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

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

Abstract Syntax Trees

```java
public interface Node {
    public int sum();
}
public class Number extends Node {
    public int n;
    public int sum() { return n; }
}
public class Plus extends Node {
    public Node left;
    public Node right;
    public int sum() { return left.sum() + right.sum(); }
}
```

Traversing Abstract Syntax Trees

```java
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

```java
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(); }
}
```

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

Sample Visitor class

```java
public interface NodeVisitor {
    void visitAssignment(AssignmentNode);
    void visitVariableRef(VariableRefNode);
}
public class TypeCheckingVisitor implements NodeVisitor {
    void visitAssignment(AssignmentNode) {
    }
    void visitVariableRef(VariableRefNode) {
    }
}
public class CodeGeneratingVisitor implements NodeVisitor {
    void visitAssignment(AssignmentNode) {
    }
    void 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

![Diagram showing traversal through objects](image)

Visitor Interaction

![Diagram showing visitor interaction](image)

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

![Diagram showing structure of visitor pattern](image)
Use of Visitor Pattern in Lexi

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
    - 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 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
    ```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
    - 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()` methods
    - Separating processing from traversal
  - `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`
  ```java
  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)

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);
  }
  ...
}

Back to Lexi

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 code
    }
}

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

Diagram for Product Classes
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