

Design Patterns, cont.

Based on material by Michael Ernst,
University of Washington

Announcements

- HW6 due today
- HW7 out tonight
 - Reuse code from previous homeworks
 - Design for extensibility
- Grades and feedback on HW0-5 available in Homework Server
- Grades on Exam1-2, Quiz1-7 in LMS

2

Outline of today's class

- Design patterns
- Creational patterns, recap
 - Subtypes: Factory method, Factory object, Prototype
 - Sharing: Singleton and Interning
- Structural patterns
 - Adapter, Composite, Decorator, Proxy
- Behavioral patterns
 - Interpreter, Procedural and Visitor

Fall 15 CSCI 2600, A Milanova

3

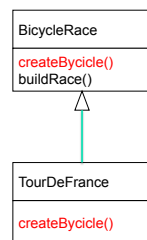
Creational Patterns

- Problem: constructors in Java (and other OO languages) are inflexible
 1. Can't return a subtype of the type they belong to
 - Factory patterns give a solution: Factory method (e.g. `createBicycle()`), Factory object, Prototype
 2. Always return a **fresh new object**, can't reuse
 - "Sharing" creational patterns present a solution to the second problem: Singleton, Interning

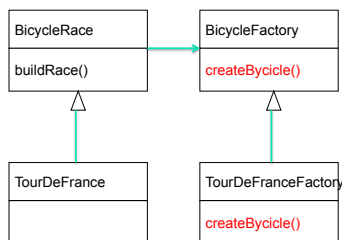
4

Factories

Factory Method



Factory Object



Fall 15 CSCI 2600, A Milanova

5

Sharing

- 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

6

Outline of today's class

- Design patterns
- Creational patterns, recap
 - Subtypes: Factory method, Factory object, Prototype
 - Sharing: Singleton and Interning
- **Structural patterns**
 - Adapter, Composite, Decorator, Proxy
- Behavioral patterns
 - Interpreter, Procedural and Visitor

Fall 15 CSCI 2600, A Milanova

7

Wrappers

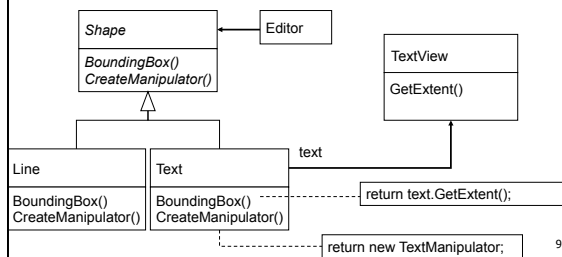
- A wrapper uses composition/delegation
- A wrapper is a thin layer over an encapsulated object
 - Modify the interface
 - Extend behavior
 - Restrict access to encapsulated object
- The encapsulated object (delegate) does most work
- **Adapter**: modifies interface, same functionality
- Decorator: same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

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

8

Adapter Pattern

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



9

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)

10

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

11

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

12

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

13

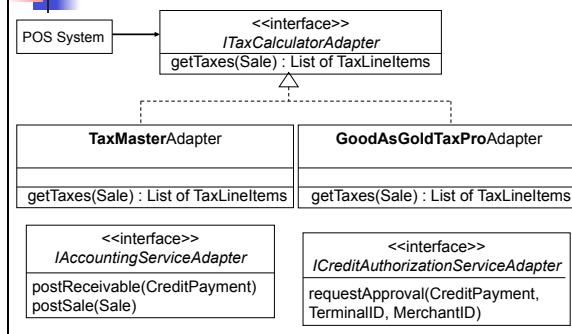
Exercise

- A Point-of-Sale system needs to support various services provided by third-party vendors:
 - **Tax calculator** service from some vendor
 - **Credit authorization** service from some vendor
 - **Inventory systems** from some vendor
 - **Accounting systems** from some vendor
- We anticipate to change vendors
 - Each vendor service has its own API, which can't be changed
- What design pattern helps address this problem?

Fall 15 CSCI 2600, A Milanova

14

The Solution: Object Adapter



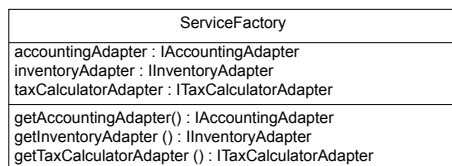
Exercise

- 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 address this problem?

Fall 15 CSCI 2600, A Milanova

16

The Solution: Factory



17

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

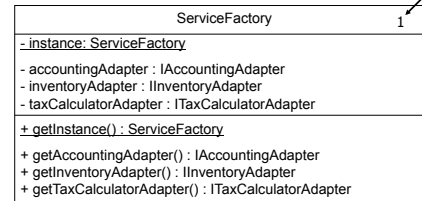
- Who creates the **ServiceFactory**?
- How is it accessed?
- We need a single instance of the **ServiceFactory**
- What pattern addresses these problems?

Fall 15 CSCI 2600, A Milanova

19

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.

Outline of today's class

- Design patterns
- Creational patterns, recap
 - Subtypes: Factory method, Factory object, Prototype
 - Sharing: Singleton and Interning
- Structural patterns
 - Adapter, **Composite**, Decorator, Proxy
- Behavioral patterns
 - Interpreter, Procedural and Visitor

Fall 15 CSCI 2600, A Milanova

21

Composite Pattern

- Client treats a **composite** object (a **collection** of objects) the **same** as a simple object (an **atomic** unit)
- Good for part-whole relationships
 - Can represent arbitrarily complex objects

Fall 15 CSCI 2600, A Milanova

22

Example: Bicycle

- Bicycle
 - Wheel
 - Skewer
 - Lever
 - Body
 - Cam
 - Rod
 - Acorn nut
 - Hub
 - Spokes
 - ...
 - Frame
 - ...

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

23

Example: Methods on Components

```

abstract class BicycleComponent {
    ...
    float cost();
}
class Skewer extends BicycleComponent {
    float price;
    float cost() { return price; }
}
class Wheel extends BicycleComponent {
    float assemblyCost;
    Skewer skewer;
    Hub hub;
    ...
    float cost() { return assemblyCost+
        skewer.cost()+hub.cost()+... }
}
  
```

Skewer is an atomic unit

Wheel is a collection of objects

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

24

Even Better

```
abstract class BicycleComponent {
    ...
    float cost();
}
class Skewer extends BicycleComponent {
    float price;
    float cost() { return price; }
}
class Wheel extends BicycleComponent {
    float assemblyCost;
    BicycleComponent skewer;
    BicycleComponent hub;
    ...
    float cost() { return assemblyCost+
        skewer.cost()+hub.cost()+... }
}
```

The skewer and hub are BicycleComponents, so we can use BicycleComponent!

Fall 15 CSCI 2600, A. Milanova

25

Another Example: Boolean Expressions

- A boolean expression can be
 - Variable (e.g., x)
 - Boolean constant: `true`, `false`
 - Or expression (e.g., x `or` `true`)
 - And expression (e.g., $(x$ `or` `true`) `and` y)
 - Not expression (e.g., `not` x , `not` $(x$ `or` $y)$)
- And, Or, Not: **collections of expressions**
- Variable and Constant: **atomic units**

Fall 15 CSCI 2600, A. Milanova

26

Using Composite to Represent Boolean Expressions

```
abstract class BooleanExp {
    abstract boolean eval(Context c);
}
class Constant extends BooleanExp {
    private boolean const;
    Constant(boolean const) { this.const=const; }
    boolean eval(Context c) { return const; }
}
class VarExp extends BooleanExp {
    String varname;
    VarExp(String var) { this.varname = var; }
    boolean eval(Context c) {
        return c.lookup(varname);
    }
}
```

Fall 15 CSCI 2600, A. Milanova

27

Using Composite to Represent Boolean Expressions

```
class AndExp extends BooleanExp {
    private BooleanExp leftExp;
    private BooleanExp rightExp;
    AndExp(BooleanExp left, BooleanExp right) {
        leftExp = left;
        rightExp = right;
    }
    boolean eval(Context c) {
        return leftExp.eval(c) && rightExp.eval(c);
    }
}

// analogous definitions for OrExp and NotExp
```

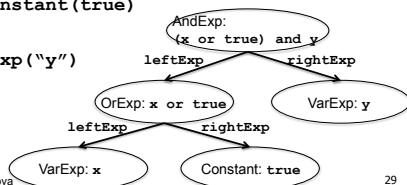
Fall 15 CSCI 2600, A. Milanova

28

Object Structure versus Class Diagram

Expression $(x$ `or` `true`) `and` y

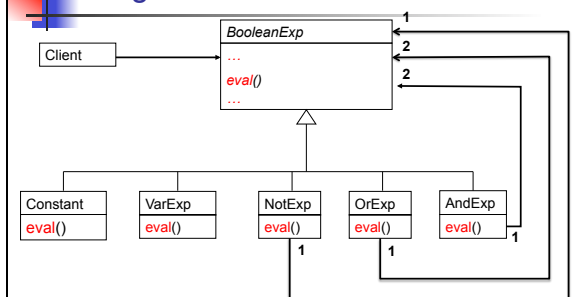
```
new AndExp(
    new OrExp(
        new VarExp("x"),
        new Constant(true)
    ),
    new VarExp("y")
)
```



Fall 15 CSCI 2600, A. Milanova

29

Object Structure vs. Class Diagram



Fall 15 CSCI 2600, A. Milanova

30

Decorator Pattern

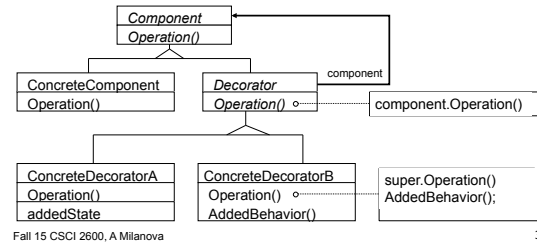
- Another wrapper pattern
 - Adapter is a wrapper, Composite is NOT
- Add functionality without changing the interface
- Add to existing method to do something in addition (while still preserving the original specification)

Fall 15 CSCI 2600, A Milanova

31

Structure of Decorator

- Motivation: add small chunks of functionality without changing the interface



Fall 15 CSCI 2600, A Milanova

32

Example

```

abstract class Component { void draw(); }
class TextView extends Component {
    public void draw() {
        // Draw the TextView
    }
}
abstract class Decorator extends Component {
    private Component component;
    public Decorator(Component c) {
        this.component = c;
    }
    public void draw() {
        component.draw();
    }
}
    
```

33

Example

```

class BorderDecorator extends Decorator {
    public BorderDecorator(Component c,
        int borderwidth) {
        super(c);
    }
    ...
    private void drawBorder() { ... }
    public void draw() {
        super.draw();
        drawBorder();
    }
}
class ScrollDecorator extends Decorator {
    ...
}
    
```

Adds a border to the text view

Adds a scroll bar to the text view

34

Example

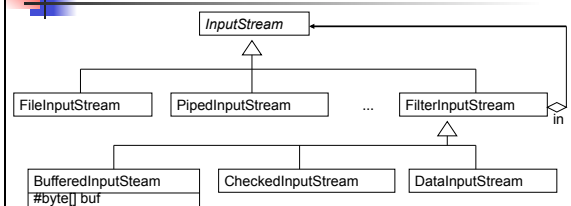
```

public class Client {
    public static void main(String[] args) {
        TextView textView = new TextView();
        Component decoratedComponent =
            new BorderDecorator(
                new ScrollDecorator(textView), 1);
        ...
        decoratedComponent.draw();
        ...
    }
}
    
```

Fall 15 CSCI 2600, A Milanova

35

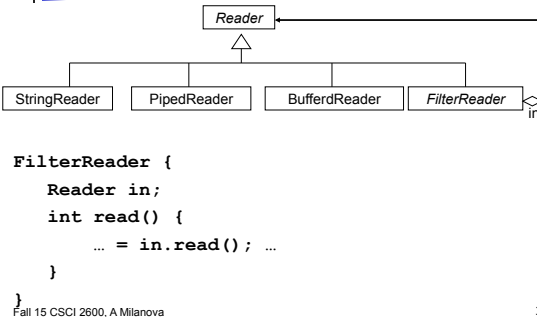
Java I/O Package



- FilterInputStream is a Decorator. Enables the “chaining” of streams
- Each FilterInputStream redirects input action to the enclosed InputStream

36

Readers: character input streams



Another Decorator Example

```

public class UppercaseConverter extends FilterReader {
    public UppercaseConverter(Reader in) {
        super(in);
    }
    public int read() throws IOException {
        int c = super.read();
        return ( c == -1 ? c :
                Character.toUpperCase((char)c) );
    }
}
// Analogous LowercaseConverter extends FilterReader

```

38

Another Decorator Example

```

public static void main(String[] args) {
    Reader f =
        new UppercaseConverter(
            new LowercaseConverter(
                new StringReader(args[0])));
    int c;
    while ((c = f.read()) != -1)
        System.out.print((char)c);
    System.out.println();
}

```

What does this code do?

39

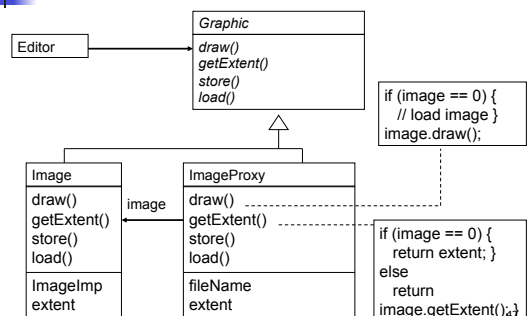
A Decorator Can Remove Functionality

- Remove functionality without changing the interface
 - Example: UnmodifiableList
 - What does it do about methods like add and put?
 - Problem: UnmodifiableList is a Java subtype of List but not a true subtype of List!
 - Decoration can create a class with no Java subtyping
- 40

Proxy Pattern

- Same interface and functionality as the enclosed class
 - Control access to enclosed object
 - Communication: manage network details when using a remote object
 - Locking: serialize access by multiple clients
 - Security: permit access only if proper credentials
 - Creation: object might not yet exist (creation is expensive). Hide latency when creating object. Avoid work if object never used
- Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst) 41

Proxy Example: manage creation of expensive object



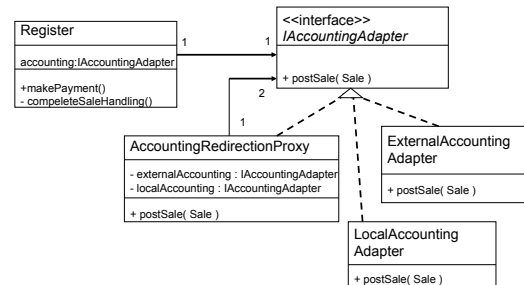
Proxy Example: manage details when dealing with remote object

- Recovery from remote service failure in the Point-Of-Sale system
 - When **postSale** is sent to an accounting service (an AccountingAdapter), if connection cannot be established, failover to a local service
 - Failover should be transparent to Register
 - I.e., it should not know whether **postSale** was sent to the accounting service or to some special object that will redirect to a local service in case of failure

Fall 15 CSCI 2600, A. Milanova

43

Proxy Example: manage details when dealing with remote object



Fall 15 CSCI 2600, A. Milanova

44

Traversing Composites

- Question: How to perform operations on all parts of a composite?
 - E.g., evaluate a boolean expression, print a boolean expression

Fall 15 CSCI 2600, A. Milanova

45

Design Patterns Summary so Far

- Factory method, Factory class, Prototype**
 - Creational patterns: address problem that constructors can't return subtypes
- Singleton, Interning**
 - Creational patterns: address problem that constructors always return a new instance of class
- Wrappers: Adapter, Decorator, Proxy**
 - Structural patterns: when we want to change interface or functionality of an existing class, or restrict access to an object

Fall 15 CSCI 2600, A. Milanova

46

Design Patterns Summary so Far

- Composite**
 - A structural pattern: expresses whole-part structures, gives uniform interface to client
- Next class: Interpreter, Procedural, Visitor**
 - Behavioral patterns: address the problem of how to traverse composite structures

Fall 15 CSCI 2600, A. Milanova

47