Bridge Pattern

• Problem
  – The client interface and implementation of an abstraction should be able to vary independently.
  – This is not possible with a single class hierarchy.
• Solution:
  – Create two class hierarchies, one for the logical abstraction, and the other for the implementation.
  – Create a “bridge” to bring them together, implementing classes in the abstraction hierarchy using ones in the implementation hierarchy.

Example: Multiple Window Systems

• Goal: separate windowing system structure from how it’s implemented
  – Applications depend on the window hierarchy and the operations it defines, but are shielded from separate (low-level) issues of implementation.
• Sketch of Solution:
  – Abstraction hierarchy rooted by class Window, defining window functionality.
  – Bridge to implementation hierarchy rooted by class WindowImp to handle a vendor’s implementation
Example: Multiple Window Systems

Structure of Bridge Pattern
Consequences

- Decouple abstraction from implementation and representation
  - Clients depend on the logical abstraction; shielded from implementation details which could change.
- Change implementation at run-time
  - Simply switch abstraction classes to point to a different implementation classes at runtime.

Command Pattern

- Problem
  - An application often performs similar actions that could be effected in different ways
- Solution
  - Parameterize objects by the commands they perform: encapsulate a command as an object
  - All commands inherit from same interface, but implement it differently
  - Permits reusing commands in different contexts, and allows commands to be aware of their receiver
Example: Lexi Commands

- Initiate a command from a button, or a menu item
- Allow some commands to be undone or redone
  - Cutting and pasting vs. printing
  - Requires command objects store history
- Each command might have a different context
  - Cut command does the same thing, but might be operating on different documents, and different text within those documents
- A sequence of commands can form a macro

Command Objects
Structure of Command Pattern

Consequences

- Decouple receiver and executor of requests
  - Lexi example: Different icons can be associated with the same command
  - Commands can change without affecting callers
- Easy to support undo and redo
  - command has method to check whether it’s reversible
  - must add state information
- Can create composite commands
  - Editor macros
Visitor: Implementing Analyses

- Problem: want to implement multiple analyses on the same kind of object data
  - Spellchecking and Hyphenating Glyphs
  - Generating code for and analyzing an Abstract Syntax Tree (AST) in a compiler
- Flawed solution: implement each analysis as a method in each object
  - Follows idea “objects are responsible for themselves”
  - But many analyses will occlude the object’s main code
  - Result is classes hard to maintain

Naïve approach (not a visitor)

One method for each analysis

- Node
  - TypeCheck()
  - GenerateCode()
  - PrettyPrint()
- VariableRefNode
  - TypeCheck()
  - GenerateCode()
  - PrettyPrint()
- AssignmentNode
  - TypeCheck()
  - GenerateCode()
  - PrettyPrint()
Visitor Pattern

- We define each analysis as a separate **Visitor** class
  - Defines operations for each element of a structure
- A separate algorithm traverses the structure, applying a given visitor
  - But, like iterators, objects must reveal their implementation to the visitor object
- Separates structure traversal code from operations on the structure
  - Observation: object structure rarely changes, but often want to design new algorithms for processing

Sample Visitor class

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

TypeCheckingVisitor
  VisitAssignment(AssignmentNode)
  VisitVariableRef(VariableRefNode)

CodeGeneratingVisitor
  VisitAssignment(AssignmentNode)
  VisitVariableRef(VariableRefNode)
```
Traversing a structure

- Add \texttt{accept(Visitor)} method to each structure class, that will invoke the given visitor on \texttt{this}.
  - Builds on Java's dynamic dispatch.
- Use an iteration algorithm to call \texttt{accept()} as each object is reached.

Sample visited objects
Double-dispatch

- Accept code is a way of doing *double-dispatch*
- Traversal routine takes two arguments, the visitor `aVisitor` and the object `o` 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).

Visitor Interaction

```
<table>
<thead>
<tr>
<th>aNodeStructure</th>
<th>aAssignmentNode</th>
<th>aVariableRefNode</th>
<th>aTypeCheckingVisitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept</td>
<td>VisitAssignment(assignmentNode)</td>
<td>VisitVariableRef(assignmentNode)</td>
<td>someOperation()</td>
</tr>
<tr>
<td>(aTypeCheckingVisitor)</td>
<td></td>
<td>(aVariableRefNode)</td>
<td></td>
</tr>
<tr>
<td>Accept</td>
<td></td>
<td>someOperation()</td>
<td></td>
</tr>
<tr>
<td>(aTypeCheckingVisitor)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Structure

- One class hierarchy for object structure
  - AST in compiler, Glyphs in Lexi
- One class hierarchy for each operation family, called visitors
  - One for typechecking, code generation, pretty printing in compiler
  - One for spellchecking or hyphenation in Lexi
Using overloading in a visitor

- You can name all of the visitCLASS(CLASS x) methods just visit(CLASS x)
  - E.g., change visitAssignmentNode(AssignmentNode n) and visitVariableRefNode(VariableRefNode n) to be visit(AssignmentNode n) and visit(VariableRefNode n)
- Calls distinguished by compile-time overload resolution
  - Distinguishes visit (AssignmentNode n) from visit(VariableRefNode n)

Visitor Pattern Consequences

- Gathers related operations into one class
- Adding new analyses is easy
  - New visitor for each one
  - Easier than modifying the object structure
- 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)
State in a visitor pattern

- 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(VariableRefNode n) {
        map.add(n, t)
        ...
    }
}
```

Visitor Traversal Choices

- Traversal in object structure (typical, see Liskov)
  - Define operation that performs traversal while applying visitor object to each component
- Traversal implemented in visitor itself
  - E.g., perform processing at this node, then pass visitor to children nodes.
  - Traversal code replicated in each concrete visitor
- External Iterator
Traversing in Object Structure

- **Accept()** method responsible for traversing children
  - Requires all visitors to have same traversal pattern
    - E.g., visit all nodes in pre-order traversal
  - Could provide Visitor **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

```java
interface Node { void accept(Visitor); ... }
class BinaryOperatorNode implements Node {
    Node lhs, rhs;
    void accept(Visitor v) {
        v.visit(this);
        lhs.accept(v);
        rhs.accept(v);
    }
    ...
}
```
abstract class PreorderVisitor {
    abstract void
    process(BinaryOperatorNode n);
    ...
    void visit(BinaryOperatorNode n) {
        process(n);
        n.lhs.accept(this);
        n.rhs.accept(this);
    }
    ...
}

Inorder visitor

abstract class InorderVisitor {
    abstract void
    process(BinaryOperatorNode n);
    ...
    void visit(BinaryOperatorNode n) {
        process(n);
        n.lhs.accept(this);
        n.rhs.accept(this);
    }
    ...
}
Parameterized Visitor Traversals

- Have only one `visit()` method
- Parameterize a “traversal visitor” by an “operational visitor”
  - Traversal visitor invokes `visit()` methods of operational visitor
- Replaces inheritance, as above, with composition

Parameterized Preorder visitor

class PreorderVisitor {
    Visitor operationalVisitor;
    void visit(BinaryOperatorNode n) {
        payload.visit(n);
        n.lhs.accept(this);
        n.rhs.accept(this);
    }
    ...
}

Traversing using Iterator

- Make it so that traversal independent of the implementation
  - Actual visit methods still need to see the underlying nodes, however
- Uses an iterator, rather than specifying sub-nodes directly.
- Can specify a different traversal using different iterators.

```java
abstract class Node {
    // generic visit routine
    void visit(Visitor v) {
        n.accept(v); // preorder
        Iterator i = iterator();  
        while (i.hasNext()) {
            Node n = (Node)i.next();
            n.visit(v);
        }
    }
    abstract Iterator iterator();
}
```
Traversal in Using Iterator Example

class BinaryOperatorNode extends Node {
    Node lhs, rhs; // visit still sees these
    void accept(Visitor v) {
        v.visit(this);
    }
    Iterator iterator() { ... }
    ...
}

Designing with Patterns

- How do you know which patterns to use?
- What if you choose the wrong pattern?
  - I.e. your code doesn’t evolve the way you thought it would.
- What if all your work to make things extensible via patterns never pays off?
  - I.e. your code doesn’t change in the way you thought it would.
- Choosing the right pattern implies prognostication
Designing with Patterns

- Some design patterns are immediately useful
  - Observer, Decorator in our EventProcessor stuff
- Some are not immediately useful, but you think they might be
  - You anticipate changing things later—prognostication
- Recently popular philosophy: XP
  - Design for your immediate needs
  - When those needs change, redesign your code to match
  - Use extensive testing to validate frequent changes
- Next time: refactoring