Compiler Techniques

Visitor Design Pattern in Lab 4

Ta N. B. Duong

Outline

This introduction is designed to give some background information for completing Lab Assignment 4

Many of the classes in Lab Assignment 4 make heavy use of the Visitor Design Pattern so a brief description is provided

There is also a discussion of the use of the Visitor Design Pattern in Lab Assignment 4, including two detailed examples: code generation for the If Statement and Comparison Expression

- Introduction to Visitor Design Pattern
- Use of the Visitor Design Pattern in Lab Assignment 4
- Examples: If Statement and Comparison Expression

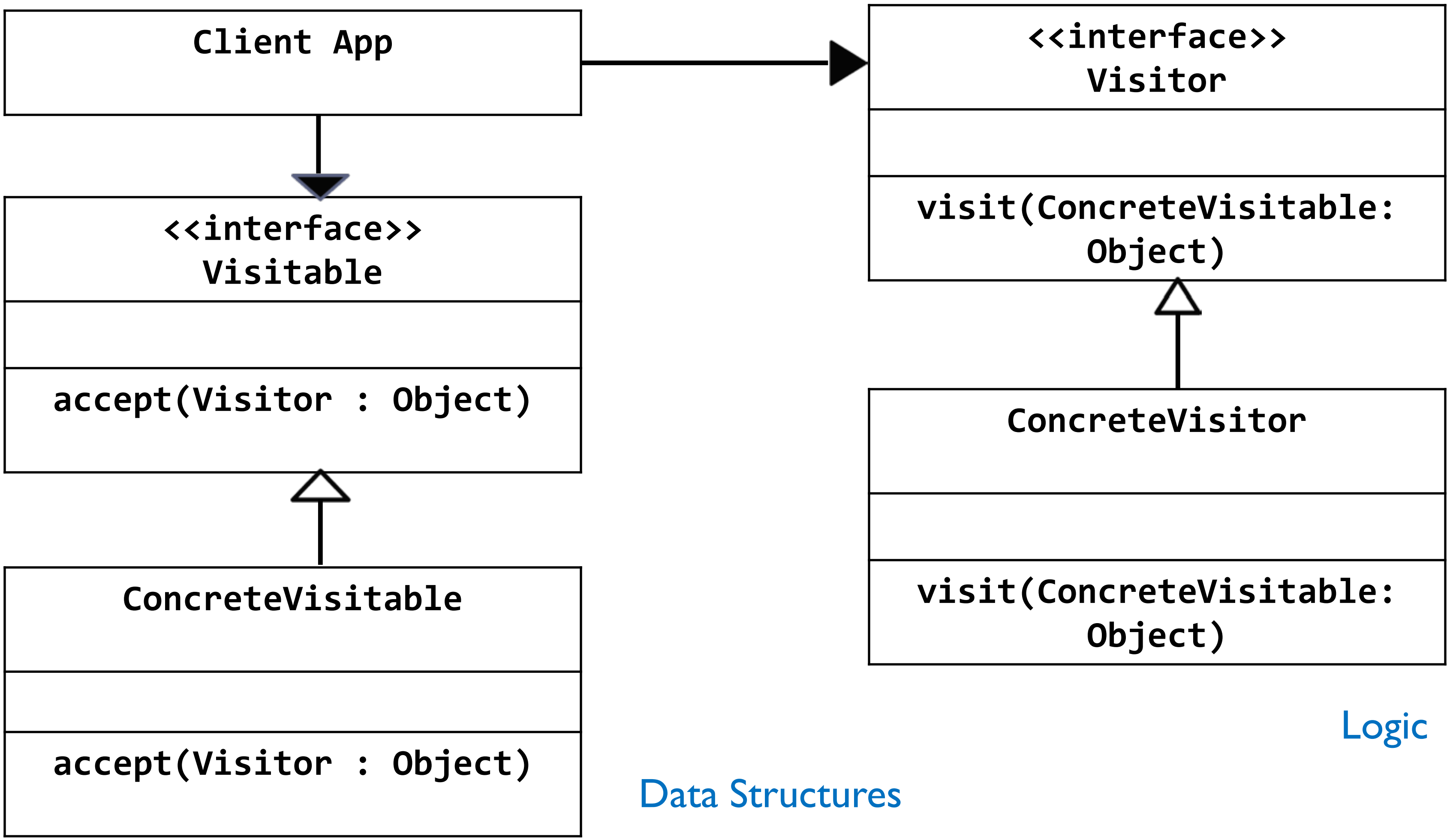
Introduction to Visitor Design Pattern

- The Visitor Design Pattern allows the logic and the data structures of an application to be decoupled, while at the same time applying the logic to the data structures
- With this pattern, classes can be built that focus only on the data structures without knowing the logic that will be applied to the structures
- At the same time, classes can be built that concentrate solely on the logic that will be applied to the structures without knowing what the structures looks like
- The Visitor Design Pattern is useful when we have to perform different operations on a group of similar objects

Classes used in this Pattern

- **Visitor** – This is an interface, abstract class, or superclass that declares the visit() method
- **ConcreteVisitor** – Implements the Visitor interface or extends from the abstract class or superclass
  - Each ConcreteVisitor class represents a different logic
- **Visitable** – This is an interface, abstract class, or superclass that declares the accept() method
- **ConcreteVisitable** – Implements the Visitable interface or extends from the abstract class or superclass
  - Each ConcreteVisitable class represents an element of the data structures
  - It implement the accept() method which calls the visit() method of the visitor, passing itself as an argument, visit(this)

Classes used in this Pattern (2)



Example Visitor Design Pattern

- See: http://www.journaldev.com/1769/visitor-design-pattern-in-java-example-tutorial
- Think of a Shopping cart where we can add different types of items (Elements)
- When we click on the checkout button, it calculates the total amount to be paid
- Instead of having the calculation logic in the item classes, we can move this logic out to another class using the visitor design pattern

Example (ItemElement.java)

Visitable Interface:

```
package com.journaldev.design.visitor;
public interface ItemElement {
    public int accept(ShoppingCartVisitor visitor);
}
```

The accept method takes a Visitor object as its argument

We can create some Concrete Visitable classes for different types of item, e.g. book, fruit, etc

Not all the methods in the classes are shown, only those relevant to the Visitor Design Pattern

Example (Book.java)

```
package com.journaldev.design.visitor;
public class Book implements ItemElement {
    private int price; private String isbnNumber;

    public Book(int cost, String isbn){ ... }

    public int getPrice() { return price; }
    ...
    @Override
    public int accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }
}
```

Note the implementation of the accept() method, it calls the visit() method of the Visitor and passes itself as the argument

Example (Fruit.java)

```
package com.journaldev.design.visitor;
public class Fruit implements ItemElement {
    private int pricePerKg; private int weight;
    private String name;

    public Fruit(int priceKg, int wt, String nm) { ... }

    public int getPricePerKg() { return pricePerKg;     }

    public int getWeight() { return weight; }
    ...
    @Override
    public int accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }
}
```

Example (ShoppingCartVisitor.java)

Visitor Interface:

```
package com.journaldev.design.visitor;
public interface ShoppingCartVisitor {
    int visit(Book book);
    int visit(Fruit fruit);
}
```

The visit method takes a particular Concrete Visitable object as an argument

Now we will implement the Visitor Interface and every item will have its own logic to calculate the cost

Example (ShoppingCartVisitorImpl.java)

```
package com.journaldev.design.visitor;

public class ShoppingCartVisitorImpl implements
    ShoppingCartVisitor {

    @Override
    public int visit(Book book) {
        int cost=0;
        if (book.getPrice() > 50){
            cost = book.getPrice()-5;  // apply discount
        } else cost = book.getPrice();
        return cost;
    }
```

Example (ShoppingCartVisitorImpl.java)

```
    @Override
    public int visit(Fruit fruit) {
        int cost =
            fruit.getPricePerKg()*fruit.getWeight();
        return cost;
    }
}
```

We see how we can use it in client applications

Example (ShoppingCartClient.java)

```java
public class ShoppingCartClient {
    public static void main(String[] args) {

        ItemElement[] items = new ItemElement[]{
            new Book(20, "1234"),
            new Book(100, "5678"),
            new Fruit(10, 2, "Banana"),
            new Fruit(5, 5, "Apple")
        };

        int total = calculatePrice(items);
        System.out.println("Total Cost = "+total);
    }
```

The calculatePrice method makes use of the Visitor Design Pattern

Example (ShoppingCartClient.java)

```
    private static int
      calculatePrice(ItemElement[] items) {
        ShoppingCartVisitor visitor = new
          ShoppingCartVisitorImpl();
        int sum=0;
        for(ItemElement item : items){
            sum = sum + item.accept(visitor);
        }
        return sum;
    }
}
```

Note that we can have different implementations of the Visitor Interface and the implementation of the accept() method can be different for different items

Why this Works

- When the accept() method is called in the program, its implementation is chosen based on both:
  - The dynamic type of the element
  - The static type of the visitor
- When the associated visit() method is called, its implementation is chosen based on both:
  - The dynamic type of the visitor
  - The static type of the element as known from within the implementation of the accept() method, which is the same as the dynamic type of the element
- Consequently, the implementation of the visit() method is chosen based on both:
  - The dynamic type of the element
  - The dynamic type of the visitor

Use of the Visitor Design Pattern in Lab 4

- The Visitable superclass is the class for node type ASTNode (the root AST node type)
- The Concrete Visitable classes are the classes for the different AST nodes
- The accept method for a node of type N will invoke the appropriate method visitN, and pass itself as the argument, e.g.
  - The accept(Visitor<A> visitor) method for IfStmt invokes visitIfStmt
    - return visitor.visitIfStmt(this)
  - The accept(Visitor<A> visitor) method for AddExpr invokes visitAddExpr
    - return visitor.visitAddExpr(this);

Use of the Visitor Design Pattern in Lab 4

- The Visitor superclass is a generic class that takes the return type of the visit method as a parameter
- The Concrete Visitor classes are:
  - StmtCodeGenerator extends Visitor<Void>
  - ExprCodeGenerator extends Visitor<Value>
  - and some anonymous classes, e.g. for comparison expressions
- The visit method for node type N is called visitN and takes a node of type N as argument
- To apply a visitor object v to an AST node nd of type N, use the method ASTNode.accept like this: nd.accept(v)
- At runtime, the accept will invoke the appropriate method visitN and pass nd itself as the only argument

Example: If Statement

```
public Void visitIfStmt(IfStmt nd) {
	Value cond = ExprCodeGenerator.generate(nd.getExpr(), fcg);
	NopStmt join = j.newNopStmt();
	units.add(j.newIfStmt(j.newEqExpr(cond, IntConstant.v(0)), join));
	nd.getThen().accept(this);
	if(nd.hasElse()) {
		NopStmt els = join;
		join = j.newNopStmt();
		units.add(j.newGotoStmt(join));
		units.add(els);
		nd.getElse().accept(this);
	}
	units.add(join);
}
```

Example: If Statement (2)

- Here, the Concrete Visitable class is IfStmt (the type of the AST node) and the Concrete Visitor class is StmtCodeGenerator
- We generate code for an If Statement by calling the accept method on the AST node nd for the If Statement, nd.accept(visitor), where visitor is an instance of StmtCodeGenerator
- The accept method will call visitor.visitIfStmt(this), where this is a reference to the AST node for the If Statement
- In the visitIfStmt method, we first generate code for the condition by calling ExprCodeGenerator.generate
  - nd is the node of type IfStmt in the AST
  - fcg is the reference to the function code generator from which we get the current list of statements (i.e. units) we have generated for the function body

Example: If Statement (3)

- We generate a conditional jump to the NOP statement representing the label join, where j is the singleton instance of Jimple
  - If there is an else part, join is the start of the else part
  - If there is no else part, join is the statement after the If Statement
- We add the conditional jump to units, the current list of statements we have generated for the function body
- We generate code for the then part by calling the accept method in the AST node for the then part, nd.getThen().accept(this), where this is the visitor, i.e. the current instance of StmtCodeGenerator
- The accept method will call the appropriate visit method on the visitor depending on the type of statement in the then part
- We handle the else part similarly if there is one, adding the appropriate statements to the list of units

Example: Comparison Expression

```
public Value visitCompExpr(CompExpr nd) {
    final Value left = wrap(nd.getLeft().accept(this)),
                right = wrap(nd.getRight().accept(this));
    Value res = nd.accept(new Visitor<Value>() {
        @Override
        public Value visitEqExpr(EqExpr nd) {
            return Jimple.v().newEqExpr(left, right);
        }
        // Similarly for NeqExpr, LtExpr, GtExpr, LeqExpr, GeqExpr
        ...
    });
    // compute a result of 0 or 1 depending on the truth value
    ...
}
```

Example: Comparison Expression (2)

- Here, the Concrete Visitable class is a subclass of CompExpr (e.g. EqExpr) and the Concrete Visitor class is ExprCodeGenerator
- We generate code for a Comparison Expression by calling the accept method of the AST node nd for the subclass (e.g. EqExpr), nd.accept(visitor), where visitor is an instance of ExprCodeGenerator
- The accept method will call visitor.visitEqExpr(this), where this is a reference to the AST node for EqExpr
- As there is no method declared for visitEqExpr in the class ExprCodeGenerator, we use the method declared in the abstract Visitor class:

```
public A visitEqExpr(EqExpr nd) {
    return visitCompExpr(nd);
}
```

Example: Comparison Expression (3)

- In the visitCompExpr method, we first generate code for the left and right parts by calling the accept method of the respective nodes, nd.getLeft() and nd.getRight()
  - wrap will introduce a new temporary variable if necessary and store the value into the temporary variable
- The visitCompExpr method then calls the accept method of the AST node nd for the subclass (e.g. EqExpr), but this time the visitor object is an instance of an anonymous class, in which visitEqExpr is declared and overrides the method declared in the Visitor class
- The visitEqExpr method generates an instance of EqExpr, where j is the singleton instance of Jimple
- Finally, we compute a result of 0 or 1 depending on the truth value of the comparison, adding the appropriate statements to the list of units

Summary

- Benefits of Visitor Design Pattern
  - If the logic changes, we only need to make changes in the visitor implementation rather than in all the item classes
  - Adding a new item requires changes only in the visitor interface and implementation of a new visit method and existing item classes and visit methods will not be affected

- References
  - Wikipedia: http://en.wikipedia.org/wiki/Visitor_pattern
  - JournalDev: http://www.journaldev.com/1769/visitor-design-pattern-in-java-example-tutorial
  - YouTube: http://www.youtube.com/watch?v=pL4mOUDi54o