CS 565 September 11, 1997
Jens Palsberg Readings: The handout on visitors

Lecture Note 4:

A mostly Functional Programming Style in Java with the help of Design Patterns

Contents

1 Design Patterns

Book: **Design Patterns**, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1995.

The book has a subtitle: **Elements of Reusable Object-Oriented Software**.

The book describes simple and elegant solutions to specific problems in object-oriented software design. Each solution is a design pattern.

The first sentence in Chapter 1: “Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder.”

All of the 23 design patterns in the book have been applied more than once in different systems.

My slogan: A design pattern is a piece of Expert Knowledge.

The idea of a pattern comes from patterns in buildings and towns.

In the book, a pattern has four essential elements:

1. the pattern name,
2. the problem,
3. the solution,
4. the consequences.

Each pattern description is divided into sections according to the following template.

Pattern Name and Classification. The essence.

Intent. A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As. Other well-known names for the pattern.

Motivation. A scenario.

Applicability. What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Structure. A graphical representation of the classes involved.

Participants. The classes and their responsibilities.

Collaborations. How the participants collaborate.

Consequences. Trade-offs and results of using the pattern.

Implementation. Pitfalls, hints, techniques, language-specific issues.

Sample Code.

Known Uses.

Related Patterns.

2 The Visitor Pattern

Intent. Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Why Visitors?

The Visitor pattern is one among many design patterns aimed at making object-oriented systems more flexible. The issue addressed by the Visitor pattern is the manipulation of composite objects. Without visitors, such manipulation runs into several problems as illustrated by the following example. Consider the following implementation of integer lists, written in Java.

```
interface List {}

class Nil implements List {}

class Cons implements List {
  int head;
  List tail;
}
```

Let us now write a program which computes the sum of all components of a given `List`-object.

First Attempt: Instanceof and Type Casts.

The first attempt may be a loop which uses instanceof to check if a given `List`-object is a `Nil`-object or a `Cons`-object. If it is a `Cons`-object, then the fields are accessed via type casts, and the loop is repeated.

```
List l;       // The List-object we are working on.
int sum = 0; // Contains the sum after the loop.
boolean proceed = true;
while (proceed) {
  if (l instanceof Nil)
     proceed = false;
  else if (l instanceof Cons) {
     sum = sum + ((Cons) l).head; // Type cast!
     l = ((Cons) l).tail;         // Type cast!
  }
}
```

The advantage of this code is that it can be written without touching the classes `Nil` and `Cons`. The drawback is that the code constantly uses type casts and instanceof to determine what class of object it is considering.

Second Attempt: Dedicated Methods.

One can dismiss the above piece of code by simply saying: it is not object-oriented! To access parts of an object, the classical approach is to use dedicated methods which both access and act on the subobjects. For our running example, we can insert sum methods into each of the classes:

```
interface List {
  int sum();
}

class Nil implements List {
  public int sum() {
    return 0;
  }
}

class Cons implements List {
  int head;
  List tail;
  public int sum() {
    return head + tail.sum();
  }
}
```

We can now compute the sum of all components of a given `List`-object `l` by writing `l.sum()`.

The advantage of this code is that the type casts and instanceof operations have disappeared, and that the code can be written in a systematic way. The disadvantage is that every time we want to perform a new operation on `List`-objects, say, compute the product of all integer parts, then new dedicated methods have to be written for all the classes, and the classes must be recompiled.

Third Attempt: The Visitor Pattern.

The Visitor pattern lets us define a new operation on an object structure without changing the classes of the objects on which it operates. Rather than writing dedicated methods for each programming task and afterwards recompiling, the idea is to (1) insert a so-called accept method in each class, and (2) write the action code in so-called visitors.

For our running example, the basic classes and the visitor class look like this:

```
interface List {
  void accept(Visitor v);
}

class Nil implements List {
  public void accept(Visitor v) {
    v.visitNil(this);
  }
}

class Cons implements List {
  int head;
  List tail;
  public void accept(Visitor v) {
    v.visitCons(this);
  }
}
```

```
interface Visitor {
  void visitNil(Nil x);
  void visitCons(Cons x);
}

class SumVisitor implements Visitor {
  int sum = 0;
  public void visitNil(Nil x) {}
  public void visitCons(Cons x) {
    sum = sum + x.head;
    x.tail.accept(this);
  }
}
```

Each accept method takes a visitor as argument. The interface `Visitor` has a header for each of the basic classes. We can now compute and print the sum of all components of a given `List`-object `l` by writing

```
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```

The advantage is that one can write code that manipulates objects of existing classes without recompiling those classes. The price is that all objects must have an accept method.

Consequences

The Visitor pattern combines the advantages of the two other approaches. In slogan-form:

	Frequent type casts?	Frequent recompilation?
Instanceof and type casts	Yes	No
Dedicated methods	No	Yes
The Visitor pattern	No	No

Visitor makes adding new operations easy. Simply write a new visitor.

A visitor gathers related operations. It also separates unrelated ones.

Adding new classes to the object structure is hard. Key consideration: are you most likely to change the algorithm applied over an object structure, or are you most like to change the classes of objects that make up the structure.

Visitors can accumulate state.

Visitor can break encapsulation. Visitor's approach assumes that the interface of the data structure classes is powerful enough to let visitors do their job. As a result, the pattern often forces you to provide public operations that access internal state, which may compromise its encapsulation.

3 The Java Compiler Compiler

- Current version: 0.6.1.
- Can be thought of as "Lex and Yacc for Java."
- It is based on LL(k) rather than LALR(1).
- Grammars are written in EBNF.
- The Java Compiler Compiler transforms an EBNF grammar into an LL(k) parser.
- The EBNF grammar can have embedded action code written in Java, just like a Yacc grammar can have embedded action code written in C.
- The whole input is given in just one file (not two).
- The lookahead can be changed by writing `LOOKAHEAD(...)`.

Example Input to JavaCC: The Java 1.1.3 Grammar

```
/**
 *
 * Copyright (C) 1996, 1997 Sun Microsystems Inc.
 *
 * Use of this file and the system it is part of is
 * constrained by the file COPYRIGHT in the root
 * directory of this system.  You may, however,
 * make any modifications you wish to this file.
 *
 * Java files generated by running JavaCC on this file
 * (or modified versions of this file) may be used in
 * exactly the same manner as Java files
 * generated from any grammar developed by you.
 *
 * Author: Sriram Sankar
 * Date: 3/5/97
 *
 * This file contains a Java grammar and actions that
 * implement a front-end.
 *
 */

options {
  JAVA_UNICODE_ESCAPE = true;
}
```

```
PARSER_BEGIN(JavaParser)

import ast_test.*;

public class JavaParser {

  public static void main(String args[]) {
    JavaParser parser;
    if (args.length == 0) {
      System.out.println("Java Parser Version 1.1:
                Reading from standard input . . .");
      parser = new JavaParser(System.in);
    } else if (args.length == 1) {
      System.out.println("Java Parser Version 1.1:
                Reading from file " + args[0] + " . . .");
      try {
        parser = new JavaParser(new
                java.io.FileInputStream(args[0]));
      } catch (java.io.FileNotFoundException e) {
        System.out.println("Java Parser Version 1.1:
                File " + args[0] + " not found.");
        return;
      }
    } else ...
  }

}

PARSER_END(JavaParser)
```

```
SKIP : /* WHITE SPACE */
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

SPECIAL_TOKEN : /* COMMENTS */
{
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])*
        ("\n"|"\r"|"\r\n")>
| <FORMAL_COMMENT: "/**" (~["*"])* "*"
        ("*" | (~["*","/"] (~["*"])* "*"))* "/">
| <MULTI_LINE_COMMENT: "/*" (~["*"])* "*"
        ("*" | (~["*","/"] (~["*"])* "*"))* "/">
}

TOKEN : /* RESERVED WORDS AND LITERALS */
{
  < ABSTRACT: "abstract" >
| < BOOLEAN: "boolean" >
| < BREAK: "break" >
| < BYTE: "byte" >
| < CASE: "case" >
| < CATCH: "catch" >
...
| < WHILE: "while" >
}
```

```
TOKEN : /* LITERALS */
{
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
      | <HEX_LITERAL> (["l","L"])?
      | <OCTAL_LITERAL> (["l","L"])?
  >
|
  < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
|
  < #HEX_LITERAL: "0" ["x","X"]
          (["0"-"9","a"-"f","A"-"F"])+ >
|
  < #OCTAL_LITERAL: "0" (["0"-"7"])* >
|
  < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])*
        (<EXPONENT>)? (["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
  >
|
  < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
  ...
}
```

```
TOKEN : /* IDENTIFIERS */
{
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|
  < #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
  >
|
  < #DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
```

```
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]
  >
}

TOKEN : /* SEPARATORS */
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACE: "{" >
| < RBRACE: "}" >
| < LBRACKET: "[" >
| < RBRACKET: "]" >
| < SEMICOLON: ";" >
| < COMMA: "," >
| < DOT: "." >
}
```

```
TOKEN : /* OPERATORS */
{
  < ASSIGN: "=" >
| < GT: ">" >
| < LT: "<" >
| < BANG: "!" >
| < TILDE: "~" >
| < HOOK: "?" >
| < COLON: ":" >
| < EQ: "==" >
| < LE: "<=" >
| < GE: ">=" >
| < NE: "!=" >
| < SC_OR: "||" >
| < SC_AND: "&&" >
| < INCR: "++" >
| < DECR: "--" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < BIT_AND: "&" >
| < BIT_OR: "|" >
| < XOR: "^" >
| < REM: "%" >
| < LSHIFT: "<<" >
...
}
```

```
/****************************************
 * THE JAVA LANGUAGE GRAMMAR STARTS HERE *
 ****************************************/

/*
 * Program structuring syntax follows.
 */

void CompilationUnit() :
{}
{
  [ PackageDeclaration() ]
  ( ImportDeclaration() )*
  ( TypeDeclaration() )*
  <EOF>
}

void PackageDeclaration() :
{}
{
  "package" Name() ";"
}

void ImportDeclaration() :
{}
{
  "import" Name() [ "." "*" ] ";"
}
```

```
void TypeDeclaration() :
{}
{
  LOOKAHEAD( ( "abstract" | "final" | "public" )* "class" )
  ClassDeclaration()
|
  InterfaceDeclaration()
|
  ";"
}


/*
 * Declaration syntax follows.
 */

void ClassDeclaration() :
{}
{
  ( "abstract" | "final" | "public" )*
  UnmodifiedClassDeclaration()
}

void UnmodifiedClassDeclaration() :
{}
{
  "class" <IDENTIFIER> [ "extends" Name() ]
                       [ "implements" NameList() ]
  ClassBody()
}
```

```
void ClassBody() :
{}
{
  "{" ( ClassBodyDeclaration() )* "}"
}

void NestedClassDeclaration() :
{}
{
  ( "static" | "abstract" | "final"
             | "public" | "protected" | "private" )*
  UnmodifiedClassDeclaration()
}
```

```
void ClassBodyDeclaration() :
{}
{
  LOOKAHEAD(2)
  Initializer()
|
  LOOKAHEAD( ( "static" | "abstract" | "final"
      | "public" | "protected" | "private" )* "class" )
  NestedClassDeclaration()
|
  LOOKAHEAD( ( "static" | "abstract" | "final" | "public"
             | "protected" | "private" )* "interface" )
  NestedInterfaceDeclaration()
|
  LOOKAHEAD( [ "public" | "protected" | "private" ]
             Name() "(" )
  ConstructorDeclaration()
|
  LOOKAHEAD( MethodDeclarationLookahead() )
  MethodDeclaration()
|
  FieldDeclaration()
}
```

```
// This production is to determine lookahead only.
void MethodDeclarationLookahead() :
{}
{
  ( "public" | "protected" | "private" | "static"
   | "abstract" | "final" | "native" | "synchronized" )*
  ResultType() <IDENTIFIER> "("
}

void InterfaceDeclaration() :
{}
{
  ( "abstract" | "public" )*
  UnmodifiedInterfaceDeclaration()
}

void NestedInterfaceDeclaration() :
{}
{
  ( "static" | "abstract" | "final" | "public"
  | "protected" | "private" )*
  UnmodifiedInterfaceDeclaration()
}

void UnmodifiedInterfaceDeclaration() :
{}
{
  "interface" <IDENTIFIER> [ "extends" NameList() ]
  "{" ( InterfaceMemberDeclaration() )* "}"
}
```

```
...

void Assignment() :
{}
{
  PrimaryExpression() AssignmentOperator() Expression()
}

void AssignmentOperator() :
{}
{
  "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<="
      | ">>=" | ">>>=" | "&=" | "^=" | "|="
}

...
```

4 The Java Tree Builder

The Java Tree Builder (JTB) has been developed here at Purdue in my group.

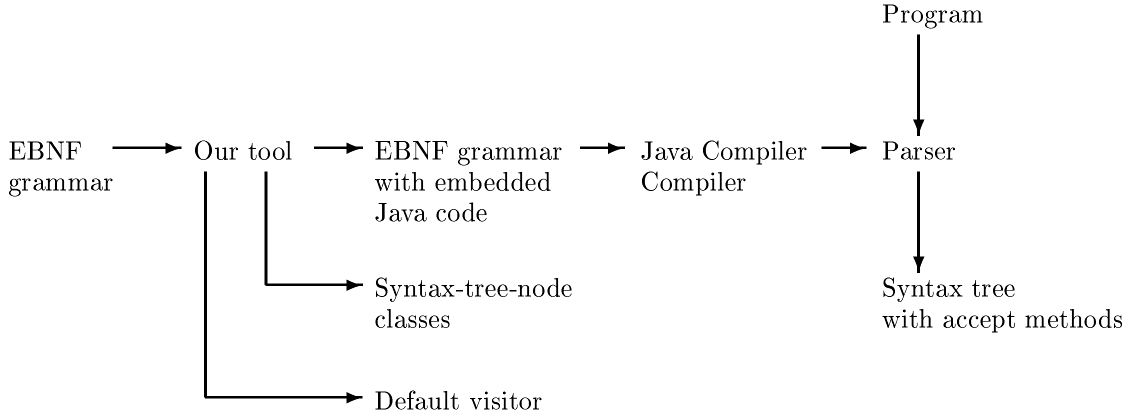
JTB is a frontend for The Java Compiler Compiler.

JTB supports the building of syntax trees which can be traversed using visitors. When all tree traversal and manipulation code is written using visitors, the classes for the syntax tree nodes are generated once by JTB and never touched again. This gives a welcome degree of flexibility and modularity for projects where several programmers, perhaps over time, write code for traversing syntax trees generated from the same grammar. Rather than modifying the classes for the syntax tree nodes, each programmer writes one or more visitors.

The current version of our tool transforms a bare EBNF grammar into three components:

- an EBNF grammar with embedded Java code for building a syntax tree;
- one class for every form of syntax tree node; and
- a default visitor which can do a depth-first traversal of a syntax tree.

The produced EBNF grammar can then be processed by the Java Compiler Compiler to give a parser which produces syntax trees. The produced syntax trees can now be traversed by a Java program by writing subclasses of the default visitor.



Example

For example, consider the Java 1.1 production

```
void Assignment() : {}
   { PrimaryExpression() AssignmentOperator() Expression() }
```

Our tool produces an annotated grammar where the production for `Assignment` is:

```
Assignment Assignment () :
{ PrimaryExpression n0;
  AssignmentOperator n1;
  Expression n2; {} }
{ n0=PrimaryExpression()
  n1=AssignmentOperator()
  n2=Expression()
  { return new Assignment(n0,n1,n2); }
}
```

Notice that the production returns a syntax tree represented as an `Assignment` object.

The produced syntax-tree-node class for `Assignment` is:

```
public class Assignment implements Node {
  PrimaryExpression f0; AssignmentOperator f1; Expression f2;

  public Assignment(PrimaryExpression n0,
                    AssignmentOperator n1,
                    Expression n2)
  { f0 = n0; f1 = n1; f2 = n2; }

  public void accept(Visitor v)
  { v.visitAssignment(this); }
}
```

Notice the `accept` method; it invokes the method `visitAssignment` in the default visitor. The default visitor looks like this:

```
public class Visitor {
  ...
   void visitAssignment(Assignment n)
   { n.f0.accept(this); n.f1.accept(this); n.f2.accept(this); }
   ...
}
```

Notice the body of the method which visits each of the three subtrees of the `Assignment` node.

Here is an example of a program which operates on syntax trees for Java 1.1 programs. The program prints the right-hand side of every assignment. The entire program is six lines:

```
public class VprintAssignRHS extends Visitor {
   void visitAssignment(Assignment n) {
      VPrettyPrinter v = new VPrettyPrinter();
      n.f2.accept(v); v.out.println(); n.f2.accept(this);
   }
}
```

When this visitor is passed to the root of the syntax tree, the depth-first traversal will begin, and when `Assignment` nodes are reached, the method `visitAssignment` in `VprintAssignRHS` is executed. Notice the use of `VPrettyPrinter`. It is a visitor which pretty prints Java 1.1 programs.

Our tool is bootstrapped.