Design Patterns

Based on material by Michael Ernst, University of Washington
Announcements

- HW6 due Friday November 10th
  - Submitty page for HW5 RESUBMIT is up
  - Submitty page for HW6 is up
  - Check Details of compilation tests

- HW grades & feedback in Submitty
- Quiz and Exam grades in Submitty

- Exam 2 is graded
Outline of today’s class

- Unified Modeling Language (UML)
- Design patterns
  - Intro to design patterns
  - Creational patterns
    - Factories: Factory method, Factory object, Prototype
    - Sharing: Singleton, Interning
  - Structural patterns
    - Adapter, Composite, Decorator, Proxy
Aside: UML Class Diagrams

- Unified Modeling Language (UML) is the “lingua franca” of object-oriented modeling and design

- **UML class diagrams** show classes, their interrelationships (*inheritance and composition*), their attributes and operations

- Also, UML sequence diagrams show dynamics of the system
Classes and Inheritance

Notation: Boxes are classes (e.g., Shape, Circle).

\[\triangle\] denotes inheritance (Square is a subclass of Shape.)

Italics denote abstract (Shape is abstract, draw() is abstract)
Classes and Inheritance

```
<<interface>>
MapEntry

equals()
getKey() …
```

denotes interface inheritance
(SimpleEntry implements interface MapEntry)

```
SimpleEntry
equals()
getKey() …
```

ImmutableEntry extends abstract class SimpleEntry
Associations

class Stack {
    Link top;
    ...
}

class Link {
    Link next;
    int data;
    ...
}

A UML association often represents a composition relationship: object of one class encloses the object(s) of the other (the rep). Above, Stack -> Link is a typical composition (has-a) relationship --- the Stack encloses/encapsulates its Link.

push() and pop() are operations

top and next are associations

int data is an attribute
Exercise

- Draw a UML class diagram that shows the interrelationships between the classes from HW0
UML

- Can use UML to model abstract concepts (e.g., Meeting) and their interrelationships
  - Attributes and associations correspond to specification fields
  - Operations correspond to ADT operations
- Can use UML to express designs
  - Close correspondence to implementation
  - Attributes and associations correspond to representation fields
  - Operations correspond to methods
A design pattern is a solution to a design problem that occurs over and over again.

The reference: Gang of Four (GoF) book
- “Design Patterns: Elements of Reusable Object-Oriented Software”, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (the Gang of Four), Addison Wesley 1995
- Documents 23 still widely used design patterns
Design Patterns

- Design patterns promote extensibility and reuse
  - Help build software that is open to extension but closed to modification (the “Open/Closed principle”)

- (Majority of) design patterns exploit subtype polymorphism
Why Should You Care?

- You can discover those solutions on your own
  - But you shouldn’t have to
- A design pattern is a known solution to a known problem
  - Well-thought software uses design patterns extensively
  - Understanding software requires knowledge of design patterns
Design Patterns

- Three categories
  - Creational patterns (deal with object creation)
  - Structural patterns (control object structure, also known as heap layout)
  - Behavioral patterns (control object behavior)
Creational Patterns

- Problem: constructors (in Java and other OO languages) are inflexible
  1. Can’t return a subtype of the type they belong to
  2. Always return a fresh new object, can’t reuse

- “Factory” creational patterns present a solution to the first problem
  - Factory method, Factory object, Prototype

- “Sharing” creational patterns present a solution to the second problem
  - Singleton, Interning
An Example

class Race {
    Race createRace() {
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle(); ...
    }
}

class TourDeFrance extends Race {
    Race createRace() {
        Bicycle bike1 = new RoadBicycle();
        Bicycle bike2 = new RoadBicycle(); ...
    }
}

class Cyclocross extends Race {
    Race createRace() {
        Bicycle bike1 = new MountainBicycle();
        Bicycle bike2 = new MountainBicycle(); ...
    }
}
Using a Factory Method

class Race {
    Bicycle createBicycle() { ... }
    Race createRace() {
        Bicycle bike1 = this.createBicycle();
        Bicycle bike2 = this.createBicycle(); ... 
    }
}

class TourDeFrance extends Race {
    Bicycle createBicycle() {
        return new RoadBicycle();
    }
}

class Cyclocross extends Race {
    Bicycle createBicycle() {
        return new MountainBicycle();
    }
}
Parallel Hierarchies

Can extend with new Races and Bikes with no modification to createRace!

```
Race
createBicycle()
createRace();
```

```
Bicycle
setSpeed()
```

```
RoadBicycle
setSpeed()
...
```

```
MountainBicycle
setSpeed()
...
```

Client

```
TourDeFrance
createBicycle()
```

```
Cyclocross
createBicycle()
```

creates
Another Factory Method Example

Motivation: Applications share common functions, but work with different documents

Factory method; usually abstract

```
MyApplication
createDocument()
```

```
Document doc = createDocument();
...
docs.add(doc);
doc.open();
...
```

```
return new MyDocument();
```
Yet Another Factory Method

Example

```
abstract class MazeGame {
    abstract Room createRoom();
    abstract Wall createWall();
    abstract Door createDoor();
    Maze createMaze() {
        ...
        Room r1 = createRoom(); Room r2 = ...
        Wall w1 = createWall(r1,r2);
        Door d1 = createDoor(w1);
        ...
    }
    ...
}
```
class EnchantedMazeGame extends MazeGame {
    Room createRoom() {
        return new EnchantedRoom(castSpell());
    }
    Wall createWall(Room r1, Room r2) {
        return new EnchantedWall(r1, r2, castSpell());
    }
    Door createDoor(Wall w) {
        return new EnchantedDoor(w, castSpell());
    }
}

// Inherits createMaze from MazeGame
class BombedMazeGame extends MazeGame {
    Room createRoom() {
        return new RoomWithBomb();
    }
    Wall createWall(Room r1, Room r2) {
        return new BombedWall(r1,r2);
    }
    Door createDoor(Wall w) {
        return new DoorWithBomb(w);
    }
}

// Again, inherit createMaze from MazeGame
Factory Methods in the JDK

**DateFormat** class encapsulates knowledge on how to format a Date

- Options: Just date? Just time? date+time? where in the world?

```java
DateFormat df1 = DateFormat.getDateInstance();
DateFormat df2 = DateFormat.getTimeInstance();
DateFormat df3 = DateFormat.getDateInstance(DateFormat.FULL, Locale.FRANCE);
Date today = new Date();
df1.format(today);   // “Jul 4, 1776”
df2.format(today);   // “10:15:00 AM”
df3.format(today);   // “jeudi 4 juillet 1776”
```
Factory Object Pattern
(also known as Abstract Factory)

Motivation: Encapsulate the factory methods into one class. Separate control over creation.

```
AbstractMazeFactory
  createRoom()
  createWall()
  createDoor()

Client: MazeGame

Wall
  EnchantedWall
  BombedWall

Room
  EnchantedRoom
  BombedRoom

BombedMazeFactory
  createRoom()
  createWall()
  createDoor()

EnchantedMazeFactory
  createRoom()
  createWall()
  createDoor()
```
Let’s Use a Factory Object

class MazeGame {
    AbstractMazeFactory mfactory;
    MazeGame(AbstractMazeFactory mfactory) {
        this.mfactory = mfactory;
    }
    Maze createMaze() {
        ...
        Room r1 = mfactory.createRoom();
        Room r2 = ...
        Wall w1 = mfactory.createWall(r1,r2);
        Door d1 = mfactory.createDoor(w1);
        ...
    }
}
The Factory Hierarchy

abstract class AbstractMazeFactory {
    Room createRoom();
    Wall createWall(Room r1, Room r2);
    Door createDoor(Wall w);
}
class EnchantedMazeFactory extends AbstractMazeFactory {
    Room createRoom() { creates enchanted ... }
    Wall createWall(...) { creates enchanted ... }
    Door createDoor(...) { creates enchanted ... }
}
class BombedMazeFactory extends AbstractMazeFactory {
    // analogous
}
class Race {
    BikeFactory bfactory;
    Race() { bfactory = new BikeFactory(); }
    Race createRace() {
        Bicycle bike1 = bfactory.createBicycle();
        Bicycle bike2 = bfactory.createBicycle();
        ...
    }
}

class TourDeFrance extends Race {
    // constructor
    TourDeFrance() {
        bfactory = new RoadBikeFactory()
    }
}

// analogous constructor for Cyclocross
class BikeFactory {
    Bicycle createBicycle() { ... }
    Frame createFrame() { ... }
    Wheel createWheel() { ... }
}

class RoadBikeFactory extends BikeFactory {
    Bicycle createBicycle() {
        return new RoadBicycle();
    }
}

class MountainBikeFactory extends BikeFactory {
    Bicycle createBicycle() {
        return new MountainBicycle();
    }
}
Separate Control Over Races and Bicycles

class Race {
    BikeFactory bfactory;
    Race(BikeFactory bfactory) {
        this.bfactory = bfactory;
    }
    Race createRace() {
        Bicycle bike1 = bfactory.createBicycle();
        Bicycle bike2 = bfactory.createBicycle();
        ...
    }
}

- No special constructor for TourDeFrance and Cyclocross
Separate Control Over Races and Bicycles

- Client can specify the race and the bicycle separately:
  
  ```java
  Race race=new TourDeFrance(new TricycleFactory());
  ```

- To specify a different race/bicycle need only change one line:
  
  ```java
  Race race=new Cyclocross(new TricycleFactory());
  ```

  or

  ```java
  Race race=new Cyclocross(new MountainBikeFactory());
  ```

- Rest of code, uses `Race`, stays the same!
Dependency Injection

- In Java, we can decide what `Factory` to initialize with at runtime!
- External dependency injection:

```java
BikeFactory f = (BikeFactory) DependencyManager.get("BikeFactory");
Race race = new Cyclocross(f);
```

- An external file specifies a value for “BikeFactory”, factory in plain text, say “TricycleFactory”
- `DependencyManager` reads file and uses Java reflection to load and instantiate class, `TricycleFactory`
The Prototype Pattern

- Every object itself is a factory
- Each class can define a `clone` method that returns a copy of the receiver object
  ```java
  class Bicycle {
      Bicycle clone() { ... }
  }
  ```
- `Object` has `protected Object clone()`
- In Java 1.4 and earlier an overriding method cannot change the return type. Since Java 5, an overriding method can change it covariantly
Using Prototypes

class Race {
    Bicycle bproto;
    // constructor
    Race(Bicycle bproto) {
        this.bproto = bproto;
    }
    Race createRace() {
        Bicycle bike1 = bproto.clone();
        Bicycle bike2 = bproto.clone();
        ...
    }
}

How do we specify the race and the bicycle?
new TourDeFrance(new Tricycle());
Outline of today’s class

- Unified Modeling Language, briefly

- Design patterns
  - Intro to design patterns
  - Creational patterns
    - Factories: Factory method, Factory object, Prototype
    - Sharing: Singleton, Interning
  - Structural patterns
    - Adapter, Composite, Decorator, Proxy
Sharing

- Recall that constructors always return a new object, never a pre-existing one.
- In many situations, we would like a pre-existing object.
- **Singleton** pattern: only one object ever exists.
  - A factory object is almost always a singleton.
- **Interning** pattern: only one object with a given abstract value exist.
Singleton Pattern

- Motivation: there must be a single instance of the class

```java
class Bank {
    private Bank() { ... }
    private static Bank instance;
    public static Bank getInstance() {
        if (instance == null)
            instance = new Bank();
        return instance;
    }
    // methods of Bank
}
```

Factory method --- it produces the instance of the class
```java
public class UserDatabaseSource implements UserDatabase {
    private static UserDatabase theInstance = new UserDatabaseSource();
    private UserDatabaseSource() { … }
    public static UserDatabase getInstance() {
        return theInstance;
    }
    public User readUser(String username) { … }
    public void writeUser(User user) { … }
}
```
Interning Pattern

- Not a GoF design pattern
- Reuse existing object with same value, instead of creating a new one
  - E.g., why create multiple Strings “car”? Create a single instance of String “car”!
  - Less space
  - May compare with `==` instead of `equals` and speed the program up
- Interning applied to immutable objects only
Interning Pattern

(Street-Segment) -> 1-100 (Street-NumberSet) -> "Univ. Way" (String) -> "O2139" (String) -> 101-200 (Street-NumberSet)

(Street-Segment) -> 1-100 (Street-NumberSet) -> "Univ. Way" (String) -> "O2139" (String) -> 101-200 (Street-NumberSet)
Interning Pattern

- Maintain a collection of all names
- If an object already exists return that object

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

- Java supports interning for Strings: `s.intern()` returns a canonical representation of `s`

Why not a HashSet but HashMap?
Why Not HashSet?

- Maintain a collection of all names
- If an object already exists return that object

```java
HashSet<String> names;
String canonicalName(String n) {
    if (names.contains(n))
        return n;
    else {
        names.add(n);
        return n;
    }
}
```
What’s wrong with java.lang.Boolean?

```java
public class Boolean {
    private final boolean value;
    public Boolean(boolean value) {
        this.value = value;
    }
    public static Boolean FALSE = new Boolean(false);
    public static Boolean TRUE = new Boolean(true);
    public static Boolean valueOf(boolean value) {
        if (value) return TRUE;
        else return FALSE;
    }
}
```

Factory method --- produces the appropriate instance
What’s wrong with java.lang.Boolean?

- Boolean constructor should have been private: would have forced *interning* through `valueOf`
- Spec warns against using the constructor
- Joshua Bloch, lead designer of many Java libraries, in 2004: The **Boolean** type 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.