courses.cs.washington.edu/courses/cse403/16wi/

Design Patterns

Emina Torlak
cmina@cs.washington.edu
Outline

• Overview of design patterns
• Creational patterns
• Structural patterns
• Behavioral patterns
overview of design patterns
What is a design pattern?
What is a design pattern?

- A **standard** solution to a common programming problem
  - a design or implementation structure that achieves a particular purpose
  - a high-level programming idiom
What is a design pattern?

- A **standard** solution to a common programming problem
  - a design or implementation structure that achieves a particular purpose
  - a high-level programming idiom
- A technique for making code **more flexible** or **efficient**
  - reduce coupling among program components
  - reduce memory overhead
What is a design pattern?

• A **standard** solution to a common programming problem
  • a design or implementation structure that achieves a particular purpose
  • a high-level programming idiom

• A technique for making code **more flexible** or **efficient**
  • reduce coupling among program components
  • reduce memory overhead

• **Shorthand** for describing program design
  • a description of connections among program components
  • the shape of a heap snapshot or object model
Why should you care?

• You could come up with these solutions on your own …
• But you shouldn't have to!
• A design pattern is a known solution to a known problem.
Types of design patterns

• Creational patterns
  • how objects are instantiated

• Structural patterns
  • how objects / classes can be combined

• Behavioral patterns
  • how objects communicate

• Concurrency patterns
  • how computations are parallelized / distributed
When (not) to use design patterns
When (not) to use design patterns

• Rule 1: delay
  • Understand the problem & solution first, then improve it
When (not) to use design patterns

• Rule 1: delay
  • Understand the problem & solution first, then improve it

• Design patterns can increase or decrease understandability of code
  • Add indirection, increase code size
  • Improve modularity, separate concerns, ease description
When (not) to use design patterns

• Rule 1: delay
  • Understand the problem & solution first, then improve it

• Design patterns can increase or decrease understandability of code
  • Add indirection, increase code size
  • Improve modularity, separate concerns, ease description

• If your design or implementation has a problem, consider design patterns that address that problem

• References:
  • Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, 1995.
creational patterns
Kinds of creational patterns

- Factory (method)
- Abstract factory
- Builder
- Prototype
- Flyweight
- Singleton

Creational patterns address inflexibility of constructors in Java:

1. Can't return a subtype of the class they belong to
2. Always return a fresh new object, never re-use one
Factory patterns (problem)

```
interface Matrix {
    ...
}

class SparseMatrix implements Matrix {
    ...
}

class DenseMatrix implements Matrix {
    ...
}
```

• Clients use the supertype (Matrix)
  • But still need to use a SparseMatrix or DenseMatrix constructor
  • Must decide concrete implementation somewhere

• Don’t want to change code to use a different constructor
Factory method pattern (one solution)

class MatrixFactory {
   public static Matrix createMatrix() {
      return new SparseMatrix();
   }
}

• Clients call createMatrix instead of a particular constructor

• Advantages:
  • To switch the implementation, change only one place
  • createMatrix can do arbitrary computations to decide what kind of matrix to make

• Frequently used in frameworks (e.g., Java swing)
  • BorderFactory.createRaisedBevelBorder()
Abstract factory pattern (another solution)

A factory class that can be subclassed (to make new kinds of factories) and that has an overridable method to create its objects.
shape

structural patterns
Kinds of structural patterns

- Composite
- Decorator
- Adapter
- Proxy
- ...

Structural patterns enable client code to
1. modify the interface
2. extend behavior
3. restrict access
4. unify access
Composite pattern

A client can manipulate the whole or any part uniformly.

Component

Leaf

Composite

+ operation()  
+ add()  
+ remove()  
+ getChild()
Composite pattern example: Java GUI

Container north = new JPanel(new FlowLayout());
north.add(new JButton("Button 1"));
north.add(new JButton("Button 2"));

Container south = new JPanel(new BorderLayout());
south.add(new JLabel("Southwest"), BorderLayout.WEST);
south.add(new JLabel("Southeast"), BorderLayout.EAST);

Container overall = new JPanel(new BorderLayout());
overall.add(north, BorderLayout.NORTH);
overall.add(new JButton("Center Button"), BorderLayout.CENTER);
overall.add(south, BorderLayout.SOUTH);
frame.add(overall);
Decorator pattern

A decorator is a wrapper object that modifies behavior of, or adds features to, another object.
**Decorator pattern example: Java IO**

- InputStream class has only public int read() method to read one letter at a time.
- Decorators such as BufferedReader add functionality to read the stream more easily.

```java
// InputStreamReader/BufferedReader decorate InputStream
InputStream in = new FileInputStream("hardcode.txt");
InputStreamReader isr = new InputStreamReader(in);
BufferedReader br = new BufferedReader(isr);

// With a BufferedReader decorator, read an entire line from the file in one call
// (InputStream only provides public int read() )
String wholeLine = br.readLine();
```
behave

behavioral patterns
Kinds of behavioral patterns

- Null object
- Template method
- Iterator
- Strategy
- ...

Behavioral patterns identify and capture common patterns of communication between objects.
Null object pattern

A client can treat the absence of an object transparently.

Client

_abstractObject_

+request()

RealObject

+request()

NullObject

+request()

do nothing
Null object pattern example: empty list

List<Object> search(String value) {
    if ("".equals(value))
        return Collections.emptyList(); // null object (empty list)
    else
        return ...;
}

if (search(userInput).isEmpty())  // no NullPointerException
    ...;
else
    ...
Template method pattern

Subclasses can redefine certain steps of an algorithm without changing the algorithm's structure.
Template method example: games

abstract class Game {

    protected int playersCount;
    abstract void initializeGame();
    abstract void makePlay(int player);
    abstract boolean endOfGame();
    abstract void printWinner();

    // template method
    public final void playOneGame(int playersCount) {
        this.playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j);
            j = (j + 1) % playersCount;
        }
        printWinner();
    }
}

class Monopoly extends Game { ... }
class Chess extends Game { ... }
Summary

• A design pattern is a known solution to a known problem.
  • Creational, structural, behavioral
• If your design or implementation has a problem, then (and only then) consider design patterns that address that problem.