Design Patterns, cont.

Announcements

- HW7, Due Friday November 20
  - Scroll down Details if errors on instructor tests
- Grades and feedback for HW0-5 in Homework Server
- Exam1-2, Quiz1-7 in LMS
- Rainbow grades coming up

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

- Started patterns for traversal of composites: Interpreter, Procedural and Visitor

Outline of Today’s Class

- Behavioral patterns for traversing composites
  - Interpreter
  - Procedural
  - Visitor
- Behavioral pattern for traversal of containers
  - Iterator
- A Design Exercise

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
- Started patterns for traversal of composites: Interpreter, Procedural and Visitor

Composite Objects

Expression (x or true) and y

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

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

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

Interpreter and Procedural Patterns

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

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); 
    }
}

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); 
    }
}
Interpreter Pattern

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

Procedural Pattern

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 ...
    }
    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)
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 Visitor Pattern

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
        visit(AndExp e);  // evaluate And exp
    }
}

class PrettyPrint implements Visitor {
    // ...
}
Exercise: Write Count Visitor which counts subexpressions in a BooleanExp object

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 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 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 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);
    }
}

class CostVisitor implements Visitor {
    // ??
    void visit(Skewer e) {
        // ??
    }
    void visit(Hub e) {
        // ??
    }
    void visit(Wheel e) {
        // ??
    }
    ...
}
The Interpreter Pattern

The Visitor Pattern

Visitor Pattern

Visitor Pattern's Double (Dynamic) Dispatch

Iterator 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!)

**Iterator Pattern**
- Motivation: access the elements of a collection without exposing representation

```
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) {
        v.visit(this); // at compile-time, chooses the
        // method family visit(AndExp). At
    }
    // runtime, dispatches the right implementation of
    // visit(AndExp), e.g.,
    // EvaluateVisitor.visit(AndExp)
```
Iterator Pattern

- Visitor vs. Iterator?
- Visitor pattern is similar to Iterator but different because it has knowledge of structure, not just sequence

Visitor vs. Iterator?

- Visitor pattern is 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, 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

A Design Exercise

- We are building a document editor --- a large rectangle that displays a document. Document can mix text, graphical shapes, etc. Surrounding the document are the menu, scrollbars, borders, etc.
- Structure, Formatting, Embellishing the UI, User commands, Spell checking

The Composite Pattern

- Hierarchical structure --- document is made of columns, a column is made up of rows, a row is made up of words, images, etc.
- Editor should treat text and graphics uniformly. Editor should treat simple and complex elements uniformly
- What design pattern?

What's missing from this picture?
Formatting

- Formatting displays the document
- Many different formatting strategies are possible
  - We would use different formatting strategies over the same hierarchical structure
- Each formatting strategy is complex

Embellishing the UI

- We would like to embellish the document display. One embellishment adds a border around the document area, another one adds a horizontal scroll bar and a third one adds a vertical scroll bar
- We would like to do this dynamically --- one can create any combination of embellished documents
- What pattern?

User Commands

- Editor supports many user “commands”: open and close, cut and paste, etc.
- There is different user interface for the same command
  - E.g., close document through a pull-down menu item, close button, key shortcut, other
- Supports undo and redo!

The Strategy Pattern

- Encapsulates an algorithm in an object

The Decorator Pattern

- Adds functionality, preserves interface

A Naïve Design

- What’s wrong with this design?
The Command Pattern

- Separates MenuItems from Commands that do the work

```java
MenuItem onClick() {
    command.execute();
}
```

```java
... = new MenuItem("Save", new Save(document));
... = new MenuItem("Close", new Close(document));
... = new ShortcutButton("Save", new Save(document));
```

Easy to Add Undo/Redo!

- Editor maintains a history (e.g., a stack) of commands that have been executed

```java
Document open() {
    save();
    close();
}
```

Adding Spell Checking

- Requires traversal of document hierarchy
- We want to avoid writing this functionality into the document structure
- We would like to add other traversals in the future, e.g., search, word count, hyphenation
- What pattern?

The Visitor Pattern

- Glyph
  - draw
  - intersect
  - accept(Visitor)
- Document
  - draw
  - intersect
  - insert
  - accept(Visitor)
- Polygon
  - draw
  - intersect
  - accept(Visitor)
- Word
  - draw
  - intersect
  - accept(Visitor)