Section 9: Design Patterns

Slides by Alex Mariakakis

with material from David Mailhot, Hal Perkins, Mike Ernst
What Is A Design Pattern

• A standard solution to a common programming problem
• A technique for making code more flexible
• Shorthand for describing program design and how program components are connected
Creational Patterns

• Problem: Constructors in Java are not flexible
  o Always return a fresh new object, never reuse one
  o Can’t return a subtype of the class they belong to

• Solution: Creational patterns!
  o Sharing
    • Singleton
    • Interning
    • Flyweight
  o Factories
    • Factory method
    • Factory object
  o Builder
Creational Patterns: Sharing

- The old way: Java constructors always create a new object
- **Singleton**: only one object exists at runtime
- **Interning**: only one object with a particular (abstract) value exists at runtime
- **Flyweight**: separate intrinsic and extrinsic state, represents them separately, and interns the intrinsic state
Singleton

• For a class where only one object of that class can ever exist

• “Ensure a class has only one instance, and provide a global point of access to it.” -- GoF, Design Patterns

• Two possible implementations
  o Eager initialization: creates the instance when the class is loaded to guarantee availability
  o Lazy initialization: only creates the instance once it’s needed to avoid unnecessary creation
Singleton

• Eager initialization

    public class Bank {
        private static Bank INSTANCE = new Bank();

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

        // factory method
        public static Bank getInstance() {
            return INSTANCE;
        }
    }

    Bank b = new Bank();
    Bank b = Bank.getInstance();
Singleton

• Lazy initialization

```java
public class Bank {
    private static Bank INSTANCE;

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

    // factory method
    public static Bank getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new Bank();
        }
        return INSTANCE;
    }
}
```

Bank b = new Bank();
Bank b = Bank.getInstance();
Singleton

• Would you prefer eager or lazy instantiation for an HTTPRequest class?
  o handles authentication
  o definitely needed for any HTTP transaction

• Would you prefer eager or lazy instantiation for a Comparator class?
  o compares objects
  o may or may not be used at runtime
public class HttpRequest {
    private static class HttpRequestHolder {
        public static final HttpRequest INSTANCE = new HttpRequest();
    }

    /* Singleton – Don’t instantiate */
    private HttpRequest() { ... }

    public static HttpRequest getInstance() {
        return HttpRequestHolder.INSTANCE;
    }
}
Singleton

public class LengthComparator implements Comparator<String> {
    private int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }

    /* Singleton – Don’t instantiate */
    private LengthComparator() { ... }
    private static LengthComparator comp = null;

    public static LengthComparator getInstance() {
        if (comp == null) {
            comp = new LengthComparator();
        }
        return comp;
    }
}

Singleton

public class LengthComparator implements Comparator<String> {
    private int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }

    /* Singleton – Don’t instantiate */
    private LengthComparator() { ... }
    private static LengthComparator comp = null;

    public static LengthComparator getInstance() {
        if (comp == null) {
            comp = new LengthComparator();
        }
        return comp;
    }
}
Interning

• Similar to Singleton, except instead of just having one object per class, there’s one object per **abstract value** of the class

• Saves memory by compacting multiple copies
public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() { return x; }
    public int getY() { return y; }

    @Override
    public String toString() {
        return "(" + x + "," + y + ")";
    }
}
Interning

```java
public class Point {
    private static Map<String, Point> instances =
        new WeakHashMap<String, Point>();

    public static Point getInstance(int x, int y) {
        String key = x + "," + y;
        if (!instances.containsKey(key))
            instances.put(key, new Point(x, y));
        return instances.get(key);
    }

    private final int x, y; // immutable
    private Point(int x, int y) {...}
}

Requires the class being interned to be immutable. Why?
Interning

• What if Points were represented in polar coordinates?

• What further checks are necessary to make sure these kinds of Points are interned correctly?
public class Point {
    private static Map<String, Point> instances =
        new WeakHashMap<String, Point>();

    public static Point getInstance(double r, double theta) {
        double normalizedTheta = normalize(theta);
        String key = r + "," + normalizedTheta;
        if (!instances.containsKey(key))
            instances.put(key,
                new Point(r, normalizedTheta));
        return instances.get(key);
    }

    private final double r, theta; // immutable
    private Point(double r, double theta) {...}
}

Why do we need to normalize?
Factories

• Suppose we want a constructor for Set that takes a list as a parameter, and produces a TreeSet if the list is sorted, and a HashSet otherwise.
• Is this possible?
Factories

- Factories solve the problem that Java constructors cannot return a subtype of the class they belong to.

- Two options:
  - Factory method
    - Helper method creates and returns objects.
    - Method defines the interface for creating an object, but defers instantiation to subclasses.
  - Factory object
    - Abstract superclass defines what can be customized.
    - Concrete subclass does the customization, returns appropriate subclass.
public static Set produceSet(List list) {
    if (isSorted(list)) {
        return new TreeSet(list);
    } else {
        return new HashSet(list);
    }
}
Factory Object

interface SetFactory {
    Set getSet();
}

class HashSetFactory implements SetFactory {
    public Set getSet() {
        return new HashSet();
    }
}

Builder

- The class has an inner class `Builder` and is created using the `Builder` instead of the constructor
- The `Builder` takes optional parameters via setter methods (e.g., `setX()`, `setY()`, etc.)
- When the client is done supplying parameters, she calls `build()` on the `Builder`, finalizing the builder and returning an instance of the object desired
- Useful when you have many constructor parameters
  - It is hard to remember which order they should all go in
- Easily allows for optional parameters
  - If you have n optional parameters, you need $2^n$ constructors, but only one builder
public class NutritionFacts {
   // required
   private final int servingSize, servings;
   // optional
   private final int calories, fat, sodium;

   public NutritionFacts(int servingSize, int servings) {
      this(servingSize, servings, 0);
   }
   public NutritionFacts(int servingSize, int servings, int calories) {
      this(servingSize, servings, calories, 0);
   }
   public NutritionFacts(int servingSize, int servings, int calories, int fat) {
      this(servingSize, servings, calories, fat, 0);
   }
   public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium) {
      this.servingSize  = servingSize;
      this.servings     = servings;
      this.calories     = calories;
      this.fat          = fat;
      this.sodium       = sodium;
   }
}
public class NutritionFacts {
    private final int servingSize, servings, calories, fat, sodium;

    public static class Builder {
        // required
        private int servingSize, servings;
        // optional, initialized to default values
        private int calories = 0;
        private int fat = 0;
        private int sodium = 0;

        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings = servings;
        }

        public Builder calories(int val) { calories = val; return this; }
        public Builder fat(int val) { fat = val; return this; }
        public Builder sodium(int val) { sodium = val; return this; }

        public NutritionFacts build() { return new NutritionFacts(this); }
    }

    public NutritionFacts(Builder builder) {
        this.servingSize = builder.servingSize;
        this.servings = builder(servings);
        this.calories = builder.calories;
        this.fat = builder.fat;
        this.sodium = builder.sodium;
    }
}
Structural Patterns

- Problem: Sometimes difficult to realize relationships between entities
  - Important for code readability
- Solution: Structural patterns!
  - We’re just going to talk about wrappers, which translate between incompatible interfaces

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Functionality</th>
<th>Interface</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>same</td>
<td>different</td>
<td>modify the interface</td>
</tr>
<tr>
<td>Decorator</td>
<td>different</td>
<td>same</td>
<td>extend behavior</td>
</tr>
<tr>
<td>Proxy</td>
<td>same</td>
<td>same</td>
<td>restrict access</td>
</tr>
</tbody>
</table>
Adapter

• Changes an interface without changing functionality
  o Rename a method
  o Convert units

• Examples:
  o Angles passed in using radians vs. degrees
  o Bytes vs. strings
Decorator

- Adds functionality without changing the interface
  - Add caching
- Adds to existing methods to do something additional while still preserving the previous spec
  - Add logging
- Decorators can remove functionality without changing the interface
  - UnmodifiableList with add() and put()
Proxy

- Wraps the class while maintaining the same interface and functionality
- Integer vs. int, Boolean vs. boolean
- Controls access to other objects
  - Communication: manage network details when using a remote object
  - Security: permit access only if proper credentials
  - Creation: object might not yet exist because creation is expensive
Activity

- What pattern would you use to...
  - add a scroll bar to an existing window object in Swing
  - We have an existing object that controls a communications channel. We would like to provide the same interface to clients but transmit and receive encrypted data over the existing channel.
Activity

- Adapter, Builder, Decorator, Factory, Flyweight, Intern, Model-View-Controller (MVC), Proxy, Singleton, Visitor, Wrapper

- What pattern would you use to...
  - add a scroll bar to an existing window object in Swing
    - Decorator
  - We have an existing object that controls a communications channel. We would like to provide the same interface to clients but transmit and receive encrypted data over the existing channel.
    - Proxy