


# 6.170 Lecture 21 Design Patterns



MIT EECS

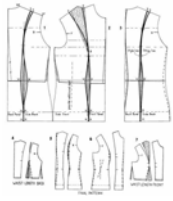
1

## What are Design Patterns?

**In Architecture - *Wikipedia***

The patterns serve as an aid to design cities and buildings.

A pattern records the design decisions taken by many builders in many places over many years in order to resolve a particular problem.



**In Software Engineering - *Wikipedia***

Design patterns are standard solutions to common problems in software design.

Instead of focusing on how individual components work, design patterns take a systematic approach, which focuses on the patterns of interaction.

Design patterns describe abstract systems of interaction between classes, objects, and communication flow.

2

## design patterns

Standard solutions to common programming problems

- Let you minimize coupling where it matters most

Distilled, debugged, documented over decades

- Lots of knowledge about trade-offs (see “Gang of Four”)
- Help you see a few steps ahead in the game of design, to foresee non-obvious problems and make wiser choices

We’ve been using some of them all term

- Some important ones are built in to the Java language, or supported in libraries
- Encapsulation, inheritance, exceptions, iterators, etc.

Today: a quick survey

3

## Example 1: Encapsulation (data hiding)

**Problem:** Exposed fields can be directly manipulated

- Violations of the representation invariant
- Dependencies prevent changing the implementation

**Solution:** Hide some components

- Permit only stylized access to the object

**Disadvantages:**

- Interface may not (efficiently) provide all desired operations
- Indirection may reduce performance

4

## Example 2: Subclassing (inheritance)

**Problem:** Repetition in implementations

- Similar abstractions have similar members (fields, methods)

**Solution:** Inherit default members from a superclass

- Select an implementation via run-time dispatching

**Disadvantages:**

- Code for a class is spread out, may reduce understandability
- Run-time dispatching introduces overhead

5

## Example 3: Iteration

**Problem:** To access all members of a collection, must perform a specialized traversal for each data structure

- Introduces undesirable dependencies
- Does not generalize to other collections

**Solution:**

- The implementation performs traversals, does bookkeeping
- Results are communicated to clients via a standard interface

**Disadvantages:**

- Iteration order is fixed by the implementation and not under the control of the client
- Modifying the collection while iterating can be problematic.

6

## Example 4: Exceptions

### Problem:

- Errors in one part of the code should be handled elsewhere.
- Code should not be cluttered with error-handling code.
- Return values should not be preempted by error codes

### Solution: Language support for exceptions

### Disadvantages:

- Code may still be cluttered.
- It may be hard to know where an exception will be handled.
- Use of exceptions for normal control flow may be confusing and inefficient.

7

## categories of design patterns

### Creational patterns

Minimize coupling introduced by object creation

### Structural patterns

Minimize coupling introduced when classes or objects are assembled to form larger structures

### Behavioral patterns

Minimize coupling introduced when classes or objects cooperate to perform distributed tasks

8

## categories of design patterns

### Creational patterns

Minimize coupling introduced by object creation

### Structural patterns

Minimize coupling introduced when classes or objects are assembled to form larger structures

### Behavioral patterns

Minimize coupling introduced when classes or objects cooperate to perform distributed tasks

9

## Example: Changing Implementations

### How do you support multiple implementations?

Use an interface

```
interface Matrix { ... }  
class SpaceMatrix implements Matrix { ... }  
class DenseMatrix implements Matrix { ... }  
Use Matrix all over...
```

But...

Still need to use a constructor of either `SpaceMatrix` or `DenseMatrix`.

Issues...

Changing from `SpaceMatrix` or `DenseMatrix` has to change all the constructors in the code → forgetting to change one is a big problem  
The user has to decide whether to use `SpaceMatrix` or `DenseMatrix`

10

## How about?

```
class MatrixFactory {  
    public static Matrix CreateMatrix() { ... }  
    ...  
}
```

### Advantages

- When switching the implementation, only need to change in one place
- Can decide what type of matrix to create
- For frequently used immutable objects, may not create a new object at all

11

## weaknesses of Java constructors

### Constructors in Java are inflexible

- Must always return a fresh new object, never re-use one
- Can't return a subtype of the class they belong to

### When you want flexibility in creation, use factories

#### Factory methods

- Hide decision about what type is instantiated, and whether the object is new or re-used

#### Factory classes

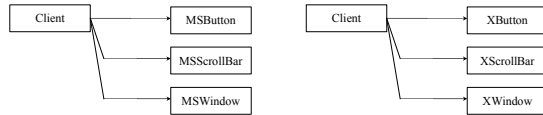
- Bundle factory methods for a whole family of types
- Methods can be overridden, factory objects interchanged

12

## factory class motivation

Sometimes we have a whole family of classes we would like to be able to interchange easily

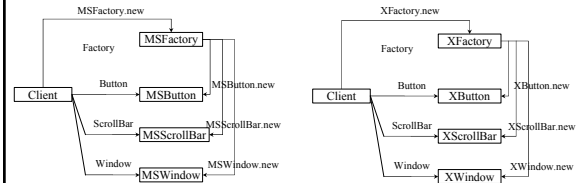
For example, sets of graphical widgets for different platforms or with different “look-and-feel”



13

## factory class example

Can create a class which bundles factory methods for each widget



14

## using a factory class

Now can replace code like this:

```
XWindow window = new XWindow(100,100,200,200);
window.add(new XButton("OK"));
window.add(new XScrollBar());
```

With code like this:

```
Factory factory = new XFactory();
Window window = factory.createWindow(100,100,200,200);
window.add(factory.createButton("OK"));
window.add(factory.createScrollBar());
```

Dependency on concrete classes of widgets reduced to exactly one point – easy to change, or make dynamic

15

## boolean in a box

Boolean's constructor :

```
public class Boolean {
    private final boolean value;
    // Construct a Boolean object
    public Boolean(boolean b) {
        value = b;
    }
}
```

16

## comment from Bloch

“For example, the Boolean type, which is a boxed primitive boolean, simply should not have had public constructors. ... There's really no great advantage to allow multiple trues or multiple falses, and I've seen programs that produce millions of trues and millions of falses, creating needless work for the garbage collector.

So, in the case of immutables, I think factory methods are great.”

*JavaWorld, January 4, 2004*

17

## constructor vs. factory in Boolean

Compare Boolean's constructor and valueOf method:

```
public class Boolean {
    private final boolean value;
    public static final Boolean FALSE = new Boolean(false);
    public static final Boolean TRUE = new Boolean(true);
    // Construct a Boolean object
    public Boolean(boolean b) {
        value = b;
    }
    // Convert primitive boolean to Boolean
    public static Boolean valueOf(boolean b) {
        return (b ? Boolean.TRUE : Boolean.FALSE);
    }
    // ...
}
```

18

## comment in constructor

```
public Boolean(boolean b)
```

Allocates a Boolean object representing the value argument.

**Note: It is rarely appropriate to use this constructor. Unless a new instance is required, the static factory valueOf(boolean) is generally a better choice. It is likely to yield significantly better space and time performance.**

19

## use of factory methods by DateFormat

DateFormat class encapsulates knowledge about how to format dates and times as text

Options: just date? just time? date+time? where in the world?

Instead of passing all options to constructor, uses factories.

Tidy, and the subtype created doesn't need to be specified.

```
DateFormat df1 = DateFormat.getDateInstance();
DateFormat df2 = DateFormat.getTimeInstance();
DateFormat df3 = DateFormat.getDateInstance(DateFormat.FULL,
                                             Locale.FRANCE);
```

```
Date today = new Date();
System.out.println(df1.format(today)); // "Mar 28, 2005"
System.out.println(df2.format(today)); // "10:15:00 AM"
System.out.println(df3.format(today)); // "lundi 28 mars 2005"
```

20

## other useful creational patterns

### Interning

Factory method which, if asked to create an immutable object indistinguishable from one created earlier, simply returns a reference to that one – huge space efficiency gain

### Flyweight pattern

Factor objects into mutable and immutable part  
Intern the immutable part, and try to remove the mutable part (may be possible to compute it from context)

### Singleton pattern

Only a single copy of the object is needed

### Prototype pattern

Can give objects a copy/clone method, and use instances as "prototypes" for objects we would like to create

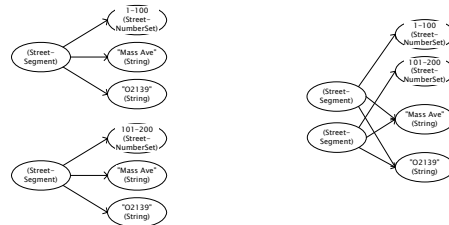
21

## Interning

Reuse existing objects instead of creating new ones

Permitted only for immutable objects

Example: StreetSegment



22

## Interning mechanism

Maintain a collection of all objects

If an object already appears, return that instead

```
HashMap<String,String> segnames = new HashMap<String,String>();
String canonicalName(String n) {
    if (segnames.containsKey(n)) {
        return segnames.get(n);
    } else {
        segnames.put(n, n);
        return n;
    }
}
```

Two approaches:

create the object, but perhaps discard it and return another  
check against the arguments before creating the new object

23

## Singleton

Only one object of the given type exists

```
class Bank {
    private static Bank theBank;

    // constructor
    private Bank() { ... }

    // factory method
    public static Bank getBank() {
        if (theBank == null) {
            theBank = new Bank();
        }
        return theBank;
    }
    ...
}
```

24

## categories of design patterns

### Creational patterns

Minimize coupling introduced by object creation

### Structural patterns

Minimize coupling introduced when classes or objects are assembled to form larger structures

### Behavioral patterns

Minimize coupling introduced when classes or objects cooperate to perform distributed tasks

25

## wrapper patterns

Wrappers modify other classes

May change an interface, extend behavior, restrict access, etc.

Often used to “tweak” classes designed by different groups so that they can fit together

Three general varieties: adapters, decorators, proxies

*client's view of wrapped class*

Pattern	Functionality	Interface
Adapter	≈ same	different
Decorator	different	≈ same
Proxy	≈ same	≈ same

26

## Adapter

Change an interface without changing functionality

rename a method

convert units

implement a method in terms of another

Example: angles passed in radians vs. degrees

27

## adapters

Suppose we're writing a drawing program and want to pull in classes from a library that relies on shapes having the same basic functionality, but expressed differently:

```
public interface ScaleableShape {
    void draw(Graphics g);
    void scale(float factor);
    Rectangle bounds();
    // ...
}

public interface SizeableShape {
    void draw(Graphics g);
    void setWidth(float width);
    void setHeight(float height);
    float getWidth();
    float getHeight();
    Point getTopLeft();
    Point getBottomRight();
    // ...
}
```

28

## adapter solution

An adapter can iron out the interface difference:

```
class ShapeAdaptor implements ScaleableShape {
    final SizeableShape sizeShape;
    public ShapeAdaptor(SizeableShape shape) {
        sizeShape = shape;
    }
    public void scale(float factor) {
        sizeShape.setWidth(sizeShape.getWidth()*factor);
        sizeShape.setHeight(sizeShape.getHeight()*factor);
    }
    public void draw(Graphics g) {
        sizeShape.draw(g);
    }
    // ...
}
```

29

## another example

NASA expects a subcontractor to build this:

```
public class ImperialSensor {
    // returns: distance traveled in miles
    public double getDistance();
    // returns: speed in miles per hour
    public double getSpeed();
}
```

But instead the subcontractor builds this:

```
public class MetricSensor {
    // returns: distance traveled in meters
    public double getDistance();
    // returns: speed in meters per second
    public double getSpeed();
}
```

30

## adapter solution

We can try to fix this interface mismatch with an adapter:

```
public class ImperialSensor extends MetricSensor {  
    // returns: distance traveled in miles  
    public double getDistance() {  
        return super.getDistance() * MILES_PER_METER;  
    }  
    // returns: speed in miles per hour  
    public double getSpeed() {  
        return super.getSpeed() * MILES_PER_METER * HOURS_PER_SEC;  
    }  
}
```

Making adapters using extension is generally fine, but in this case is dangerous (ImperialSensor is not a true subtype of MetricSensor; disaster if passed to a module expecting one)

31

## adapter solution

Better:

```
public class ImperialSensor {  
    private final MetricSensor ms;  
    public ImperialSensor() {  
        ms = new MetricSensor();  
    }  
    // returns: distance traveled in miles  
    public double getDistance() {  
        return ms.getDistance() * MILES_PER_METER;  
    }  
    // returns: speed in miles per hour  
    public double getSpeed() {  
        return ms.getSpeed() * MILES_PER_METER * HOURS_PER_SEC;  
    }  
}
```

32

## Decorator

Add functionality without changing the interface  
Add to existing methods to do something additional  
(while still preserving the previous specification)  
Not all subclassing is decoration

33

## Decorator example: Bordered windows

```
interface Window {  
    // rectangle bounding the window  
    Rectangle bounds();  
    // draw this on the specified screen  
    void draw(Screen s);  
    ...  
}  
  
class WindowImpl implements Window {  
    ...  
}
```

34

## Bordered windows (2)

Via subclassing:

```
class BorderedWindow1 extends WindowImpl {  
    void draw(Screen s) {  
        super.draw(s);  
        bounds().draw(s);  
    }  
}
```

Via delegation:

```
class BorderedWindow2 implements Window {  
    Window innerWindow;  
    BorderedWindow2(Window innerWindow) {  
        this.innerWindow = innerWindow;  
    }  
    void draw(Screen s) {  
        innerWindow.draw(s);  
        innerWindow.bounds().draw(s);  
    }  
}
```

35

## proxies

Suppose as a word processor is loading a document,  
embedded images are created and loaded:

```
public class EmbeddedImage implements Drawable {  
    public EmbeddedImage(String filename) {  
        // loads image from file ...  
    }  
    public void draw(Graphics g) {  
        // renders image ...  
    }  
    // ...  
}
```

Problem: slow! All images have to load, even for pages  
user isn't looking at.

36

## proxy example

Drop in replacement that delays loading until viewing:

```
class EmbeddedImageProxy implements Drawable {
    private EmbeddedImage image = null;
    private final String filename;
    public EmbeddedImageProxy(String filename) {
        this.filename = filename;
    }
    public void draw(Graphics g) {
        if (image==null) {
            image = new EmbeddedImage(filename);
        }
        image.draw(g);
    }
    // ...
}
```

37

## uses of proxies

Proxy has same interface&functionality as wrapped object

But access is displaced or indirect or stylized in some way

- Access remote objects across the network through local proxies with the same interface
- Take care of locking in multi-threaded applications
- Delay object creation speculatively

Isolates client code from these complexities

For EmbeddedImageProxy example, could easily and transparently improve performance by adding a background thread to load images while user is reading

Specifications need to be careful to allow proxies

e.g. what is specified behavior if an image doesn't exist?

38

## Composite Example: Collecting Shapes...

```
interface Shape { ... }
class Rectangle implements Shape { ... }
class Triangle implements Shape { ... }
class Polygon implements Shape { ... }
```

How about groups of shapes?

```
class ShapeGroup {
    List<Shape> shapes;
    ...
}
```

Now I want Groups of Groups.

39

## composite pattern

For nested structures, it helps to have a uniform interface to the “atomic” parts and the “composite” parts

Example: The Shape class

Groups of Group require you add code to ShapeGroups so that it can accept other ShapeGroups

Messy! – simpler if ShapeGroup is a Shape, so you get nesting for free.

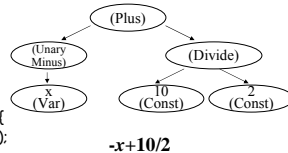
Particularly important when there are many forms of composition – e.g. representing math expressions

40

## composite

Representing expressions using composites:

```
class Plus implements Expression {
    private Expression left;
    private Expression right;
    public double eval (Context context) {
        return left.eval (context) + right.eval (context);
    }
    // ...
}
class Var implements Expression {
    private String name;
    public double eval (Context context) {
        return context.getVarValue (name);
    }
    // ...
}
```



41

## categories of design patterns

Creational patterns

Minimize coupling introduced by object creation

Structural patterns

Minimize coupling introduced when classes or objects are assembled to form larger structures

Behavioral patterns

Minimize coupling introduced when classes or objects cooperate to perform distributed tasks

42

## visitor pattern motivation

Each type of operation on a composite require a method in every class of that composite

Fine if the set of operations is stable, and the set of classes is growing/changing

But what if the set of classes is stable, and the set of operations is flexible?

E.g. our Expression classes may be fixed, but we want to compute many different properties of expressions

Annoying to have to keep modifying all the classes

Visitor pattern allows us to group methods by operation rather than by composite class

Trivial in some languages, non-trivial in Java

43

## visitor pattern idea

Instead of having methods for each computation, we instead delegate to a “Visitor” that gets passed in

```
class Plus implements Expression {
    private Expression left;
    private Expression right;
    public void accept (Visitor visitor) {
        visitor.visitPlus (this);
    }
}
class Var implements Expression {
    private String name;
    public void accept (Visitor visitor) {
        visitor.visitVar (this);
    }
}
```

44

## replacement for eval method

Here's a Visitor to evaluate Expressions:

```
class Evaluator implements Visitor {
    Context context; // maps variables to values
    double result; // value of last subexpr evaluated
    void visitPlus (Plus e) {
        e.left().accept (this); // evaluate left subexpr
        double left = result; // hang onto its result
        e.right().accept (this); // evaluate right subexpr
        double right = result;
        result = left + right; // combine the two results
    }
    void visitVar (Var e) {
        result = context.getVarValue(e.name());
    }
    // ...
}
```

45

## a new computation

Here's a Visitor to collect the set of variables used in an Expression. For these Visitors, I've assumed lots of observer methods.

```
class FindVariables implements Visitor {
    private Set vars = new HashSet ();
    public void visitPlus (Plus e) {
        e.left().accept (this);
        e.right().accept (this);
    }
    public void visitVar (Var e) {
        vars.add(e.name());
    }
    // ...
}
```

46

## model/view/controller

MVC pattern is a separation of concerns between:

The model: manages application data

- text in a document, values in a spreadsheet

Views: manage how parts of the model are presented

- draft-view vs print-preview, tables vs graphs

Controllers: manage how input from the user changes the model

Makes heavy use of Observer pattern (behavioral pattern covered in lecture 16), Composite pattern, and others

Variant: Model/View

Integrates views and controllers – hard to separate

47

## when (not) to use design patterns

Delay – get something basic working first, then improve it once you understand it

Design patterns can increase or decrease understandability

Add indirection, increase code size

Improve modularity, separate concerns, ease description

If your design has a problem, consider design patterns that address that problem

Canonical reference: the “Gang of Four” book

Good Java reference: the Bloch book

48