CREATIONAL DESIGN PATTERNS

CSE 219, Computer Science III
DESIGN PATTERNS

• In software engineering, a design pattern is a reusable solution to a common and repeating problem within a specific context
  • Cannot be converted directly into source code
  • It is simply a description or “template” for solving the repeating problem
  • Can be reused in many situations
• Think of design patterns as a formal way of coming up with “best practices” that a programmer can use to solve common problems
CATEGORIZING DESIGN PATTERNS

There are a few different ways of classifying design patterns into groups:

- Creational, Structural and Behavioral
- Architectural design pattern
  - Applied at the software architecture level
  - E.g., the “model-view-controller” (MVC) pattern
### CATEGORIZING DESIGN PATTERNS

<table>
<thead>
<tr>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Factory</td>
<td>• Decorator</td>
<td>• Strategy</td>
</tr>
<tr>
<td>• Singleton</td>
<td>• Adapter</td>
<td>• Template</td>
</tr>
<tr>
<td>• <strong>Builder</strong></td>
<td>• Facade</td>
<td>• Observer</td>
</tr>
<tr>
<td>• Prototype</td>
<td>• Flyweight</td>
<td>• Command</td>
</tr>
<tr>
<td></td>
<td>• Bridge</td>
<td>• Iterator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• State</td>
</tr>
</tbody>
</table>

- Design patterns that deal with object creation mechanisms
- Different situations may call for different ways of creating objects
Factories make stuff!

Factory classes make object-oriented stuff like … well … objects!

**Why do we need factory classes when every class can have constructors?**

- Factory classes use constructors
- But we may want to keep the constructor hidden
  - in order to prevent misuse
  - to provide a more convenient API
  - to provide a “one stop shop” to get objects of a specific type
WHAT OBJECTS DO FACTORIES MAKE?

- Typically objects of the same *family*
  - common ancestor
  - same apparent type
  - **may have different actual type**
- Example of Factory Patterns in the Java SWING API:
  - `BorderFactory.createXXXBorder` methods
    - return apparent type of interface `Border`
    - return actual types of `BevelBorder`, `EtchedBorder`, etc ...
- Factory classes in security packages:
  - `java.security.KeyFactory`, `java.security.cert.CertificateFactory`
• First, we build a Dog interface
• Any dog returned by the factory must implement this interface
  • We will just specify a “speak” method

```java
interface Dog {
    public void speak ();
}
```
A FEW CONCRETE CLASSES

class Poodle implements Dog {
    public void speak() { System.out.println("Arf"); }
}

class Rottweiler implements Dog {
    public void speak() { System.out.println("WOOF!"); }
}

class SiberianHusky implements Dog {
    public void speak() { System.out.println("What's up?"); }
}
class DogFactory {
    public static Dog getDog(String criteria) {
        if (criteria.equals("small")) {
            return new Poodle();
        } else if (criteria.equals("big")) {
            return new Rottweiler();
        } else if (criteria.equals("working")) {
            return new SiberianHusky();
        }
        return null;
    }
}
abstract class Car { ... }

class Bmw extends Car { ... }

class Bmw320 extends Bmw { ... }

abstract class CarFactory { public abstract Car createCar(String type);}

class BmwFactory extends CarFactory {
    public Car createCar(String type) {
        return "Bmw320".equals(type) ? new Bmw320() : new Bmw();
    }
}

public class Dealer {

    public static void main(String[] args) {
        Car bmw1 = new BmwFactory().createCar("Bmw320");
        Car bmw2 = new BmwFactory().createCar("Bmw");
        //Car camry1 = new ToyotaFactory().createCar("Camry");
    }
}
ADVANTAGES OF THE FACTORY PATTERN

• The programmer using the Factory class never needs to know about the actual class or type
  • simplifies use for programmer
  • fewer classes to learn
THE SINGLETION PATTERN

• Define a type where only one object of that type may be constructed
  • make the constructor private!
  • singleton object favorable to fully static class, why?
    • can be used as a method argument
    • class can be extended

• **What makes a good singleton candidate?**
  • central app organizer class
    • e.g., a basic/simple Web/FTP server can be a singleton class
  • something everybody needs
    • e.g., a class that stores global properties for the application, a logging service class
public class PropertiesManager {

    private static PropertiesManager singleton;

    private PropertiesManager() {}

    public static PropertiesManager getPropertiesManager() {
        if (singleton == null)
            singleton = new PropertiesManager();
        return singleton;
    }
}
SINGLETON: THE ADVANTAGES

• Other classes can easily use the PropertiesManager
  
  PropertiesManager pm = PropertiesManager.getPropertiesManager();

• They don’t have to worry about
  
  • passing around objects
  
  • object consistency

• But
  
  • only good for classes that will need at most one instance
• **Java API**
  
  • `java.lang.Runtime#getRuntime()`
    
    - `public static Runtime getRuntime()` – returns the runtime object associated with the current Java application. Most of the methods in the Runtime class are instance methods, and must be invoked w.r.t. the current runtime object.
  
  • `java.awt.Desktop#getDesktop()`
    
    - `public static Desktop getDesktop()` – returns the Desktop instance of the current browser context.
      
      - Not supported on all platforms, use `isDesktopSupported()` to check
class Singleton {
    private static Singleton instance = new Singleton();  // eager init
    private Singleton() { }
    public static Singleton getInstance() { return instance; }

    public synchronized static Singleton getInstanceSync() {  // lazy init
        return instance == null ? new Singleton() : instance;
    }
}

denum SingletonEnum {
    Elvis;  // there was only one Elvis ...
    
    public String getSong() { return "Heartbreak"; }
}

class Main {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        System.out.println(SingletonEnum.Elvis.getSong());
    }
}
CATEGORIZING DESIGN PATTERNS

<table>
<thead>
<tr>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Factory</td>
<td>• Decorator</td>
<td>• Strategy</td>
</tr>
<tr>
<td>• Singleton</td>
<td>• Adapter</td>
<td>• Template</td>
</tr>
<tr>
<td>• <strong>Builder</strong></td>
<td>• Facade</td>
<td>• Observer</td>
</tr>
<tr>
<td>• Prototype</td>
<td>• Flyweight</td>
<td>• Command</td>
</tr>
<tr>
<td></td>
<td>• Bridge</td>
<td>• Iterator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• State</td>
</tr>
</tbody>
</table>

- Design patterns that deal with object creation mechanisms
  - Different situations may call for different ways of creating objects
THE TELESCOPING CONSTRUCTOR

• An anti-pattern

• an increase of object constructor parameter combination leads to an exponential list of constructors

```java
public class FourStrings {
    String a, b, c, d;

    public FourStrings(String a, String b, String c, String d) {
        this.a = a; this.b = b; this.c = c; this.d = d;
    }

    public FourStrings(String a, String b, String c) {
        this(a, b, c, null);
    }
}
```
THE BUILDER PATTERN

• Instead of using numerous constructors, the builder pattern uses another object, a builder
  • that receives each initialization parameter step by step
  • and then returns the resulting constructed object at once.
• Use this to
  • encapsulate the construction of an object
  • allow the construction to be ‘incremental’
• Good for creating complex structures
  • that need a lot of other smaller custom-bits already in place
• Example: a JSONBuilder
A BUILDER PATTERN EXAMPLE

- Planning a vacation

Each vacation is planned over some number of days.

Each day can have any combination of hotel reservations, tickets, meals and special events.
PLANNING A VACATION

- **A flexible construction design is needed**
- **Lots of Customization:**
  - some customers might not want a hotel
  - some might want multiple rooms in multiple hotels
  - some might want restaurant reservations
  - some might want stuff no one else does
- **We need:**
  - a flexible data structure to represent such variations
  - a sequence of (potentially) complex steps to create the planner
PLANNING A VACATION

The Client directs the builder to construct the planner.

```java
builder.buildDay(date);
builder.addHotel(date, "Grand Facadian");
builder.addTickets("Patterns on Ice");

// plan rest of vacation
Planner yourPlanner = builder.getVacationPlanner();
```

The client uses an abstract interface to build the planner.

The concrete builder creates real products and stores them in the vacation composite structure.

```java
AbstractBuilder

buildDay()
addHotel()
addReservation()
addSpecialEvent()
addTickets()
getVacationPlanner()
```

```java
VacationBuilder

vacation

buildDay()
addHotel()
addReservation()
addSpecialEvent()
addTickets()
getVacationPlanner()
```
We want to create an immutable “User”

but we don’t know all the properties of the user (yet)

```java
public class User {
    private final String firstName; // required
    private final String lastName; // required
    private final int age; // optional
    private final String phone; // optional
    private final String address; // optional
}
```
• Naïve coding will end up with the telescoping constructor anti-pattern

```java
public User(String fname, String lname) {
    this(fname, lname, 0);
}
public User(String fname, String lname, int age) {
    this(fname, lname, age, "");
}
public User(String fname, String lname, int age, String phone) {
    this(fname, lname, age, phone, "");
}
public User(String fname, String lname, int age, String phone, String address) {
    this.firstName = fname;
    this.lastName  = lastName;
    this.age       = age;
    this.phone     = phone;
    this.address   = address;
}...
```
THE BUILDER PATTERN SOLUTION

// inner class
public static class UserBuilder {
    private final String firstName;
    private final String lastName;
    private int age;
    private String phone;
    private String address;

    public UserBuilder(String fname, String lname) {
        this.firstName = fname;
        this.lastName = lname;
    }

    public UserBuilder age(int age) {
        this.age = age; return this;
    }

    public UserBuilder phone(String phone) {
        this.phone = phone; return this;
    }

    public UserBuilder address(String address) {
        this.address = address; return this;
    }

    public User build() {
        return new User(this);
    }
}
THE BUILDER PATTERN SOLUTION

• The constructor is private
  • class can not be directly instantiated from the client code.

• The class is immutable
  • all attributes are final and they’re set in the constructor

```java
public User getUser() {
    return new User.UserBuilder("John", "Smith")
        .age(50)
        .phone("1234567890")
        .address("Main St. 1234")
        .build();
}
```
ADVANTAGES OF THE BUILDER PATTERN

• Encapsulates the way a complex object is constructed.
• Allows objects to be constructed in a multistep and varying process
  • as opposed to “one-step” factories.
• Hides the internal representation of the product from the client.
• Product implementations can be swapped in and out if the client only sees an abstract interface.
<table>
<thead>
<tr>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factory</td>
<td>Decorator</td>
<td>Strategy</td>
</tr>
<tr>
<td>Singleton</td>
<td>Adapter</td>
<td>Template</td>
</tr>
<tr>
<td>Builder</td>
<td>Facade</td>
<td>Observer</td>
</tr>
<tr>
<td><strong>Prototype</strong></td>
<td>Flyweight</td>
<td>Command</td>
</tr>
<tr>
<td></td>
<td>Bridge</td>
<td>Iterator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>State</td>
</tr>
</tbody>
</table>

*Design patterns that deal with object creation mechanisms

* Different situations may call for different ways of creating objects
THE PROTOTYPE PATTERN

• First of all, this is **not** the same as software prototyping!
• Use scenario
  • when the type of objects to create is determined by a ‘prototypical instance’
    • this first prototypical instance is cloned to produce new instances
  • when creating a new instance is either expensive or complicated with the traditional use of “new” keyword
    • modern JVMs are highly optimized, so the ‘expensive’ part is not that relevant any more
IMPLEMENTING THE PROTOTYPE PATTERN

• Declare an abstract base class with a pure virtual `clone()` method
• Classes derived from the abstract base class implement `clone()`
• The client, instead of writing code that invokes the “new” operator on a hard-coded class name
  • calls the `clone()` method on the prototype, or
  • calls a factory method with a parameter designating the particular concrete derived class, or
  • invokes the `clone()` method through some mechanism provided by another design pattern
public abstract class Prototype implements Cloneable {
    public abstract Prototype clone();
}

public class ConcretePrototype1 extends Prototype {
    public Prototype clone() { return super.clone(); } // override
}

public class ConcretePrototype2 extends Prototype {
    public Prototype clone() { return super.clone(); } // override
}
ADVANTAGES OF THE PROTOTYPE PATTERN

• Hides the complexities of making new instances from the client
• Provides the option for the client to generate objects whose type is not known
• In some circumstances, copying an object can be more efficient than creating a new object
USES AND DRAWBACKS OF THE PROTOTYPE PATTERN

- Prototype should be considered when
  - a system must create new objects of many types in a complex class hierarchy
- A drawback to using the Prototype
  - making a copy of an object can sometimes be complicated.
FACTORY & PROTOTYPE

- A factory might store a set of prototypes from which it clones and return objects.
public class PrototypeFactory {

    interface Minion { Minion clone(); }

    static class Stuart implements Minion {
        public Minion clone() { return new Stuart(); }
        public String toString() { return "Stuart"; }
    }

    static class Kevin implements Minion {
        public Minion clone() { return new Kevin(); }
        public String toString() { return "Kevin"; }
    }

    static class Bob implements Minion {
        public Minion clone() { return new Bob(); }
        public String toString() { return "Banana!!"; }
    }

    ...

static class GrusLab {

    private static Map<String, Minion> prototypes = new HashMap<>();

    static {
        prototypes.put("stuart",new Stuart());
        prototypes.put("kevin", new Kevin());
        prototypes.put("bob", new Bob());
    }

    public static Minion makeObject( String s ) {
        return ((Minion) prototypes.get(s)).clone();
    }

}

public static void main( String[] args ) {
    for (int i=0; i < args.length; i++) {
        System.out.print( GrusLab.makeObject(args[i]) + "   ");
    }

}