Design Patterns
• We discussed the following design patterns
  – Adapter
  – Abstract Factory
  – Bridge
  – Observer
What else we’re going to cover

• Some quickies
  – Iterator
  – Decorator
  – Singleton

• One important and big pattern
  – Visitor pattern
Midterm

• Tuesday, Oct 24th
  – Let me know now if you have a scheduled conflict, or need to take the test in anyway other than being in this room on the 24th

• Cover everything discussed so far
  – plus, refactoring and bad code smells
• There are lots of things you can iterate over.
• Lots of places where you want to iterate over one of those things

• Rather than have each place worry about “how do I iterate over the elements here”
  – use a design pattern, and standard interface, so we can use the same idiom and the same code
What could we iterate over?

- Collection
- Tree (pre-order, post-order, in-order)
- Lines of text in a file
- Prime numbers
- Random numbers
More on Iterators

- Java 5 added Iterable
  - anything that has an iterator method
- Can use the new for loop syntax over anything that implements Iterable
- Iterating over something shouldn’t change it
- Java’s Iterator class provides a remove method
  - not supported by all implementations
- The semantics of iterating over something that is modified during the iteration is implementation dependent
Decorator pattern

• Read sample chapter from Head First Design Patterns
Comments on decorators

- Multiple decorators results in a sequences of decorators
  - not all orders give the same behaviors
  - if a decorator adds any new methods, they are only visible if outermost
    • or if a reference to the inner decorator is retained
Singleton

• A class for which you want to create at most one instance in any JVM
  – You want any code in the JVM using this class to be using the same instance of the class
• Singletons often implement an interface or extend a class
  – otherwise, you could just use static methods
Two basic cases

• Stateless singletons
  – the advantage of a stateless singleton is that you only create is once (which might have side effects) and you don’t pay the cost of duplicated instances

• Stateful singletons
  – everyone needs to see updates to the state of the one and only singleton
    • everyone includes multiple threads
Creating a singleton

- Far easier than you might suspect.
- Lot of books and references get this wrong.

```java
final class Elvis extends Person {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }
}
```
Guaranteed Lazy and Thread Safe

• The field will be set by the class’s static initializer
• The static initializer is not invoked until the first active use of the class:
  – invoking a static method or accessing a static field
  – creating an instance of the class
  – initializing a subclass
• JVM uses locking so that even if multiple threads try to access INSTANCE at the same time, it just works
Double Checked Locking

- If someone tells you should use double checked locking for a singleton, tell them they are wrong
- Double checked locking does have some use cases
  - lazy thread safe initialization
- But you need to use volatile to make it work
  - we’ll come back to this
Visitor Pattern
Visitor: Implementing Analyses

• Often want to implement multiple analyses on the same kind of object data
  – Book example: computing with Menus
  – Project example: Generating code for and analyzing an Abstract Syntax Tree (AST) in a compiler

• One solution: implement each analysis as a method in each object
public interface Node {
}

public class Number extends Node {
    public int n;
}

public class Plus extends Node {
    public Node left;
    public Node right;
}
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
Tradeoffs with this Approach

- Follows idea “objects are responsible for themselves”
- But many analyses will occlude the object’s main code
- Result is classes that are hard to maintain
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
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
Vistor Interaction

\[ a\text{NodeStructure} \quad a\text{AssignmentNode} \quad a\text{VariableRefNode} \quad a\text{TypeCheckingVisitor} \]

- Accept (aTypeCheckingVisitor)
- VisitAssignment(aAssignmentNode)
- someOperation()
- Accept (aTypeCheckingVisitor)
- VisitVariableRef (aVariableRefNode)
- someOperation()
public interface Visitor {
    public void visitNumber(Number n);
    public void visitPlus(Plus p);
}

public class SumVisitor implements Visitor {
    int sum;
    public void visitNumber(Number n) { sum += n; }
    public void visitPlus(Plus p) {
        p.left.accept(this);
        p.right.accept(this);
    }
}
public interface Node {
    public void accept(Visitor v);
}

public class Number extends Node {
    ...
    public void accept(Visitor v) {v.visitNumber(this);}
}
public class Plus extends Node {
    ...
    public void accept(Visitor v) {v.visitPlus(this);}
}
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
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
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

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

• Or visitors pass around a separate state object
  – Impacts the type of the Visitor superclass
Who is responsible for traversing object structure?

- **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
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
• Class BinaryPlusOperatorNode {
  void accept(Visitor v) {
    v.visit(this);
    lhs.accept(v);
    rhs.accept(v);
  }
  ...
}
• 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
Visitor/Process Methods

- Can have two parallel sets of methods in visitors
  - Visit() methods
  - Process() methods
- 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)
- Allows finer-grained subtyping of Visitor classes that include traversal
  - Subclass a visitor, and just change the process method
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
• 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
• Class PreorderVisitor {
    Visitor payload;
    PreorderVisitor(Visitor p) { payload = p; }
    void visit(BinaryPlusOperatorNode n) {
        payload.visit(n);
        n.lhs.accept(this);
        n.rhs.accept(this);
    }
    ...
}