Design Patterns, cont.

Based on material by Michael Ernst, University of Washington

Announcements

- HW6 due today
- HW7 out tomorrow at 2pm
  - Reuse code from previous homework
  - Design for extensibility

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

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
     “Sharing” creational patterns present a solution to the second problem: Singleton, Interning

Factories

- Factory Method
  - BicycleRace
  - createBycicle()
  - buildRace()
  - TourDeFrance
  - createBycicle()

- Factory Object
  - BicycleRace
  - createBycicle()
  - BicycleFactory
  - createBycicle()
  - TourDeFrance
  - createBycicle()
  - TourDeFranceFactory
  - createBycicle()
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

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

Adapter Pattern

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

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

class Client {
    void clientMethod(Rectangle r) {
        ... r.scale(2);
    }
}

class NonScalableRectangle {
    void setWidth(); ...
    // no scale method!
}
```

```
Adapter Example: Scaling Rectangles
```

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

class Client {
    void clientMethod(Rectangle r) {
        ... r.scale(2);
    }
}

class NonScalableRectangle {
    void setWidth(); ...
    // no scale method!
}
```

```
Adapter Pattern
```

```
Adapter Example: Scaling Rectangles
```

```
Adapter Example: Scaling Rectangles
```

```
Adapter Example: Scaling Rectangles
```

```
Object Adapter
```

```
Object Adapter
```

```
Object Adapter
```

```
Object Adapter
```

```
Object Adapter
```

```
Object Adapter
```

```
Object Adapter
```

```
Object Adapter
```
**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

**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?

**The Solution: Object Adapter**

<table>
<thead>
<tr>
<th>POS System</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITaxCalculatorAdapter</td>
</tr>
<tr>
<td>getTaxes(Sale) : List of TaxLineItems</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TaxMasterAdapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>getTaxes(Sale) : List of TaxLineItems</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GoodAsGoldTaxProAdapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>getTaxes(Sale) : List of TaxLineItems</td>
</tr>
</tbody>
</table>

**Exercise**

- Who creates the appropriate adapter object?
  - Is it a good idea to let some domain object from Point-Of-Sale system (e.g., Register, Sale) create the adapters?
    - That would assign responsibility beyond the 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?

**The Solution: Factory**

```java
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?

The Solution: Singleton

ServiceFactory instance:
- accountingAdapter: IAccountingAdapter
- inventoryAdapter: IInventoryAdapter
- taxCalculatorAdapter: ITaxCalculatorAdapter

getInstance(): ServiceFactory
getAccountingAdapter(): IAccountingAdapter
getInventoryAdapter(): IInventoryAdapter
getTaxCalculatorAdapter(): ITaxCalculatorAdapter

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

Composite Pattern

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

Example: Bicycle

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

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() + … }
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!

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

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);
    }
}  

AndExp:
(x or true) and y

OrExp:
x or true

VarExp:
y

VarExp:
x

Object Structure versus Class Diagram

Expression (x or true) and y
new AndExp(
    new OrExp(
        new VarExp("x"),
        new Constant(true)
    ),
    new VarExp("y")
)
Exercise: Object Structure

- Draw the object structure (a tree!) for expression \( x \) and \( true \) and \( (y \ or \ z) \)

Structure of Composite

- Client
  - Component
    - operation()
    - add(Component)
    - remove(Component)
    - getChild(int)
  - Leaf
    - operation()
  - Composite
    - operation()
    - add(Component)
    - remove(Component)
    - getChild(int)

Another Option: Add Operations to Manage Composites

- Component
  - operation()
  - add(Component)
  - remove(Component)
  - getChild(int)
- Leaf
  - operation()
- Composite
  - operation()
  - add(Component)
  - remove(Component)
  - getChild(int)

Wrapper Patterns

- 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

Decorator Pattern

- Another wrapper pattern
  - Adapter is a wrapper, Composite is not
- Add functionality without changing the interface
- Problem, i.e., when to use: Add to existing method to do something in addition (while preserving the original specification and interface)

Structure of Decorator

- Component
  - operation()
- ConcreteComponent
  - Operation()
- Decorator
  - Operation()
  - component
  - super.Operation() AddedBehavior();
- ConcreteDecoratorA
  - Operation()
  - addedState
- ConcreteDecoratorB
  - Operation()

Spring 17 CSCI 2600, A Milanova
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();
    }
}

Example

processor 38

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();
    }
}

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 {
    ...
}

Example

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

Java I/O Package

InputStream: byte input streams

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

Readers: character input streams

FilterReader extends Reader {
    Reader in;
    int read() {
        return in.read();
    }
}

Another Decorator Example

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

We also have LowercaseConverter extends
FilterReader, which (surprise!) converts to lowercase
Another Decorator Example

```java
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 is the object structure of `f`?
What does this code do?

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

Proxy Example: Manage Creation of Expensive Object

```java
Editor
Graphic
    draw() 
    getExtent() 
    store() 
    load() 
Image
    draw() 
    getExtent() 
    store() 
    load() 
ImageImpl
    extent
    draw() 
    getExtent()
```

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 (remember, 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

Proxy Example: Manage Details When Dealing with Remote Object

```java
Register
    AccountingAdapter
        + signPayment()
        + completeSaleHandling()

AccountingRedirectionProxy
    + externalAccounting: AccountingAdapter
    + - postSale(Sale)
    + localAccounting: AccountingAdapter
        + postSale(Sale)

ExternalAccountingAdapter
    + postSale(Sale)

LocalAccountingAdapter
    + postSale(Sale)
```

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
Traversing Composites

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

Perform Operations on boolean expressions

- Need to write code for each Operation/Object pair
  
<table>
<thead>
<tr>
<th>Objects</th>
<th>VarExp</th>
<th>Constant</th>
<th>AndExp</th>
<th>OrExp</th>
<th>NotExp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td>evaluate</td>
<td></td>
<td>pretty-print</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Question: do we group together (in a class) the code for a particular operation or the code for a particular object

Interpreter and Procedural Patterns

- Interpreter: groups code per object, spreads apart code for similar operations
- Procedural: groups code per operation, spreads apart code for similar objects

<table>
<thead>
<tr>
<th>VarExp</th>
<th>AndExp</th>
<th>NotExp</th>
<th>OrExp</th>
</tr>
</thead>
<tbody>
<tr>
<td>evaluate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pretty-print</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VarExp</th>
<th>AndExp</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>pretty-print</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BooleanExp</th>
<th>VarExp</th>
<th>AndExp</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract class BooleanExp {</td>
<td>abstract boolean eval(Context c) {</td>
<td>abstract String prettyPrint() {</td>
</tr>
</tbody>
</table>
| } | } | }

<table>
<thead>
<tr>
<th>VarExp</th>
<th>AndExp</th>
<th>NotExp</th>
<th>OrExp</th>
</tr>
</thead>
<tbody>
<tr>
<td>evaluate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pretty-print</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VarExp</th>
<th>AndExp</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>pretty-print</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Procedural pattern

// Classes for expressions don’t have eval
class Evaluate {
    boolean evalConstExp(Constant c) {
        c.value(); // returns value of constant
    }
    boolean evalAndExp(AndExp e) {
        BooleanExp leftExp = e.leftExp;
        BooleanExp rightExp = e.rightExp;
        // Problem: How to invoke the right
        // implementation for leftExp and rightExp?
    }
}
Procedural pattern

```java
// Classes for expressions don’t have eval

class Evaluate {
    Context c;
    ...
    boolean evalExp(BooleanExp e) {
        if (e instanceof VarExp)
            return evalVarExp((VarExp) e);
        else if (e instanceof Constant)
            return evalConstExp((VarExp) e);
        else if (e instanceof OrExp)
            return evalOrExp((OrExp) e);
        else ...
    }
}
```

What is the problem with this code?

Visitor Pattern, a variant of the Procedural pattern

- Visitor helps traverse a hierarchical structure
- Nodes (objects in the hierarchy) accept visitors
- Visitors visit nodes (objects)

```java
class SomeBooleanExp extends BooleanExp {
    void accept(Visitor v) {
        for each child of this node {
            child.accept(v);
        }
        v.visit(this);
    }
}
class Visitor {
    void visit(SomeBooleanExp e) { do work on e }
}
```

The Interpreter Pattern

```java
class VarExp extends BooleanExp {
    void accept(Visitor v) { v.visit(this); }
}
class AndExp extends BooleanExp {
    BooleanExp leftExp;
    BooleanExp rightExp;
    void accept(Visitor v) { leftExp.accept(v); rightExp.accept(v); v.visit(this); }
}
class Evaluate implements Visitor {
    void visit(VarExp e) {
        //evaluate var exp
    }
    void visit(AndExp e) {
        //evaluate And exp
    }
}
class PrettyPrint implements Visitor {
    ...
}
```

The Visitor Pattern

```java
Visitor
accept(Constant e)
accept(VarExp e)
accept(NotExp e)
accept(OrExp e)
accept(AndExp e)
```

```java
EvaluateVisitor
accept(Constant e)
accept(VarExp e)
accept(NotExp e)
accept(OrExp e)
accept(AndExp e)
```
Visitor Pattern

- Must add definitions of visit (in Visitor hierarchy) and accept (in Object hierarchy)
- Visit may do many different things: evaluate, count nodes, pretty print, etc.
- It is easy to add operations (a new Visitor class), hard to add nodes (must modify entire hierarchy of Visitors)
- Visitors are similar to iterators but different because they have knowledge of structure not just sequence

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

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