Section 9:
Design Patterns

Slides adapted from Alex Mariakakis, with material from David Mailhot, Hal Perkins, Mike Ernst

Agenda

- What are design patterns?
- Creational patterns review
- Structural patterns preview

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
  - Always return a fresh new object, never reuse one
  - Can’t return a subtype of the class they belong to
- Solution: Creational patterns!
  - Sharing
    - Singleton
    - Interning
    - Flyweight
  - Factories
    - Factory method
    - Factory object
  - Builder (new!)

Creational Patterns: Sharing

- The old way: Java constructors always return a new object
- **Singleton:** only one object exists at runtime
  - Factory method returns the same object every time
- **Interning:** only one object with a particular (abstract) value exists at runtime
  - Factory method returns an existing object, not a new one
- **Flyweight:** separate intrinsic and extrinsic state, represents them separately, and interns the intrinsic state
  - Implicit representation uses no space
  - Not as common/important

Creational Patterns: Singleton

- For a class where only one object of that class can ever exist
- Two possible implementations
  - Eager instantiation: creates the instance when the class is loaded to guarantee availability
  - Lazy instantiation: only creates the instance once it’s needed to avoid unnecessary creation

Creational Patterns: Singleton

- Eager instantiation

```
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();
```

Creational Patterns: Singleton

- Lazy instantiation

```
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();
```

Creational Patterns: Singleton

- Would you prefer eager or lazy instantiation for an HTTPRequest class?
  - handles authentication
  - definitely needed for any HTTP transaction
- Would you prefer eager or lazy instantiation for a Comparator class?
  - compares objects
  - may or may not be used at runtime

Creational Patterns: 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
- Requires the class being interned to be immutable. Why?

Creational Patterns: Interning

```
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 + “)”;
	}
}
```

Creational Patterns: Interning

```
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) {…}
}
```

If our point was represented with r and theta, we'd need to constrain them for use in the key. Otherwise, we'd have "5, pi" and "5, 3pi" as different entries in our map even though they are the same abstract value.

Creational Patterns: Factories

```
public class City {
	public Stereotype getStereotypicalPerson() {…}
}

City seattle = new City();
seattle.getSterotypicalPerson();
// we want a SeattleStereotype
```

Creational Patterns: 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
    - Object provides the interface for creating families of related/dependent objects without specifying their concrete classes

Creational Patterns: Factory Method

```
public class City {
	public Stereotype getStereotypicalPerson() {…}
}
public class Seattle extends City {
	@Override
	public Stereotype getStereotypicalPerson() {
		return new SeattleStereotype();
	}
}
City seattle = new Seattle();
seattle.getSterotypicalPerson();
```

Creational Patterns: Factory Object

```
interface StereotypeFactory {
        Stereotype getStereotype();
}
class SeattleStereotypeFactory implements StereotypeFactory {
        public Stereotype getStereotype() {
                return new SeattleStereotype();
        }
}
public class City {
        public City(StereotypeFactory f) {…}
        public Stereotype getStereotypicalPerson() {
                f.getStereotype();
        }
}
City seattle = new City(new SeattleStereotypeFactory());
seattle.getSterotypicalPerson();
```

Creational Patterns: 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

Creational Patterns: 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;
        }
}
```

Creational Patterns: Builder

```
public class NutritionFacts {
        private final int servingSize, servings, calories, fat, sodium;

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

Creational Patterns: Builder

- 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

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

Pattern	Functionality	Interface	Purpose
Adapter	same	different	modify the interface
Decorator	different	same	extend behavior
Proxy	same	same	restrict access

Structural Patterns: Adapter

- Changes an interface without changing functionality
  - Rename a method
  - Convert units
- Why?
  - We already have a class that does most/all what we want
  - It just doesn't do it in the form we want

Structural Patterns: Adapter

- MSR has a library and API for image composition
  - `public void photoComposite(List<Image> photoStack, List<Weight> objectives, Image output);`
- MS PhotoGallery wants to use this API to make new features
  - `PhotoGalleryDisplay` object contains a `DisplayImage` that is a 2d array of pixels representing the main image in view
- Both codebases are very large and changes will affect a lot of other pieces

Structural Patterns: Adapter

```
public class PhotoSynthAdapter {
    public DisplayImage
makePanoramic(List<Image> photos)
    {
     Image output = new Image();
     List<Weight> objectives =
generatePanoWeights();
     photoComposite(photos, objectives,
output);
     return ImageToDisplayImage(output);
    }

    public DisplayImage removeTourists(…)
```

Structural Patterns: Adapter

- Other examples:
  - Angles passed in using radians vs. degrees
  - Bytes vs. strings
  - Hex vs. decimal numbers

Structural Patterns: 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()`

```
public abstract class Cake{
	public abstract double getWeight();
	public abstract String getDescription();
}

public class PlainCake extends Cake{
	public double getWeight()
	{
		return 8.0;
	}

	public String getDescription()
	{
		return “sponge cake”;
	}
}
```

from http://en.wikipedia.org/wiki/Decorator_pattern

```
public abstract class CakeDecorator extends Cake
{
    protected final Cake decoratedCake;
    protected String separator = ", ";

    public CakeDecorator (Cake decoratedCake)
    {
        this.decoratedCake = decoratedCake;
    }

    public double getWeight()
    {
        return decoratedCake.getWeight();
    }

    public String getDescription()
    {
         return decoratedCake.getDescription();
    }
}
```

from http://en.wikipedia.org/wiki/Decorator_pattern

```
class Frosting extends CakeDecorator
{
    public Frosting (Cake decoratedCake)
    {
        public double getWeight()
        {
            return super.getWeight() + 4.0;
        }

        public String getDescription()
        {
            return super.getDescription +
separator + “vanilla frosting”;
        }
    }
}
```

from http://en.wikipedia.org/wiki/Decorator_pattern

```
public static void main(String[] args)
{
    Cake c = new PlainCake();
    System.out.println("Weight: " +
c.getWeight() + "; Contains: " +
c.getDescription());

    c = new
    System.ou
c.getWeight()
c.getDescripti
…
    c = new
…
}
```

Weight: 8.0
Contains: sponge cake

Weight: 12.0
Contains: sponge cake, vanilla frosting

Weight: 15.5
Contains: sponge cake, vanilla frosting, buttercream roses

from http://en.wikipedia.org/wiki/Decorator_pattern

Structural Patterns: Proxy

- Wraps the class while maintaining the same interface and same 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