

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

2

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

Fall 15 CSCI 2600, A Milanova

3

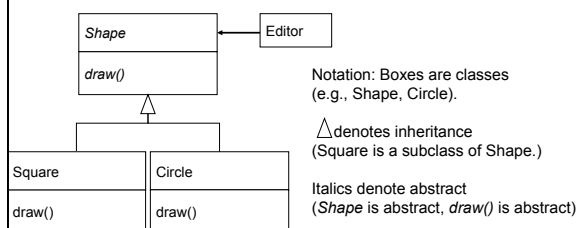
Aside: UML Class Diagrams

- 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

Fall 15 CSCI 2600, A Milanova

4

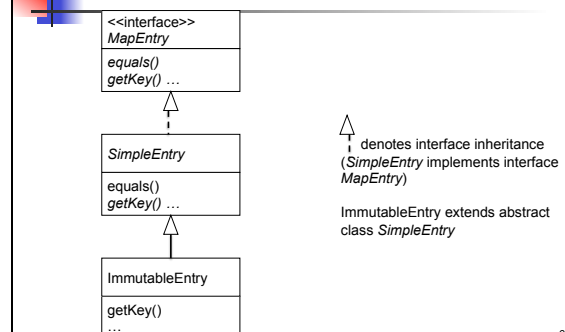
Classes and Inheritance



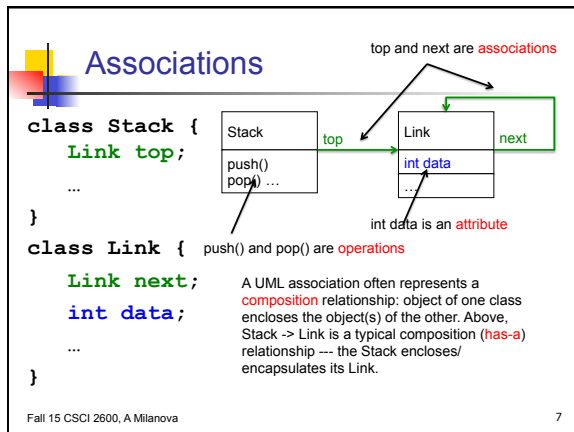
Fall 15 CSCI 2600, A Milanova

5

Classes and Inheritance



6



Exercise

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

Fall 15 CSCI 2600, A Milanova 8

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

9

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

Fall 15 CSCI 2600, A Milanova 10

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**

Fall 15 CSCI 2600, A Milanova 11

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

Fall 15 CSCI 2600, A Milanova 12

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)

Fall 15 CSCI 2600, A Milanova

13

Creational Patterns

- Problem: constructors in Java (and other OO languages) are inflexible
 - Can't return a **subtype** of the type they belong to
 - 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

14

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(); ...
    }
}
```

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

15

Using a Factory Method

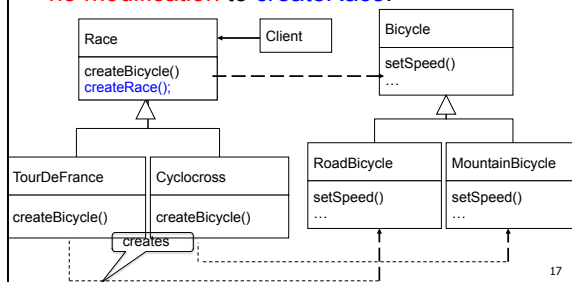
```
class Race {
    Bicycle createBicycle() { ... }
    Race createRace() {
        Bicycle bike1 = this.createBicycle();
        Bicycle bike2 = this.createBicycle(); ...
    }
}
class TourDeFrance extends Race {
    Bicycle createBicycle() {
        return new RoadBicycle();
    }
}
class Cyclocross extends Race {
    Bicycle createBicycle() {
        return new MountainBicycle();
    }
}
```

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

16

Parallel Hierarchies

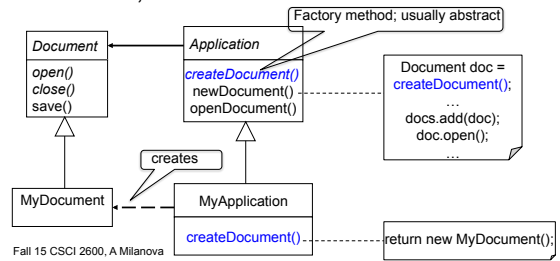
- Can **extend** with new Races and Bikes with **no modification** to **createRace**!



17

Another Factory Method Example

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



Fall 15 CSCI 2600, A Milanova

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);
        ...
    }
    ...
}
```

Fall 15 CSCI 2600, A. Milanova

19

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
```

Fall 15 CSCI 2600, A. Milanova

20

Yet Another Factory Method Example

```
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
```

Fall 15 CSCI 2600, A. Milanova

21

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?

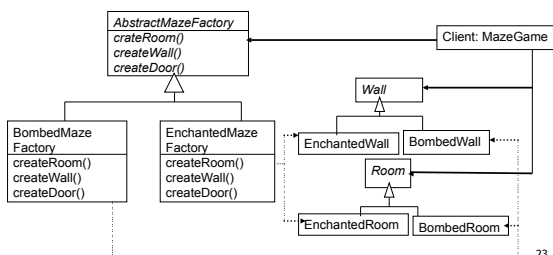
```
DateFormat df1 = DateFormat.getDateInstance();
DateFormat df2 = DateFormat.getTimeInstance();
DateFormat df3 = DateFormat.getDateInstance(
    (DateFormat.FULL.Locale.FRANCE));
Date today = new Date();
df1.format(today); // "Jul 4, 1776"
df2.format(today); // "10:15:00 AM"
df3.format(today); // "jeudi 4 juillet 1776"
```

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

22

Factory Object Pattern (also known as Abstract Factory)

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



23

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);
        ...
    }
}
```

Fall 15 CSCI 2600, A. Milanova

24

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
}
```

25

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 {
    // constructor
    TourDeFrance() {
        bfactory = new RoadBikeFactory()
    }
}

// analogous constructor for Cyclocross
```

26

The Factory Hierarchy

```
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();
    }
}
```

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

27

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**

28

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!

29

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,

```
TricycleFactory
```

Fall 15 CSCI 2600, A Milanova (modified from a slide by Michael Ernst)

30

The Prototype Pattern

- Every object itself is a factory
- Each class can define a `clone` method that returns a *copy* of the receiver object

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

- Often `Object` is the return type of `clone`
 - `Object` declares `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)

31

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());
```

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

32

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

33

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

34

Singleton Pattern

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

```
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

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

35

Another Singleton Example

```
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

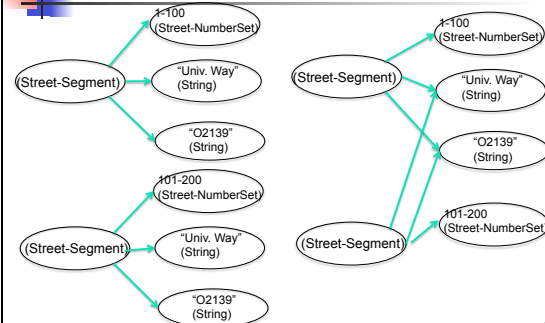
Interning Pattern

- 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

Fall 15 CSCI 2600, A Milanova

37

Interning Pattern



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

38

Interning Pattern

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

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

```
- Why not a HashSet but HashMap?
- Java supports interning for Strings:

`s.intern()` returns a canonical representation of `s`

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

39

## Why Not HashSet?

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

```

HashSet<String> names;
String canonicalName(String n) {
 if (names.contains(n))
 return n;
 else {
 names.add(n);
 return n;
 }
}

```

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

40

## What's wrong with java.lang.Boolean?

```

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

Fall 15 CSCI 2600, A Milanova

41

## 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.

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

42

## 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

43

## Wrappers

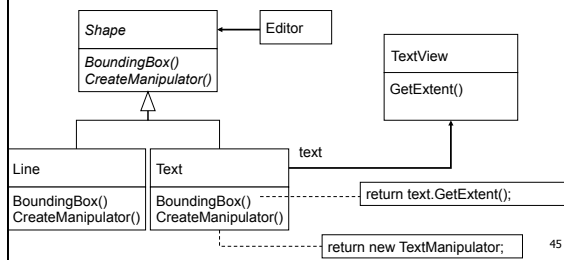
- A wrapper uses composition/delegation
- A wrapper is a thin layer over an encapsulated class
  - Modify the interface
  - Extend behavior
  - Restrict access to encapsulated object
- The encapsulated object (delegate) does most work
- **Adapter**: modifies interface, same functionality
- Decorator: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

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

44

## Adapter Pattern

- Motivation: reuse a class with an interface different than the class' interface



45

## 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

Fall 15 CSCI 2600, A Milanova

46

## Adapter Example: Scaling Rectangles

```

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!
}

```

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

47

## Class Adapter

- Class adapter adapts via subclassing

```

class ScalableRectangle1
 extends NonScalableRectangle
 implements Rectangle {
 void scale(int factor) {
 setWidth(factor*getWidth());
 setHeight(factor*getHeight());
 }
}

```

Fall 15 CSCI 2600, A Milanova

48

## Object Adapter

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

```
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(); }
}
```

Fall 15 CSCI 2600, A. Milanova

49

## 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

50

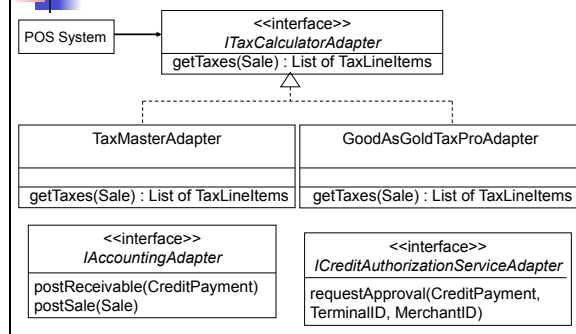
## Exercise

- 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?

Fall 15 CSCI 2600, A. Milanova

51

## The Solution: Object Adapter



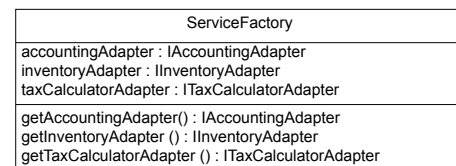
## Exercise

- 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 assign 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?

Fall 15 CSCI 2600, A. Milanova

53

## The Solution: Factory



54

## 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

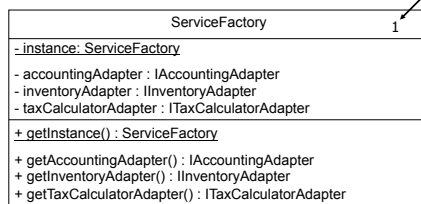
- Who creates the **ServiceFactory**?
- How is it accessed?
- We need a single instance of the **ServiceFactory** class
- What pattern solves these problems?

Fall 15 CSCI 2600, A Milano

56

## The Solution: Singleton

Special UML notation.



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.