Bridge Pattern

- Name
  - Bridge or Handle or Body
- Applicability
  - handles abstract concept with different implementations
  - implementation may be switched at run-time
  - implementation changes should not affect clients
  - hide a class’s interface from clients
- Structure: use two hierarchies
  - logical one for clients,
  - physical one for different implementations

Structure of Bridge Pattern

Consequences:
- decouple interface from implementation and representation
- change implementation at run-time
- improve extensibility
  - logical classes and physical classes change independently
  - hides implementation details from clients
    - sharing implementation objects and associated reference counts

Supporting User Commands

- Support execution of Lexi commands
  - GUI doesn’t know
    - who command is sent to
    - command interface
- Complications
  - different commands have different interfaces
  - same command can be invoked in different ways
  - Undo and Redo for some, but not all, commands (print)
Supporting User Commands (cont’d)

• An improved solution
  – create abstract “command” class
  – create action-performing glyph subclass
  – delegate action to command
• Key ideas
  – pass an object, not a function
  – pass context to the command function
  – store command history

Command Objects

Command Pattern

• Name
  – Command or Action or Transaction
• Applicability
  – parameterize objects by actions they perform
  – specify, queue, and execute requests at different times
  – support undo by storing context information
  – support change log for recovery purposes
  – support high-level operations
  
  * macros

Structure of Command Pattern
Command Pattern

- Consequences:
  - decouple receiver and executor of requests
    - Lexi example: Different icons can be associated with the same command
  - commands are first class objects
  - easy to support undo and redo
    - command must have method to check whether it’s reversible
    - must add state information
  - can create composite commands
    - Editor macros
    - can extend commands more easily

- Implementation notes
  - how much should command do itself?
  - support undo and redo functionality
    - operations must be reversible
      - may need to copy command objects
      - don’t record commands that don’t change state
  - avoid error accumulation in undo process

Spell-Checking and Hyphenation

- Must do textual analysis
  - multiple operations and implementations
- Must add new functions and operations easily
- Must efficiently handle scattered information and varied implementations
  - different traversal strategies for stored information
- Should separate actions from traversal

Structure of Iterator Pattern
Visitor: Implementing Analyses

- Often 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
- One 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

Use a Visitor

- Alternatively, we can define each analysis as 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 the operations to be specified separately from the structure
  - But doesn’t require putting all of the structure traversal code into each visitor/operation
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

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, Glyphs in Lexi
  - one for each operation family, called visitors
    - One for typechecking, code generation, pretty printing in compiler
    - One for spellchecking or hyphenation in Lexi

Use of Visitor Pattern in Lexi

Visitor Pattern Consequences

- Adding new operations is easy
  - add new operation subclass with a method for each concrete element 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
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

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 both on the actual identity of o (the object being considered), and on the identity of aVisitor (the object visiting it).

Using overloading in a visitor

- You can name all of the visitXXX(XXX x) methods just visit(XXX x)
  - Calls to Visit(AssignmentNode n) and 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)
State in a visitor pattern

• A visitor can contain state
  – E.g., the results of typechecking the program so far

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

• Or visitors pass around a separate state object
  – Impacts the type of the Visitor superclass

Traversals

• It’s preferred 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
  – Separating processing from traversal
    • Visit/Process methods
    • Traversal visitors applying an operational visitor

acceptAndTraverse methods

• 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

Accept and traverse

• Class BinaryPlusOperatorNode {
    void accept(Visitor v) {
        v.visit(lthis);
        lhs.accept(v);
        rhs.accept(v);
    }
    ...
}
Visitor/process methods

- Can have two parallel sets of methods in visitors
  - Visit() methods
  - Process() methods
- Allows finer-grained subtyping of Visitor classes that include traversal
  - Subclass a visitor, and just change the process method
- How it works: the visit() method on a node:
  - Calls process() method of visitor, passing node as an argument
  - Calls accept() on all children of the node (passing the visitor as an argument)

Preorder visitor

- Class PreorderVisitor
  ```java
  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

Traversal 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
PreorderVisitor with payload

- Class PreorderVisitor {
  Visitor payload;
  void visit(BinaryPlusOperatorNode n) {
    payload.visit(n);
    n.lhs.accept(this);
    n.rhs.accept(this);
  }
  ...
}

Pattern hype

- Patterns get a lot of hype and fanatical believers
  – We are going to have a design pattern reading group, and this week we are going to discuss the Singleton Pattern!
- Patterns are sometimes wrong (e.g., double-checked locking) or inappropriate for a particular language or environment
  – Patterns developed for C++ can have very different solutions in Smalltalk or Java