Announcements

- HW7, Due Tuesday November 21
- Grades and feedback on HW0-5 in Submitty
- A bit behind with Rainbow Grades
  - Rainbow Grades coming up tomorrow!
- Quiz 7

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.
    2. “Factory” patterns address the issue: Factory method (e.g. createBicycle()), Factory class/object, Prototype
- “Sharing” patterns address the issue: Singleton, Interning

Design Patterns 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

```
Client
    └── Data
        ├── SpreadsheetView
        └── BarChartView
```

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

A Better Design

- Data class has minimal interaction with Views
- Only needs to **update** Views when it changes

```
class Data {
    ... void updateViews() {
        spreadsheet.update(newData);
        barChart.update(newData);
        // Edit this method when different views are added.
        // Bad!
    }
}
```

```
interface Observer {
    void update(Data);
}
```

Class Diagram

```
Client
    └── Data
        ├── SpreadsheetView
        └── BarChartView
```

For (Observer obs : observers)
  obs.update(...)

Observer Pattern

```
Data
    └── observers
        ├── SpreadsheetView
        └── BarChartView
```

```
for (Observer obs : observers)
  obs.update(Data);
```

GrabStockPrice();

```
for (Observer obs : observers)
  obs.update(this);
```

Update 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

Fall 17 CSCI 2600, A Milanova (based on slide by Michael Ernst)
Example of Observer

```java
public 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;
        setChanged();
        notifyObservers(name);
    }
    public void setPrice(float price) {
        // analogous to setName
    }
}
```

An Observer of Name Changes

```java
public 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!");
    }
}
```

The View

```java
// Represent a simple text-based UI
```

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).

```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();
    }
}
```

Another Example of Observer

```java
// Represents sign-up sheet of students
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();
    }
}
```
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 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 View

```java
THE VIEW
```

The Controller

```java
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

Facade 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?

Facade Pattern

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

```java
SecureCopy
```
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 classes 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
- Bad design: classes strongly coupled
- Better design: weakly coupled

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

Observer promotes low coupling
- Bad. Data does not need depend on Views
  - Client
  - Data
  - BarChartView
  - SpreadsheetView
- Better: Weaken dependency of Data on Views
  - Introduce a weaker spec in the form of interface
  - Data
  - Observer
  - SpreadsheetView
  - BarChartView
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 \((x \text{ or } true) \text{ and } y\)
  - \(\text{new AndExp(}\)
  - \(\text{new OrExp(}\)
  - \(\text{new VarExp("x"),}\)
  - \(\text{new Constant(true)}\)
  - \(}\),
  - \(\text{new VarExp("y")}\)
  - \(\text{)}\)

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 \(x \text{ and } true \text{ and } (y \text{ or } z)\)

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?

VarExp Constant AndExp OrExp NotExp

Operations

evaluate
pretty-print

Interpreted 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

VarExp AndExp

Interpreter Pattern

abstract class BooleanExp {
  abstract boolean eval(Context c);
  abstract String prettyPrint();
}
class VarExp extends BooleanExp {
  ... boolean eval(Context c) { ... }
  String prettyPrint() { ... }
}
class AndExp extends BooleanExp {
  ... boolean eval(Context c) { ... }
  String prettyPrint() { ... }
}

Dynamic dispatch chooses right implementation at call
BooleanExp myExpr = ...
myExpr.eval(c);

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

// analogous definitions for OrExp and NotExp

Client

boolean eval()
String prettyPrint()
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?
    }
    // also, evalVarExp, evalOrExp, evalNotExp
}

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)

class SomeBooleanExp extends BooleanExp {
    void accept (Visitor v) {
        for each child of this node {
            child.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
    }
    void visit (OrExp e) {
        // evaluate Or exp
    }
}

class PrettyPrint implements Visitor {
    //...
Question

```java
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

```java
void accept(Visitor v) up into superclass BooleanExp?
```

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

```java
class EvaluateVisitor 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

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

```java
class AndExp extends BooleanExp {
    BooleanExp leftExp;
    BooleanExp rightExp;
    void accept(Visitor v) {
        leftExp.accept(v);
        rightExp.accept(v);
        v.visit(this);
    }
}
```

```java
class CountVisitor implements Visitor {
    int count = 0;
    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)

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

```java
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's Double Dispatch

Design Patterns Summary so Far