State Pattern

• Problem
  – An object is always in one of several known states
  – The state an object is in determines the behavior of several methods

• Solution
  – Could use if/case statements in each method
  – Better solution: use dynamic dispatch
State Pattern Approach

- Encode different states as objects with same superclass

- To change state, change the state object

- Methods delegate to state object

Example – Finite State Machine

class FSM {
    State state;
    public FSM(State s) { state = s; }
    public void move(char c) { state = state.move(c); }
    public boolean accept() { return state.accept(); }
}

public interface State {
    State move(char c);
    boolean accept();
}
class State1 implements State {
    static State1 instance = new State1();
    private State1() {}
    public State move (char c) {
        switch (c) {
            case 'a': return State2.instance;
            case 'b': return State1.instance;
            default: throw new IllegalArgumentException();
        }
    }
    public boolean accept() {return false;}
}

class State2 implements State {
    static State2 instance = new State2();
    private State2() {}
    public State move (char c) {
        switch (c) {
            case 'a': return State1.instance;
            case 'b': return State1.instance;
            default: throw new IllegalArgumentException();
        }
    }
    public boolean accept() {return true;}
}

Structure of State Pattern
State Pattern Notes

- Can use singletons for instances of each state class
  - State objects don’t encapsulate (mutable) state, so can be shared
- Easy to add new states
  - New states can extend the base class, or
  - New states can extend other states
    - Override only selected functions
Lexi: Simple GUI-Based Editor

• Lexi is a WYSIWYG editor
  – Supports documents with textual and graphical objects
  – Scroll bars to select portions of the document
  – Be easy to port to another platform
  – Support multiple look-and-feel interfaces
• Highlights several OO design issues
• Case study of design patterns in the design of Lexi

Lexi User Interface
### Design Issues

- Representation and manipulation of document
- Formatting a document
- Adding scroll bars and borders to Lexi windows
- Support multiple look-and-feel standards
  - Motif and Presentation Manager (!)
- Handle multiple windowing systems
- Support user operations
- Advanced features
  - spell-checking and hyphenation

### Structure of a Lexi Document

- **Goals:**
  - Store text and graphics in document
  - Generate visual display
  - Maintain info about location of display elements
- **Caveats:**
  - Treat different objects uniformly
    - E.g., text, pictures, graphics
  - Treat individual objects and groups of objects uniformly
    - E.g., characters and lines of text
Structure of a Lexi Document

• Use **recursive composition** for defining and handling complex objects
  – Abstract class Glyph for all displayed objects
  – Glyph responsibilities:
    • Know how to draw itself
    • Knows what space it occupies
    • Knows its children and parent
  – Glyph instances can recursively **compose** other Glyph instances
The Composite Pattern

- **Motivation:**
  - Support recursive composition in such a way that a client need not know the difference between a single and a composite object (as with Glyphs)

- **Applicability:**
  - When dealing with hierarchically-organized objects (e.g., columns containing rows containing words ...)
Composite Pattern Structure

- Class hierarchy has both simple and composite objects
- Simplifies clients
- Aids extensibility
  - Clients do not have to be modified
- Too general a pattern?
  - Difficult to restrict functionality of concrete leaf subclasses
Formatting Lexi Documents: Strategy

• We know that documents are represented as Glyphs, but not how documents are constructed.
• Formatting:
  – Document structure will be determined based on rules for justification, margins, line breaking, etc.
  – Many good algorithms exist;
    • different tradeoffs between quality and speed
• Design decision: implement different algorithms, decide at run-time which algorithm to use
  – define root class that supports many algorithms
  – each algorithm implemented in a subclass

Strategy Pattern

• Name
  – Strategy (aka Policy)
• Applicability
  – Many related classes differ only in their behavior
  – Many different variants of an algorithm
  – Need to encapsulate algorithmic information
Strategy Pattern: Structure

- Clear separation of algorithm definition and use
  - Glyphs and formatting algorithms are independent
  - Alternative (many subclasses) is unappealing
    - Proliferation of classes
    - Algorithms cannot be changed dynamically
- Elimination of conditional statements
  - Like State, Template, …
  - Typical in OO programming

Strategy Pattern: Consequences
Strategy Pattern Consequences (cont’d)

- Clients must be aware of different strategies
  - When initializing objects
- Proliferation of instances at run-time
  - Each Glyph has a strategy object with formatting information
  - If strategy is stateless, share strategy objects

Lexi: Using Strategy

- Compositor and Composition classes
  - Compositor: class encapsulating formatting algorithm
    - Pass Composition objects to be formatted as parameters to Compositor methods
  - Composition: things being formatted
    - Glyph subclass
    - Each Composition object refers to its Compositor object
    - When a Composition needs to format itself, it sends a message to its Compositor instance
Class Diagram

Object Structure after Formatting
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
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
Abstract Syntax Trees

public interface Node {
}

public class Number extends Node {
    public int n;
}

public class Plus extends Node {
    public Node left;
    public Node right;
}

Traversing Abstract Syntax Trees

public interface Node {
    public int sum();
}

public class Number extends Node {
    public int n;
    public int sum() { return n; }
}

public class Plus {
    public Node left;
    public Node right;
    public int sum() { return left.sum() + right.sum(); }
}
Naïve approach (not a visitor)

One method for each analysis

Node
- TypeCheck()
- GenerateCode()
- PrettyPrint()

VariableRefNode
- TypeCheck()
- GenerateCode()
- PrettyPrint()

AssignmentNode
- TypeCheck()
- GenerateCode()
- PrettyPrint()

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
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

```plaintext
aNodeStructure  aAssignmentNode  aVariableRefNode  aTypeCheckingVisitor
```

```
Accept (aTypeCheckingVisitor)  VisitAssignment(aAssignmentNode)  someOperation()

Accept (aTypeCheckingVisitor)  VisitVariableRef(aVariableRefNode)  someOperation()
```
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
Structure of Visitor Pattern

Use of Visitor Pattern in Lexi

a Character ("a") another Character ("_") aSpellingChecker

CheckMa(aSpellingChecker) CheckCharacter(this) GetCharacter() checks completed word

CheckMa(aSpellingChecker) CheckCharacter(this) GetCharacter()
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

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
    - \texttt{o.accept(aVisitor)} will dispatch on the actual identity of \texttt{o} (the object being considered)
    - ...and accept will internally dispatch on the identity of \texttt{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)

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)
        ...
    }
}
```
- Or visitors pass around a separate state object
  - Impacts the type of the Visitor superclass
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

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(this);
        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)
• 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
Traversing 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);
  }
  ...
}
Adding Scroll Bars and Borders: Decorator

- How to define classes for scrollbars and borders?
- Define as subclasses of Glyph
  - Scrollbars and borders are displayable objects
  - Will use notion of **transparent enclosure**
    - Clients don’t need to know whether they are dealing with a component or with an enclosure
- Inheritance increases number of classes
  - Use composition instead (“has a”)
Transparent Enclosure

- Two features:
  - Single-child composition
    - Calls its child, then adds its own behavior
  - Compatible interfaces
    - Can use the enclosing object in place of the one it encloses
- Implemented by the Decorator pattern
  - Saw this earlier

---

Monoglyph class: a Decorator

```java
Class Monoglyph { …
    void Draw (Window w) {
        component.Draw(w);
    } …
}

Class Border extends Monoglyph { …
    void Draw (Window w) {
        super.Draw(w);
        DrawBorder(w);
    } …
}
```
Changing Look-and-Feel: Abstract Factory

- Goal: easily change Lexi’s look-and-feel
  - When new libraries are available (future variability)
  - At run-time by switching between them (present variability)
- Thoughtless implementation technique:
  - Use distinct class for each widget and standard
  - Let clients handle different instances for each standard
    - Button pb = new MotifButton(); // bad

Abstracting Creation

- Concrete Creation problems:
  - Class of object is fixed at compile-time
    - Can’t change standard at run-time
  - Changing the class means making changes all over the code
- Instead:
  - Use a class to create abstract classes:
    - Button pb = guiFactory.createButton(); // better
Solution: Use Abstract Factory

- Define abstract class GUIFactory with creation methods for widgets
  - Concrete subclasses of GUIFactory actually define creation methods for each look-and-feel standard
    - MotifFactory, MacFactory, etc.
  - Specialize each widget into subclasses for each look-and-feel standard
- Thus, can easily change the kind of factory without changes all over the place

Class Diagram for GUIFactory
Abstract Factory Pattern

• Name
  – Abstract Factory or Kit

• Applicability
  – Different families of components (products)
  – Must be used in mutually exclusive and consistent way
  – Hide existence of multiple families from clients
Abstract Factory: Consequences

- Isolate instance creation and handling from clients
- Can easily change look-and-feel standard
  - Reassign a global variable;
  - Recompute and redisplay the interface
- Enforce consistency among products in each family
- Adding to family of products is difficult
  - Have to update factory abstract class and all concrete classes
Multiple Window Systems

- Want portability to different window systems
  - Similar to multiple look-and-feel problem, but different vendors will build widgets differently
- Solution:
  - Define abstract class Window, with basic window functionality (e.g., draw, iconify, move, resize, etc.)
  - Define concrete subclasses for specific types of windows (e.g., dialog, application, icon, etc.)
  - Define WindowImp hierarchy to handle window implementation by a vendor
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
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 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

[Diagram showing the structure of the Command Pattern with nodes labeled as Client, Invoker, Command, Receiver, and ConcreteCommand, with arrows indicating the relationships between them.]
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
Comparing Objects

- Java has two designs for objects that can be (totally) ordered
  - These are things for which sorting makes sense
  - E.g., strings, integers, etc.

Comparable

```java
public interface Comparable {
    // Returns negative integer, zero, or a positive integer if this object is less than, equal to, or greater than o.
    public int compareTo(Object o);
}
```

- Advantages and disadvantages?
  - Can only implement one compareTo operation
  - No extra levels of indirection; objects know how to compare themselves
public interface Comparator {
    int compare(Object o1, Object o2);
}

• Advantages and disadvantages?
  – Can have multiple comparison operations
  – An example of delegation
  – Comparator needs to know innards of your objects
    • Can make the Comparator implementer an inner class
  – Extra indirection; more objects floating around

• 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