Announcements

- HW7, Due Friday November 20
- Grades and feedback for HW0-5 in Homework Server
- Exam1-2, Quiz1-7 in LMS
- Rainbow Grades coming up!
- Quiz 8

Design Patterns So Far

- Creational patterns: Factories, Prototype, Singleton, Interning
  - Problem: constructors in Java (and other OO languages) are inflexible
    1. Can’t return a subtype of the type they belong to. “Factory” patterns address the issue: Factory method (e.g. createBicycle()), Factory class/object, Prototype
    2. Always return a fresh new object, can’t reuse. “Sharing” patterns address the issue: Singleton, Interning

Design Patterns Summary so Far

- 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

Outline of Today’s Class

- Behavioral patterns
  - Observer
  - Façade
  - Dependences and coupling

- Behavioral patterns for traversing composites
  - Interpreter
  - Procedural
  - Visitor

Observer Pattern

- Question: how to handle an object (model), which has many “observers” (views) that need to be notified and updated when the object changes state

- For example, an interface toolkit with various presentation formats (spreadsheet, bar chart, pie chart). When application data, e.g., stocks data (model) changes, all presentations (views) should change accordingly
**A Naïve Design**
- Client stores information in Data
- Then Data updates the views accordingly

Problem: to add a view, or change a view, we must change Data. Better to insulate Data from changes to Views!

**Class Diagram**

Client is responsible for creation:
```java
Data data = new Data();
data.attach(new BarChartView());
```

Data keeps list of Views, notifies them when change. Data is minimally connected to Views!

**Push vs. Pull Model**
- Question: How does the object (Data in our case) know what info each observer (View) needs?
  - Push model: Object sends the info to Observers
  - Pull model: Object does not send info directly. It gives access to itself to the Observers and lets each Observer extract the data they need

**Observer Pattern**

Pull model: observers have access to Data, they can pull the info they need.
Example of Observer

```java
class SaleItem extends Observable {
    private String name;
    private float price;
    public SaleItem(String name, float price) {
        this.name = name;
        this.price = price;
    }
    public void setName(String name) {
        this.name = name;
        notifyObservers(name);
    }
    public void setPrice(float price) {
        // analogous to setName
    }
}
```

From JDK

THE MODEL

An Observer of Name Changes

```java
class NameObserver implements Observer {
    private String name;
    public void update(Observable obj, Object arg) {
        if (arg instanceof String) {
            name = (String)arg;
            System.out.println("NameObserver:
                                 Name changed to " + name);
        } else
            System.out.println("NameObserver:
                                 Some other change to observable!");
    }
}
```

From JDK

THE VIEW

THE CONTROLLER

An Observer of Price Changes

```java
class PriceObserver implements Observer {
    private Float price;
    public void update(Observable obj, Object arg) {
        if (arg instanceof Float) {
            price = (Float)arg;
            System.out.println("PriceObserver:
                                 Price changed to " + price);
        } else
            System.out.println("PriceObserver:
                                 Some other change to observable!");
    }
}
```

THE VIEW

THE CONTROLLER

THE MODEL

Another Example

- An application that computes a path on a map and displays the path. When user requests different path, display changes
- Initially, application displays using a simple text-based UI
- Therefore, a text-based View (i.e., Observer)
- Later, application will display using a GUI interface
- A GUI-based View (another Observer)

Fall 15 CSCI 2600, A Milanova

Another Example of Observer

```java
public class SignupSheet extends Observable {
    private List<String> students = new ArrayList<String>();
    public void addStudent(String student) {
        students.add(student);
        notifyObservers();
    }
    public int size() {
        return students.size();
    }
}
```

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

```java
public class SignupObserver extends Observer {
    // called from notifyObservers, which
    // was called when SignupSheet changed
    public void update(Observable o, Object arg) {
        System.out.println("Signup count: "+ ((SignupSheet)o).size());
    }
}
```

The SignupSheet observable was sent when notifyObservers called update(this, …)

We don't care for arg now.

THE VIEW

The Client

```java
SignupSheet s = new SignupSheet();
s.addStudent("Ana");
// nothing visible happens. Why?
s.addObserver(new SignupObserver());
s.addStudent("Katarina");
// what happens now?
```

What model's used here? Push model or pull model?

THE CONTROLLER

Model-view Principle

- Observer pattern known as Model-view or Model-view-controller
- “Model” objects (e.g., Sale, SignupSheet) should not know about concrete “view” objects (e.g., SaleTotalFrame, SignupObserver)
- Domain layer should be minimally connected with presentation layer
- Open/closed principle: if user decides to change/upgrade interface, the change shall trigger no modification to domain layer

Façade Pattern

- Question: how to handle the case, when we need a subset of the functionality of a powerful, extensive and complex library
- Example: We want to perform secure file copies to a server. There is a powerful and complex general purpose security library. What is the best way to interact with this library?

Façade Pattern

Build a Façade to the library, to hide its (mostly irrelevant) complexity. SecureCopy is the Façade.
Façade Pattern

- Façade reduces interactions between client and the complex library
- Façade hides (mostly irrelevant) complexity of the library
- If library changes, we'll only need to change the Façade, the client remains insulated
  - Open/closed principle: when change happens, the change has minimal impact

Outline of Today’s Class

- Behavioral patterns
  - Observer
  - Façade
  - Dependences and coupling
- Behavioral patterns for traversing composites
  - Interpreter
  - Procedural
  - Visitor

Interactions Between Modules

- Interactions between modules (in our designs, module = class) cause complexity
- To simplify, split design into parts that don’t interact much
- Coupling is the amount of interaction among classes
  - Roughly, if class A calls methods/uses fields of class B, then there is coupling from A to B
  - In design, we strive towards low (weak) coupling, i.e., minimal, necessary interactions

Low Coupling

- Arrow means dependency. Here, A depends on C.

Observer promotes low coupling

- Bad. Data does not need depend on Views
- Better: Weaken dependency of Data on Views
  - Introduce a weaker spec in the form of interface

Coupling is the Path to the Dark Side

- Coupling leads to complexity
- Complexity leads to confusion
- Confusion leads to suffering
- If once you start down the dark path, forever will it dominate your destiny, consume you it will
Façade promotes low coupling

Façade weakens the dependency between Client and library. Introduce Façade object: reduce #dependences from 3*5 to 3+5!

Outline of Today’s Class

- Behavioral patterns
  - Observer
  - Façade
  - Dependences and coupling

- Behavioral patterns for traversing composites
  - Interpreter
  - Procedural
  - Visitor

Composite Objects

Expression (x or true) and y

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

We have a hierarchical structure: AndExp is top, OrExp and VarExp are below in the hierarchy, etc.

Composite Pattern: Class diagram

Exercise: Object Structure

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

Traversing Composites

- Question: How to perform operations on composite objects (on all parts of the component)?
  
  The Interpreter, Procedural and Visitor patterns address this question
Operations on Boolean Expressions

- Need to write code for each Operation/Object pair
- Question: do we group together (in a class) the code for a particular object or the code for a particular operation?

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

Interpreter and Procedural Patterns

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

- Interpreter Pattern
  - abstract class BooleanExp {
  - abstract boolean eval(Context c);
  - abstract String prettyPrint();
  }
  - class VarExp extends BooleanExp {
  - boolean eval(Context c) {
  - return c.lookup(varname);
  - }
  - String prettyPrint()
  }
  - class AndExp extends BooleanExp {
  - boolean eval(Context c) {
  - return leftExp.eval(c) && rightExp.eval(c); 
  - }
  - String prettyPrint()
  }
  - // analogous definitions for OrExp and NotExp

- Interpreter Pattern
  - abstract class BooleanExp {
  - 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) { varname = var; }
  - boolean eval(Context c) {
  - return c.lookup(varname);
  - }
  }
  - // analogous definitions for OrExp and NotExp
Procedural Pattern

```java
// 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?
    }
    // also, evalVarExp, evalOrExp, evalNotExp
}
```

Visitor Pattern, a Variant of the Procedural Pattern

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

Visitor 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 {
    // state, needed to evaluate
    void visit(VarExp e) {
        // evaluate Var exp
    }
    void visit(AndExp e) {
        // evaluate And exp
    }
    class PrettyPrint implements Visitor {
        ...
    }
}
```

The Visitor Pattern

```
```

The Visitor Pattern

```
```
Question

```
class VarExp extends BooleanExp {
    void accept(Visitor v) {
        v.visit(this);
    }
}

class Constant extends BooleanExp {
    void accept(Visitor v) {
        v.visit(this);
    }
}

Why not move void accept(Visitor v) up into superclass BooleanExp?
```

Exercise: Write Count Visitor which counts #subexpressions in a BooleanExp object

```
class CounterVisitor implements Visitor {
    int count = 0;

    void visit(VarExp e) {
    //??
    }

    void visit(Constant e) {
    //??
    }

    void visit(AndExp e) {
    //??
    }
    ...
}
```

Exercise: Write Evaluate Visitor which evaluates a BooleanExp object

```
class EvaluateVisitor implements Visitor {
    // ??

    void visit(VarExp e) {
        // ??
    }

    void visit(Constant e) {
        // ??
    }

    void visit(AndExp e) {
        // ??
    }
    ...
}
```

Exercise: Write a Visitor that Computes the Cost of a Bicycle Component (Note: Cost of a composite is sum of costs of components + assembly cost)

```
class Skewer extends BicycleComponent {
    void accept(Visitor v) {
        v.visit(this);
    }
}

class Wheel extends BicycleComponent {
    BicycleComponent skewer;
    BicycleComponent hub;
    ...
    void accept(Visitor v) {
        skewer.accept(v);
        hub.accept(v);
        v.visit(this);
    }
}
```

The Interpreter Pattern
The Visitor Pattern

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 (just add a new Visitor class!), but it is hard to add nodes (must modify entire hierarchy of Visitors!)
- Visitor pattern similar to Iterator but different because it has knowledge of structure, not just sequence

Design Patterns Summary so Far

- Factory method, Factory object, Prototype
  - Creational patterns: address problem that constructors can’t return subtypes
- Singleton, Interner
  - 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

Visitor Pattern

Visitor Pattern’s Double Dispatch

myExp.accept(v): we want to choose the right operation
myExp.accept(v) // dynamically dispatch the right
// implementation of accept, e.g., AndExp.accept
class AndExp {
    void accept(Visitor v) {
        // at compile-time, chooses the
        // method family: visit(AndExp). At
        // runtime, dispatches the right implementation of
        // visit(AndExp). e.g.,
        // EvaluateVisitor.visit(AndExp)