}}| = 3\)

Possible Solution

- LSP is violated for Triangle
  - Also for subclasses Rectangle, Hexagon, etc.
- One solution: two subclasses of Polygon
  - FixedSidedPolygon, with its own subclasses Triangle, Rectangle, etc.
  - VariableSidedPolygon in which addVertex in defined
  - addVertex is not defined in Polygon
Protected Variations

Principle of Protected Variations

- Fundamental problem in design: current and future variations
- Protect the rest of the system from these variations
- Typical mechanisms for protection
  - Encapsulation
  - Abstraction
  - Polymorphism
  - Indirection

Encapsulation

- Basic mechanism, typically in the programming language
- Group related entities into a single unit, and provide a restricted external view of the unit
- First form of encapsulation: procedures
  - Encapsulates a set of statements
  - External view: name and parameters
  - Protection against variations in algorithm

Object-Oriented Encapsulation

- Packaging of operations and attributes
- Attributes represent internal state that is not directly accessible
  - Hidden behind a "wall" of operations
- State is accessible and modifiable only via the operations
- Protection against
  - Changes in data representation
  - Changes in algorithm

Example

- Robot in a maze
  - Internal attributes: location, direction
  - Operations: turnLeft, turnRight, advance, getLocation, facingWall
  - Location may be internally represented as Cartesian or polar coordinates
  - Changes in the internal representation do not affect any other piece of code

Abstraction

- Common theme in software design
- Low-level abstractions
  - Procedural abstractions: subroutines
  - Data abstractions: e.g. classes
    - Protect against variations within the class
- Higher-level abstractions for object-oriented design
  - Protection against multiple classes
Using Abstractions

- Principle: write the client code against an abstraction of the server
- In case of current or future variations of the server

Abstraction Through Subclasses

- Write the client against a superclass
- Often an abstract superclass
- Variations in the server: introduce a new subclass
- If a new kind of server is added, client code does not need to be changed
- Protection against new kinds of servers

Example

- An application needs to draw shapes (circles and rectangles) on the screen
- Possible solution in C
  
  ```
  enum ShapeType {circle, square};
  struct Shape { ShapeType t; }
  struct Circle { ShapeType t; double radius; Point center; }
  struct Square { ShapeType t; double side; Point topleft; }
  ```

Example: Solution in C

```c
void DrawAll(struct Shape* list[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        struct Shape* s = list[i];
        switch (s->t) {
            case square: DrawSquare(s); break;
            case circle: DrawCircle(s); break;
        }
    }
}
```

Problems

- What if later we want to add Triangle shapes?
- Need to change the switch in DrawAll every time we add a new shape
- The switch statements are all over the code (not only in DrawAll)

Object-Oriented Solution in Java

```java
abstract class Shape {
    public abstract void draw();
}

class Circle extends Shape {...}
class Square extends Shape {...}
void drawAll(Shape[] list) {
    for (int i = 0; i < list.length; i++) {
        Shape s = list[i];
        s.draw();
    }
}
```

A new shape requires only creating a new subclass of Shape: no changes in DrawAll
Another Example: POS System

- External tax calculator
  - Invoked across the network
    - e.g. through TPC, SOAP, Java RMI, etc.
- Want to be able to plug different tax calculators from different vendors
  - Adaptability is specified in the requirements

Problem

- Different calculators have different interfaces
  - CalcX listens to raw bytes coming on some port
  - CalcY uses a higher-level mechanisms
    - e.g. remote procedure call, remote method invocation, web services, etc.
- Cannot anticipate the interfaces we will encounter in the future

Indirection and Abstraction

- Different interfaces: can create **adapter classes** running on the same machine as the POS system
  - Will deal with component-specific issues
  - Provide a level of **indirection**
- All adapter classes will implement a common interface that will be used by the POS system
  - Provides an **abstraction** of an adapter

Example

```java
ITaxCalculatorAdapter
getTaxes(Sale): TaxList

TaxMasterAdapter
getTaxes(Sale): TaxList

TurboTaxAdapter
getTaxes(Sale): TaxList

JoesTaxBonanzaAdapter
getTaxes(Sale): TaxList
```

The Impact of Change

- How do we replace the current calculator with a new one?
- Need to write a new adapter
  - New code: no way around it
- Only minor change in existing code
  - "new XAdapter() -> "new YAdapter()"
  - Or use the Factory design pattern

Polymorphism

- Abstraction through **subclasses**
- Based on polymorphism: the abstraction can take **many forms**
  - Many possible subclasses
  - Many possible overriding methods
- Polymorphism is a core feature of object-oriented programming languages
**Indirection**

- Very popular mechanism for protection from variations
- Simple idea: use an intermediary
- "Most problems in computer science can be solved by another level of indirection"
- Tax calculator adapters: an example of indirection
  - Adapter pattern

**Summary**

- Protecting one part of the program from changes in another part
  - protected variations", "open-closed principle", "information hiding"
  - Common mechanisms to achieve it: encapsulation, abstraction, polymorphism, indirection
- Many more “ambitious” mechanisms
  - e.g. at run time, clients use a lookup service to find a server object (protects against location and implementation changes)

**Design Patterns**

- Design patterns have become very popular in the last decade or so
- Major source: GoF book (gang of four)
  - Gamma, Helm, Johnson, Vlissides
  - “Design Patterns: Elements of Reusable Object-Oriented Software”
    - On reserve at SEL
- Patterns describe well-known solutions to common design problems

**Example**

- Standard Java libraries: patterns are everywhere
- Example: classes Component, Container, ComponentPeer, LayoutManager, Toolkit
  - From the AWT in JDK 1.0
- These 5 classes exhibit several patterns
  - Bridge, Singleton, AbstractFactory, Composite, Strategy

**Adapter Pattern**

- Problem: incompatible interfaces
- Solution: create a wrapper that maps one interface to another
- Example:
  - Client written against some interface
  - Server with the right functionality but with the wrong interface
**Example**

- Option 1: Change the client
- Option 2: Change ZServer
- Option 3: Create an adapter

**Sample Java Code**

```java
abstract class AbstractServer { abstract void foo(); }
class ZAdapter extends AbstractServer {
    private ZServer z;
    public ZAdapter() { z = new ZServer(); }
    public void foo() { z.bar(5000); }
}

Sample Java Code

```java
abstract class AbstractServer { abstract void foo(); }
class ZAdapter extends AbstractServer {
    private ZServer z;
    public ZAdapter(int perf) {
        if (perf > 10)  z = new BestZServer();
        else if (perf > 3) z = new BetterZServer();
        else z = new ZServer();
    }
    public void foo() { z.bar(5000); }
}
```

**Another Example (GoF)**

- Complicated, so let's reuse existing code
- Problem: mismatched interfaces
- Solution: create a TextShape adapter

**Adding TextShape**

```java
FreeText
    origin:Point
    width, height:double
    getOrigin():Point
    getWidth():double
    getHeight():double
    isEmpty():boolean
```
**Sample Java Code**

```java
class TextShape implements Shape {
    private FreeText t;
    public TextShape() { t = new FreeText(); }
    public boolean isEmpty() { return t.isEmpty(); }
    public Box boundingBox() {
        int x1 = toInt(t.getOrigin().getX());
        int y1 = toInt(t.getOrigin().getY());
        int x2 = toInt(x1 + t.getWidth());
        int y2 = toInt(y2 + t.getHeight());
        return new Box(x1,y1,x2,y2); }
    private int toInt(double) { … } }
```

**Pluggable Adapters**

- Preparation for future adaptation
- Define a narrow interface
- Future users of our code will write adapters that implement the interface

**Example: Display of Trees**

```
TreeDisplay
    TreeAccess
        getChildren(Node)
        getLabel(Node)
        getRoot()

FileSystem
    DirectoryAccess
    ClassHierarchyAccess
    OrganizationalChartAccess
    CompanyModel
```

**Bridge Pattern**

- GoF: “Decouple an abstraction from its implementation so that the two can vary independently”
- Key issue: dimensions of variability
- Single dimension: polymorphism based on inheritance or interfaces
  - e.g. different kinds of shapes
- What if there are several dimensions?

**Example**

- Program that draws rectangles
  - Two drawing classes D1 and D2
    - Each rectangle uses only one of the two
    - When a rectangle is created, we are told which drawing class it will use
  - D1 provides
    - `draw_a_line(x1,y1,x2,y2)` - two vertices
    - `draw_a_circle(x,y,r)` - center and radius
  - D2: `drawLn(x1,x2,y1,y2), drawCr(x,y,r)`

**Possible Design**

```
Client
    +draw_a_line: void
    +draw_a_circle: void
    +drawLn: void
    +drawCr: void

Rectangle
    +draw_a_line: double
    +draw_a_circle: double
    +drawLn: double
    +drawCr: double
    +drawLine: void
    +drawCircle: void

V1Rectangle
    +draw_a_line: void
    +draw_a_circle: void
    +drawLine: void
    +drawCircle: void

V2Rectangle
    +draw_a_line: void
    +draw_a_circle: void
    +drawLine: void
    +drawCircle: void
```
**Rectangle Code**

```java
abstract class Rectangle {
    public void draw() {
        drawLine(p1x, p1y, p2x, p1y);
        drawLine(p1x, p1y, p1x, p2y);
        drawLine(p2x, p2y, p2x, p1y);
        drawLine(p2x, p2y, p1x, p2y);
    }

    protected abstract void drawLine(double x1, double y1, double x2, double y2);

    private double p1x, p1y, p2x, p2y;

    // constructor not shown
}
```

**Subclasses**

```java
class V1Rectangle extends Rectangle {
    public void drawLine(double x1, double y1, double x2, double y2) {
        d1.drawLine(x1, y1, x2, y2);
    }

    public V1Rectangle(double x1, double y1, double x2, double y1, D1 d) {
        super(x1, y1, x2, y2); d1 = d;
    }

    private D1 d1;
}
```

```java
class V2Rectangle extends Rectangle {
    …
}
```

**Change in Requirements**

- The boss wants circles
- Client code should not make a distinction between circles and rectangles w.r.t. drawing
- Solution: superclass Shape, subclasses Rectangle and Circle
- All existing client code is rewritten to use Shape

**Problems**

- Too many classes
  - What if there is another drawing class? And two other kinds of shapes?
    - Total of 3 x 4 = 12 classes
  - Many classes are coupled to D1 and D2
    - What if the interface of D1 changed?

**Possible Alternative Solution**

- Solution: concentration on polymorphism and abstraction at the interface level
Deeper Problem

- Overuse of inheritance
- Two separate dimensions of variability
  - Kinds of shapes
  - Implementations for aspects of the display
- Common mistake: using inheritance when there are better solutions
- Solution: consider object composition instead of inheritance

Bridge Pattern

- Consider the two dimensions separately
- Connect ("bridge") them w/ composition
- Kinds of shapes
  - Shape is an abstract concept
    - Abstract class
  - Subclasses Rectangle and Circle
    - Each shape is responsible for drawing itself
      - Abstract method draw in Shape
      - Normal methods draw in the subclasses

Dimension 1

- Kinds of shapes
  - Abstract class Shape
  - Subclasses Rectangle and Circle
    - Each shape is responsible for drawing itself
      - Abstract method draw in Shape
      - Normal methods draw in the subclasses
- Association with the appropriate D1/D2

Dimension 2

- Kinds of drawing
  - Abstract class Drawing
  - Subclasses V1Drawing and V2Drawing
    - Each drawing is responsible for knowing how to draw lines and circles
      - Abstract methods drawLine and drawCircle in Drawing
      - Normal methods in the subclasses
    - Association with the appropriate D1/D2

The Bridge
**GoF Formulation**

- Decouple an abstraction from its implementation so that the two can vary independently
- `operation` calls `operationImpl`

**Why Use It?**

- Separate an implementation aspect into a separate hierarchy
  - Independent variations in both dimensions
  - Avoid explosion in # of classes
- Choosing and changing the implementor
  - e.g. choose at Rectangle creation time
  - Can be changed during the lifetime of a rectangle
    - add method `setDrawing(Drawing)` to `Shape`
- An implementor object can be shared

**Some Options for Implementor Creation**

- How, when, and where is the Implementor “hooked up” to Abstraction?
- **Option 1**: Constructors of Abstraction decide, based on input parameters
  - e.g. container: if size < 4 use linked list, otherwise use a hash table
- **Option 2**: Default implementor, plus changes based on run-time usage
  - If a container grows too much -> switch
- **Option 3**: Factory pattern (more later)

**Facade Pattern**

- A large subsystem, many classes
- Simplified view for the clients
  - High-level interface that is easier to use
  - e.g. we have a 3D drawing library, but we only want 2D, for a subset of the functionality
- Typical reasons
  - Use only a subset of the capabilities
  - Use in a particular "specialized" way
  - Reduce coupling with client code

**Example**

**Using a Facade**
Observations
- Possibly does some work, but mostly uses the existing classes
- Reduces access to underlying classes
  - If necessary, allows complete hiding: e.g. to reduce coupling, or to track all accesses
- Makes programmer's life easier
- Facade vs. Adapter
  - What are the differences?

Facade vs. Adapter

Yet Another Example

Summary
- Structural patterns
  - How are classes and objects composed to form larger structures?
  - Adapter: two incompatible interfaces
  - Bridge: independence of an abstraction from some implementation aspect
  - Facade: simpler interface
- Other structural patterns
  - Composite, Decorator, Flyweight, Proxy (GoF)

More Patterns (GoF)
- Creational patterns: how are objects created?
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton
- Behavioral patterns: how do objects interact?
  - Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor

Factory Method
- Isolate the creation of certain objects into a separate method
  - The method "manufactures" the objects
- Declared return type: a supertype of the created object
  - The caller does not know the exact type of the new object
- Several aspects of the pattern

abstract class NumberFormat { //from java.text // all of these are static methods
  NumberFormat getInstance() {…}
  NumberFormat getInstance(Locale l) {…}
  NumberFormat getPercentInstance() {…}
  NumberFormat getPercentInstance(Locale l) {…}
  NumberFormat getCurrencyInstance() {…}
  NumberFormat getCurrencyInstance(Locale l) {…}
  …}
More Details

class DecimalFormat extends NumberFormat {

in NumberFormat:
static NumberFormat getInstance(Locale l) {
  ... x = new DecimalFormat(...); ...; return x;
}

client code:
NumberFormat nf = NumberFormat.getInstance(Locale.FRENCH);

Aspect 2: Creation by Subclasses

- Abstract factory method in a superclass:
different implementations in subclasses

body of newDocument

Application
<table>
<thead>
<tr>
<th>MyApplication</th>
</tr>
</thead>
</table>
Document
<table>
<thead>
<tr>
<th>MyDocument</th>
</tr>
</thead>
</table>
newDocument
<table>
<thead>
<tr>
<th>newDocument</th>
</tr>
</thead>
</table>
openDocument
<table>
<thead>
<tr>
<th>openDocument</th>
</tr>
</thead>
</table>
MyApplication
<table>
<thead>
<tr>
<th>MyApplication</th>
</tr>
</thead>
</table>
newDocument
<table>
<thead>
<tr>
<th>newDocument</th>
</tr>
</thead>
</table>
openDocument
<table>
<thead>
<tr>
<th>openDocument</th>
</tr>
</thead>
</table>

Creation by Subclasses

- The superclass does not know what is being instantiated; it just uses it
- Need subclasses of both Document and Application for client-specific behavior
- Sometimes: in the superclass, have a "default" factory implementation
- Subclasses can override it if they need to

Aspect 3: Parallel Class Hierarchies

- X in one hierarchy has to be "connected" to a particular Y in the other

Abstract Factory

- A generalization of the "factory" idea
- Put factory method(s) in a separate class
- Usually: several categories of objects
  - e.g. GUI elements: windows, scroll bars, etc.
- Separate factory method for each category
  - createWindow():Window
  - createScrollBar():ScrollBar
  - Window and ScrollBar are abstract classes

Factory Example

- Abstract factory + concrete factories
Observations

- Creation of entire families of objects can be controlled transparently
  
  ```java
  AbstractFactory f = new ConcreteFactory1();
  AbstractFactory f = new ConcreteFactory2();
  ```

- Separation of responsibilities
  - Factory: decides which objects are needed
  - Client: uses objects through their supertypes

- Sometimes: no abstract factory, just a single concrete factory

Bridges and Factories

- Factory method: e.g. static method `createDrawing()` in `Drawing`; called by `Shape`
- Concrete factory class `DrawingFactory` with method `createDrawing()`; called by `Shape`

Singleton Pattern

- Goal: ensure that there is only one instance of a given class `X`

- Step 1: make all of `X`’s constructors private

- Step 2: add in `X` a private static field
  - Usually called “instance”

- Step 3: add a public static method that returns the static field
  - Usually called “getInstance()”

Example

```java
class Logger {
    private Logger() { }
    private static Logger instance = null;
    public static Logger getInstance() {
        if (instance == null)
            instance = new Logger();
        return instance;
    }
    client code: Logger.getInstance().writeLog(...)
```

Observations

- On-demand creation
  - For performance reasons
  - Alternative: upfront creation, by initializing the static field

- Problems for multi-threaded programs

- Common reasons for using it
  - Unique resources: file system, window manager, printer spooler, factory, ...

Real-World Example [Grosso 2002]

- Illustration of polymorphism, indirection, `AbstractFactory`, and `Singleton`

```java
class Socket {
    public Socket(String host, int port) { ... }
    public InputStream getInputStream() { ... }
    public OutputStream getOutputStream() { ... }
}

s = new Socket("foo.com",80);
s.getOutputStream().write(...); // send info
```
Compression

- Suppose we wanted to compress all data
  - e.g., HTML is very verbose
  - We already have compression implemented: wrapper classes around "normal" streams

```java
class CmpOutputStream extends OutputStream {
    public CmpOutputStream(OutputStream s) {…} }
class CmpInputStream extends InputStream {
    public CmpInputStream(InputStream s) {…} }
```

Possible Solution

```java
CmpOutputStream cos =
    new CmpOutputStream(skt.getOutputStream());
    cos.write(…);  
CmpInputStream cis =
    new CmpInputStream(skt.getInputStream());  
cis.read(...);
```

- Are there any problems with this simple approach?

Better Solution

```java
class CmpSocket extends Socket {
    public CmpSocket(String host, int port)
    { super(host,port); }  
    private CmpInputStream cis = null;
    private CmpOutputStream cos = null;
    public getInputStream() {
        if (this.cis == null) {
            InputStream orig = super.getInputStream();
            this.cis = new CmpInputStream(orig);
        }  
        return this.cis;  
    // similarly for getOutputStream()  
}
```

Creating the Right Sockets: Factory

- If we want to change the type of socket
  - Still have to replace "new Socket(...)" with "new CmpSocket(...)" all over the code
  - How about a factory?

```java
abstract class SocketFactory {
    public abstract Socket createSocket(String host, int port);  
    public static SocketFactory getDefault() {
        if (instance == null) instance = new CmpFactory();
        return instance;  
    private static SocketFactory instance;  
}
```

Creating the Right Sockets: Factory

- The client code will do this:
  - SocketFactory factory = SocketFactory.getDefault();
  - Socket skt = factory.createSocket(...);

```java
class CmpFactory extends SocketFactory {
    public Socket createSocket(String host, int port) {
        return new CmpSocket(host,port);  
    }
```

GoF Patterns

- Structural patterns:
  - Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
- Creational patterns
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton
- Behavioral patterns
  - Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor
**Observer Pattern**

- **Goal**: when one object $X$ changes state, all its dependent objects $Y_i$ are notified and updated automatically.

- **Want reduced coupling**
  - Do not hard-code calls from $X$ to $Y_i$.
  - Want if we want to add a new $Y_i$?

- **Abstract class Observer, subclassed by observer classes**

- **$X$ has a list of all observer objects**

**GoF Formulation (Modified)**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td>attach(Observer)</td>
<td>update(Subject)</td>
</tr>
<tr>
<td>detach(Observer)</td>
<td></td>
</tr>
<tr>
<td>notify()</td>
<td></td>
</tr>
<tr>
<td>getXYZ()</td>
<td></td>
</tr>
</tbody>
</table>

**ConcreteSubject**

- getXYZ()
- doSomething()

**ConcreteObserver**

- update(Subject)

**Interactions**

- doSomething() changes the state of the concrete subject, and invokes notify().

- notify() goes through all attached observers and invokes update(this) on each one.
  - this call invokes the corresponding update method in the concrete observers.

- update(Subject s) calls s.getXYZ() to get details about the state change in order to update its own state.

**Possible Implementation in Java**

```java
class Subject {
    private HashSet oset = new HashSet();
    public attach(Observer o) { oset.add(o); }
    public detach(Observer o) { oset.remove(o); }
    public notify() {
        Iterator it = oset.iterator();
        while (it.hasNext()) {
            Observer o = (Observer) it.next();
            o.update(this);
        }
    }
    ...
}
```

**Spreadsheet Example**

- In a spreadsheet, we may have multiple GUI views of the same data.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>50</td>
<td>30</td>
</tr>
<tr>
<td>Y</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>Z</td>
<td>80</td>
<td>10</td>
</tr>
</tbody>
</table>

- a = 50% b = 30% c = 20%

**Why Use It?**

- Changes in object state in one object require state changes in other objects.
- Do not know in advance the dependent objects.
- To decouple the objects.
  - In anticipation of future changes.
  - For reuse: can reuse the subjects without the observers, and vice versa.
- The subject is simplified.
  - Only responsibility: broadcast to observers.
Example: standard package java.util

class Observable { // subject
    public void addObserver(Observer o) { …
    public void deleteObserver(Observer o) { …
    public void deleteObservers() { …
    public int countObservers() { …
    public void notifyObservers()
        { notifyObservers(null); }  
    public void notifyObservers(Object arg) { …
    }
    interface Observer {
        public void update(Observable o, Object arg); }

Variation

- The observer tracks only one subject
  - Attribute visibility from ConcreteObserver to ConcreteSubject
  - e.g. in some method in the observer:
    - this.subject = s
    - s.addObserver(this)
  - Instead of using update(Subject), just use update()
    - The observer already has access to the observed object, and "asks" it for state info

Iterator

- Goal: access the elements of an aggregate object without exposing its underlying representation
  - e.g. elements of a List, Set, Table, …
- Use separate "iterator" classes
  - Each iterator object corresponds to a particular traversal of the elements
- Used very often, especially in standard libraries (C++, Java, etc.)

General Form

Example: java.util.ArrayList

- ArrayList; subclass of AbstractList
  - add(Object, index), isEmpty(), …
  - Factory method: "Iterator iterator()"
    - Inherited from the superclass
- Interface Iterator
  - Methods hasNext() and next()
- iterator() in AbstractList returns an instance of an internal class that:
  - implements the Iterator interface
  - is specific for lists

Sample Client Code

ArrayList ar;
...
Iterator iter = ar.iterator();
while ( iter.hasNext() ) {
    Object element = iter.next();
    // do something with the list element
}
Observation

- An iterator is associated with a particular aggregate object
- There may be several active iterators for the same object at the same time
- Iterate over different aggregates
  - AbstractList has subclasses LinkedList, ArrayList, Vector, and Stack
- Are iterators allowed to change the aggregate object?
  - Sometimes yes, but should be done carefully

Example: Fail-Fast Iterators in Java

```java
List list = new ArrayList();
list.add("X"); list.add("Y"); list.add("Z");
Iterator iter = list.iterator();
while (iter.hasNext()) {
    Object o = iter.next();
    if (…) list.add("W"); // not OK: exception
}
list.add("V"); // OK, iteration has ended
```