Design Patterns

Based on material by Michael Ernst, University of Washington

Announcements

- HW5 due Tuesday November 10th
  - Check Details of compilation tests
- HW grades & feedback in Homework Server
- Quiz and Exam grades are in the LMS
- Exam 2 is graded

Outline of today's class

- The 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).

\[\text{Square} \rightarrow \text{Shape}]\]
\[\text{Circle} \rightarrow \text{Shape}]\]
\[\text{draw()} \rightarrow \text{Shape}]\]

\[\text{ImmutableEntry}\]
\[\text{equals()} \rightarrow \text{MapEntry}]\]
\[\text{getKey()} \rightarrow \text{MapEntry}]\]

\[\text{SimpleEntry}\]
\[\text{equals()} \rightarrow \text{ImmutableEntry}]\]
\[\text{getKey()} \rightarrow \text{ImmutableEntry}]\]

\[\text{ImmutableEntry}\]
\[\text{equals()} \rightarrow \text{ImmutableEntry}]\]
\[\text{getKey()} \rightarrow \text{ImmutableEntry}]\]

\[\text{SimpleEntry}\]
\[\text{equals()} \rightarrow \text{SimpleEntry}]\]
\[\text{getKey()} \rightarrow \text{SimpleEntry}]\]
**Associations**

```java
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. Above, `Stack -> Link` is a typical composition (has-a) relationship — the Stack encloses/encapsulates its Link.

**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

**Design Patterns**

- 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!

Another Factory Method Example

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

Diagram:

- Client
- Bicycle
- RoadBicycle
- MountainBicycle
- MyDocument
- MyApplication
- Document
- Application
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);
        ...
    } ...
}

// Inherits createMaze from MazeGame

Yet Another Factory Method

Example

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

Yet Another Factory Method

Example
class BombedMazeGame extends MazeGame {
    Room createRoom() {
        return new BombedRoom();
    }
    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?

    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"

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);
        ...
    }
}

Factory Object Pattern
(also known as Abstract Factory)

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

Let’s Use Factory Object

class Race {
    BikeFactory bfactory;
    Race() {
        bfactory = new BikeFactory();
    }
    Race createRace() {
        Bicycle bike1 = bfactory.createBicycle();
        Bicycle bike2 = bfactory.createBicycle();
    }
}
class TourDeFrance extends Race {
    TourDeFrance() {
        bfactory = new RoadBikeFactory();
    }
}

Separate Control Over Races and Bicycles

Client can specify the race and the bicycle separately:
Race race = new TourDeFrance(new TricycleFactory());
To specify a different race/bicycle need only change one line:
Race race = new Cyclocross(new TricycleFactory());
Or
Race race = new Cyclocross(new MountainBikeFactory());

Rest of code, uses Race, stays the same!

Separate Control Over Races and Bicycles

Class BikeFactory {
    Bicycle createBicycle() { return new Bicycle(); }
    Frame createFrame() { ... }
    Wheel createWheel() { ... }
}
class RoadBikeFactory extends BikeFactory {
    Bicycle createBicycle() { return new RoadBicycle(); }
}
class MountainBikeFactory extends BikeFactory {
    Bicycle createBicycle() { return new MountainBicycle(); }
}

Dependency Injection

In Java, we can decide what Factory to initialize with at runtime!
External dependency injection:
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,
The Prototype Pattern

- Every object itself is a factory
- Each class can define a \texttt{clone} method that returns a copy of the receiver object

```java
class Bicycle {
    Bicycle clone() { ... }
}
```

- Often \texttt{Object} is the return type of \texttt{clone}
  - \texttt{Object} declares \texttt{protected Object clone()}
  - In Java 1.4 and earlier an overriding method cannot change the return type. Now an overriding method can change it covariantly

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

Using Prototypes

```java
class Race {
    Bicycle bproto;
    // constructor
    Race(Bicycle bproto) {
        this.bproto = bproto;
    }
}
```

How do we specify the race and the bicycle?
```java
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

Fall 15 CSCI 2600, A Milanova

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

Fall 15 CSCI 2600, A Milanova

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
}
```

Fall 15 CSCI 2600, A Milanova (example due to Michael Ernst)

Another Singleton Example

```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) { ... }
}
```

Static initializer --- executed when class is loaded
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

Why not a HashSet but HashMap?

- 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 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;
    }
}
```

- 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 true or multiple false, and I've seen programs that produce millions of trues and millions of false creating needless work for the garbage collector.

So, in the case of immutables, I think factory methods are great.
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

Wrappers

- A wrapper uses composition/delegation
- A wrapper is a thin layer over an encapsulated class
  - Modify the interface
  - Extend behavior
  - Restrict access to encapsulated object
- The encapsulated object (delegator) does most work
  - Adapter: modifies interface, same functionality
  - Decorator: same interface, extends functionality
  - Proxy: same interface, same functionality

Wrappers

- A wrapper uses composition/delegation
- A wrapper is a thin layer over an encapsulated class
  - Modify the interface
  - Extend behavior
  - Restrict access to encapsulated object
- The encapsulated object (delegator) does most work
  - Adapter: modifies interface, same functionality
  - Decorator: same interface, extends functionality
  - Proxy: same interface, same functionality

Adapter Pattern

- Motivation: reuse a class with an interface different that the class’ interface

Adapter Pattern

- The purpose of the Adapter is: change an interface, without changing the functionality of the encapsulated class
  - Allows reuse of functionality
  - Protects client from modification
- Reasons
  - Rename methods
  - Convert units
  - Implement a method in terms of another

Adapter Example: Scaling Rectangles

```java
interface Rectangle {
    void scale(int factor); // grow or shrink by factor
    void setWidth();
    float getWidth();
    float area(); ...
}

class Client {
    void clientMethod(Rectangle r) {
        r.scale(2);
    }
}
class NonScalableRectangle {
    void setWidth(); // no scale method!
}
```

Class Adapter

```java
class ScalableRectangle extends NonScalableRectangle implements Rectangle {
    void scale(int factor) {
        setWidth(factor*getWidth());
    }
    setHeight(factor*getHeight());
}
```
Object Adapter

- Object adapter adapts via delegation: it forwards work to delegate

```java
class ScalableRectangle2 implements Rectangle {
    NonScalableRectangle r; // delegate
    ScalableRectangle2(NonScalableRectangle r) {
        this.r = r;
    }
    void scale(int factor) {
        setWidth(factor * getWidth());
        setHeight(factor * getHeight());
    }
    float getWidth() { return r.getWidth(); }
}
```

Subclassing Versus Delegation

- Subclassing
  - Automatically gives access to all methods in the superclass
  - More efficient
- Delegation
  - Permits removal of methods
  - Multiple objects can be composed
  - Bottom line: more flexible

Exercise

- A Point-of-Sale system needs to support services from different third-party vendors:
  - Tax calculator service from different vendors
  - Credit authorization service from different vendors
  - Inventory systems from different vendors
  - Accounting systems from different vendors
  - Each vendor service has its own API, which can’t be changed
  - What design pattern helps solve this problem?

The Solution: Object Adapter

- POS System
  - ITaxCalculatorAdapter
    - getTaxes(Sale) : List of TaxLineItems
  - IAccountingAdapter
    - postReceivable(CreditPayment)
    - postSale(Sale)
  - ICreditAuthorizationServiceAdapter
    - requestApproval(CreditPayment, TerminalID, MerchantID)

Exercise

- Who creates the appropriate adapter object?
  - Is it a good idea to let some domain object from the Point-of-Sale system (e.g., Register, Sale) create the adapters?
    - That would assigns responsibility beyond domain object’s logic. We would like to keep domain classes focused, so, this is not a good idea
- How to determine what type of adapter object to create? We expect adapters to change.
- What design patterns solve this problem?

The Solution: Factory

- ServiceFactory
  - accountingAdapter : IAccountingAdapter
  - inventoryAdapter : IInventoryAdapter
  - taxCalculatorAdapter : ITaxCalculatorAdapter
  - getAccountingAdapter() : IAccountingAdapter
  - getInventoryAdapter() : IInventoryAdapter
  - getTaxCalculatorAdapter() : ITaxCalculatorAdapter
Using the Factory

public ITaxCalculatorAdapter getTaxCalculatorAdapter() {
    if (taxCalculatorAdapter == null) {
        String className = System.getProperty("taxcalculator.classname");
        taxCalculatorAdapter = (ITaxCalculatorAdapter) Class.forName(className).newInstance();
    }
    return taxCalculatorAdapter;
}

- What design pattern(s) do you see here?

Java reflection: creates a brand new object from String className!

Exercise

- Who creates the ServiceFactory?
- How is it accessed?
- We need a single instance of the ServiceFactory class
- What pattern solves these problems?

The Solution: Singleton

In UML, - means private, + means public. All (shown) fields in ServiceFactory are private and all methods are public. underline means static. instance and getInstance are static. Single instance of ServiceFactory ensures single instance of adapter objects.