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

```
Node
TypeCheck()
GenerateCode()
PrettyPrint()

VariableRefNode
TypeCheck()
GenerateCode()
PrettyPrint()

AssignmentNode
TypeCheck()
GenerateCode()
PrettyPrint()
```
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

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

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 hierachies
  - Iterator needs a common superclass (i.e. composite pattern)
- Visitor can accumulate state rather than pass it a 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

```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 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
    - 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)

Preorder visitor

- 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

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