Decorators, Adapters and Visitors

Oct 2nd
• Design patterns capture software architectures and designs
  – Not direct code reuse!
  – Instead, solution/strategy reuse
  – Sometimes, interface reuse
• To support **reuse** of successful designs

• To facilitate **software evolution**
  – Add new features easily, without breaking existing ones

• In short, we want to **design for change**
• Reduce implementation dependencies between elements of a software system

• Sub-goals:
  – Program to an interface, not an implementation
  – Favor composition over inheritance
  – Use delegation
Program to Interface, Not Implementation

- Rely on abstract classes and interfaces to hide differences between subclasses from clients
  - Interface defines an object’s use (protocol)
  - Implementation defines particular policy

- Example: **Iterator** interface, compared to its implementation for a **LinkedList**
Rationale

• Decouples clients from the implementations of the applications they use
• When clients manipulate an interface, they remain unaware of the specific object types being used.

• Therefore: clients are less dependent on an implementation, so it can be easily changed later.
Favor Composition over Class Inheritance

- **White box reuse:**
  - Inheritance

- **Black box reuse:**
  - Composition
Delegation

- Forward messages (delegate) to different instances at run-time; a form of composition
  - May pass invoking object’s this pointer to simulate inheritance

```java
Window
  area()

Rectangle
  area()
  width
  height

return rectangle.area()
```

Return width * height
Rationale

• White-box reuse has results in implementation dependencies on the parent class
  – Reusing a subclass may require rewriting the parent
  – But inheritance easy to specify

• Black-box reuse often preferred
  – Eliminates implementation dependencies, hides information, object relationships non-static for better run-time flexibility
  – But adds run-time overhead (additional instance allocation, communication by dynamic dispatch)
  – Sometimes code harder to read and understand
• **Motivation**
  – Want to add responsibilities/capabilities to individual objects, not to an entire class
  – Inheritance requires a compile-time choice of parent class

• **Solution**
  – Enclose the component in another object that adds the responsibility/capability
    • The enclosing object is called a **decorator**.
// Slightly simplified
class LineNumberReader extends BufferedReader {
    private int lineNumber;
    public LineNumberReader(Reader in) { super(in); }
    public int getLineNumber() { return lineNumber; }

    public int read() {
        int c = super.read();
        if (c == '\n') {
            lineNumber++;
            return '\n';
        }
        return c;
    }
}

Decorator Pattern: Features

• Decorator conforms to interface of decorated component
  – Its presence is transparent to the component's clients.

• Decorator forwards requests to encapsulated component
  – May perform additional actions before or after

• Can nest decorators recursively
  – Allows unlimited added responsibilities

• Can add/remove responsibilities dynamically
Structure
Decorator Pattern Analysis

• Advantages
  – Fewer classes than with static inheritance
  – Dynamic addition/removal of decorators
  – Keeps root classes simple

• Disadvantages
  – Proliferation of run-time instances
  – Abstract Decorator must provide common interface

• Tradeoffs:
  – Useful when components are lightweight
  – Otherwise use Strategy
• Implementing a decorator for an interface with 20 methods is tedious
  – tedious work is error prone

• Two approaches that can remove some of the tedium
  – define an abstract base class that implements most methods in terms of just a few key abstract methods
    • only need to implement those abstract methods
  – define an abstract delegating base class, that delegates everything
    • override those methods you don't want to have handled by pure delegation.
Adapter (aka Wrapper) Pattern

• Problem:
  – You have some code you want to use for a program
  – You can’t incorporate the code directly (e.g., you just have the .class file, say as part of a library)
  – The code does not have the interface you want
    • Different method names
    • More or fewer methods than you need

• To use this code, you must adapt it to your situation
Here’s what we have:

- Client is already written, and it uses the Target interface
- Adaptee has a method that works, but has the wrong name

How do we enable the Client to use the Adaptee?
• Solution: adapter class to implement client’s expected interface, forwarding methods
Decorator vs. Adapter

• A decorator *adds* to the responsibilities of an object
  – Usually has the same interface plus more features

• An adapter *changes* the interface
  – But usually not the responsibilities
Visitor: Implementing Analyses

• Often want to implement multiple analyses on the same kind of object data
  – Project example: Generating code for and analyzing an Abstract Syntax Tree (AST) in a compiler

• One solution: implement each analysis as a method in each object
public interface Node { }

public class Number extends Node {
    public int n;
}

public class Plus extends Node {
    public Node left;
    public Node right;
}
public interface Node {
    public int eval();
}

public class Number extends Node {
    public int value;
    public int eval() { return value; }
}

public class Plus {
    public Node left;
    public Node right;
    public int eval()
        { return left.eval() + right.eval(); } }

Traversing Abstract Syntax Trees
Naïve approach (not a visitor)

One method for each analysis
Tradeoffs with this Approach

• Follows idea “objects are responsible for themselves”

• But many analyses will occlude the object’s main code

• Result is classes that are hard to maintain
Use a Visitor

• Alternatively, can define a separate **visitor** class
  – A visitor encapsulates the operations to be performed on an entire structure, e.g., all elements of a parse tree

• Allows operations to be separate from structure
  – But doesn’t necessarily require putting all of the structure traversal code into each visitor/operation
Double dispatch (aka visitor pattern)

• I have two type hierarchies
  – Letter, with subtypes A, B, ...
  – Number, with subtypes _1, _2, ...

• The Number class defines methods
  – visit(A a)
  – visit(B b), ...

• Implement static void visit(Letter x, Number y)
Not an example of the pattern

- Can do it with instance of checks

- static void visit(Letter x, Number y) {
  y.visit(x); // won't compile
  if (x instanceof A) y.visit((A) x);
  else if (x instanceof B) y.visit((B) x);
  ...
}

- Not recommended, fragile
interface Letter {
    void accept(Number x); ...
}
class A implements Letter {
    void accept(Number x) {
        x.visit(this); // visit(A)
    }
}
class B implements Letter {
    void accept(Number x) {
        x.visit(this); // visit(B)
    }
}
Invoking accept

- static void visit(Letter x, Number y) {
  x.accept(y);
}
Sample Visitor class

- **NodeVisitor**
  - `VisitAssignment(AssignmentNode)`
  - `VisitVariableRef(VariableRefNode)`

- **TypeCheckingVisitor**
  - `VisitAssignment(AssignmentNode)`
  - `VisitVariableRef(VariableRefNode)`

- **CodeGeneratingVisitor**
  - `VisitAssignment(AssignmentNode)`
  - `VisitVariableRef(VariableRefNode)`
How to perform traversal?

• Now that we have a visitor class, how do we apply its analysis to the objects of interest?
  – Add `accept(visitor)` method to each structure class, that will invoke the given visitor on `this`
  – Builds on Java’s dynamic dispatch
  – Use an iteration algorithm (like an Iterator) to call `accept()` on each relevant object
Sample visited objects
Visitor Interaction

\text{aNodeStructure} \text{ aAssignmentNode} \text{ aVariableRefNode} \text{ aTypeCheckingVisitor}

Accept (\text{aTypeCheckingVisitor}) \\
\text{VisitAssignment(} \text{aAssignmentNode}) \\
\text{someOperation()}

Accept (\text{aTypeCheckingVisitor}) \\
\text{VisitVariableRef (} \text{aVariableRefNode}) \\
\text{someOperation()}

\text{someOperation()}
public interface Visitor<T> {
    public T visitNumber(Number n);
    public T visitPlus(Plus p);
}

public class EvalVisitor implements Visitor<Integer> {
    public Integer visitNumber(Number n) {
        return n.value;
    }
    public Integer visitPlus(Plus p) {
        return p.left.accept(this) + p.right.accept(this);
    }
}
public interface Node {
    public void accept(Visitor v);
}

public class Number extends Node {
    ...
    public void accept(Visitor v) {v.visitNumber(this);}
}
public class Plus extends Node {
    ...
    public void accept(Visitor v) {v.visitPlus(this);}
}
Visitor pattern

• Name
  – Visitor or double dispatching

• Applicability
  – Related objects must support different operations and actual op depends on both the class and the op type
  – Distinct and unrelated operations pollute class defs
  – **Key**: object structure rarely changes, but ops changed often
Visitor Pattern Structure

- Define two class hierarchies
  - One for object structure
    - AST in compiler
  - One for each operation family, called visitors
    - One for typechecking, code generation, pretty printing in compiler
Structure of Visitor Pattern
Visitor Pattern Consequences

• Adding new operations is easy
  – Add new op subclass with method for each concrete elt class
  – Easier than modifying every element class
• Gathers related operations and separates unrelated ones
• Adding new concrete elements is difficult
  – Must add a new method to each concrete Visitor subclass
• Allows visiting across class hierarchies
  – Iterator needs a common superclass (i.e., composite pattern)
• Visitor can accumulate state rather than pass it as parameters
Double-Dispatch

• Accept code is always trivial
  – Just dynamic dispatch on argument, with runtime type of structure node taking into account in method name

• A way of doing double-dispatch
  – Traversal routine takes two arguments, the visitor and the object to traverse
    • `o.accept(aVisitor)` will dispatch on the actual identity of `o` (the object being considered)
    • ...and accept will internally dispatch on the identity of `aVisitor` (the object visiting it)
Using Overloading in a Visitor

- You can name all of the \texttt{visitXXX(XXX x)} methods just \texttt{visit(XXX x)}
  - Calls to \texttt{Visit(AssignmentNode n)} and \texttt{Visit(VariableRefNode n)} distinguished by compile-time overload resolution
Visitors Can Forward Common Behavior

• Useful for composites
  – If subclasses of a particular object all treated the same
  – Can have visit(SubClass) call visit(SuperClass)

• For example
  – visit(BinaryPlusOperatorNode) can just forward call to superclass
    visit(BinaryOperatorNode)
A visitor can contain state
- E.g., the results of typechecking the program so far

```java
class TypeCheckingVisitor extends Visitor {
    private TypeMap map;
    void visit(VariableDefNode n) {
        map.add(n,t)
    }
}
```

Or visitors pass around a separate state object
- Impacts the type of the Visitor superclass
Implementing Traversal

• Who is responsible for traversing object structure?
• Plausible answers:
  – **Visitor**
    • But, must replicate traversal code in each concrete visitor
  – **Object structure**
    • Define operation that performs traversal while applying visitor object to each component
  – **Iterator**
    • Iterator sends message to visitor with current element as arg
Traversals

• It’s sometimes preferable to try to keep traversal separate from the Visitor
  – E.g., use an Iterator
  – Thus traversal and analysis can evolve independently
• But can also do it within node or visitor class. Several solutions here:
  – acceptAndTraverse methods
    • traverse from within accept()
  – Separating processing from traversal
    • Visit/process methods
  – Traversal visitors applying an operational visitor
• Class BinaryPlusOperatorNode {
    void accept(Visitor v) {
        v.visit(this);
        lhs.accept(v);
        rhs.accept(v);
    }
    ...
}
• Accept method could be responsible for traversing children
  – Assumes all visitors have same traversal pattern
    • E.g., visit all nodes in pre-order traversal
  – Could provide previsit and postvisit methods to allow for more complicated traversal patterns
    • Still visit every node
    • Can’t do out of order traversal
    • In-order traversal requires inVisit method
Visitor/Process Methods

• Can have two parallel sets of methods in visitors
  – Visit() methods
  – Process() methods

• How it works: the visit(Node n) method:
  – Calls process(n) method of visitor
  – Calls accept() on all children of n (passing the visitor as an argument)

• Allows finer-grained subtyping of Visitor classes that include traversal
  – Subclass a visitor, and just change the process method
• Class PreorderVisitor {
    void visit(BinaryPlusOperatorNode n) {
        process(n);
        n.lhs.accept(this);
        n.rhs.accept(this);
    }
    ...
}
Visit/Process, Continued

• Can define a PreorderVisitor
  – Extend it, and just redefine process method
    • Except for the few cases where something other than preorder traversal is required

• Can define other traversal visitors as well
  – E.g., PostOrderVisitor
Traversals Visitors Applying an Operational Visitor

• Define a Preorder traversal visitor
  – Takes an operational visitor as an argument when created

• Perform preorder traversal of structure
  – At each node
    • Have node accept operational visitor
    • Have each child accept traversal visitor
• Class PreorderVisitor {
    Visitor payload;
    PreorderVisitor(Visitor p) { payload = p; }
    void visit(BinaryPlusOperatorNode n) {
        payload.visit(n);
        n.lhs.accept(this);
        n.rhs.accept(this);
    }
    ...
}