Comp-304 : Visitor
Lecture 29

Alexandre Denault
Original notes by Marc Provost
and Hans Vangheluwe
Computer Science
McGill University
Fall 2007
2 / 23 = 8.7%
It's got 11 Questions.  
It's 33% pre-midterm material.  
Do you want a tutorial?
Scene Graphs

Universe

Room 1
- Desk
  - Books
- Lamp

Room 2
- Bed
- Wardrobe
  - Doors
  - Drawers
I want to print out the content of the room.
To do this, I need to build a string containing a list of the items in the room.
How do I do this?
The class calling the universe.toString() method should not have information on how data is store in the universe.

Thus, universe.toString() should take care of traversing the tree.

This means that each node will need to have it's own toString() method.

If I want to calculate the weight of the universe, I will also need to add a getWeight() function to each node.

Is there a generic way I can traverse a tree without having to add new methods?
Visitor Pattern

- Represent an operation to be performed on the elements of an object structure.
- In other words, it allows you to separate the algorithm from the data structure.
Introduction to Compilers

- A compiler is a tool that transforms a program for a high level representation to a lower level representation.
  - Java -> Bytecode
  - C -> Assembler

- The first step of a compiler is to take the grammar of a language and transform the code into an abstract syntax tree.
  - Flex + Bison in C
  - SableCC in Java
int i = 5;
float j = 4.5;
float k = i + j;
Further operations are done by traversing the tree:
- Weeding
- Type Checking
- Symbol Table
- Code Generation

Do we want to add functions to every node we need to traverse?
- This would be the intuitive solution
- We would need the following functions: weed(), typeCheck(), symbol(), code()
Each node class is 'polluted' with several methods.

The implementation of an algorithm spread over all classes.
  • i.e. The weeding algo is spread across several node.

Do keep track of the traversal, either
  • must use global variables
  • must arguments passed by reference in each method call
Visitor Pattern Solution

Visitor

+visitStatementList(elem: StatementList)
+visitDeclaration(elem: Declaration)
+visitAssignment(elem: Assignment)
+visitMultiply(elem: Multiply)
+visitFloat(elem: Float)
+visitInt(elem: Int)
+visitIdentifier(elem: Identifier)

PrettyPrinterVisitor

+visitStatementList(elem: StatementList)
+visitDeclaration(elem: Declaration)
+visitAssignment(elem: Assignment)
+visitMultiply(elem: Multiply)
+visitFloat(elem: Float)
+visitInt(elem: Int)
+visitIdentifier(elem: Identifier)
Advantages

- The algorithm is now located in a single class.
  - All variables needed to execute the algorithm are also in the class.
  - No need for global variables anymore (or variables passed by reference).
- The AST class structure (tree) was not modified!
- It's easy to add new operations.
- A visitor can iterate over elements which are not sharing a common parent class.
However, if a new subtype of Node is added, all the visitors must be modified.

- For instance, we might want to add an 'Addition' node.
- This would require a new function 'visitAddition' in each visitor.

Encapsulation could be broken if a visitor needs to access an element internal state.
When dealing with data structures, it's highly possible that a node will contain references to other nodes (children, etc).

For the visitor pattern to work, the accept() calls must be propagated to the children nodes (other references).

Most often, the simplest solution is add this propagation to the accept() call of the parent.

```java
public void accept(Visitor visitor) {
    visitor.visit(this);
    for (Node node : nodes) {
        node.accept(visitor);
    }
}
```
Add the visitor pattern

```
«interface»
Visitor
+visitCar(e: Car)
+visitEngine(e: Engine)
+visitBody(e: Body)
+visitWheel(e: Wheel)
```
class Wheel {
    public void accept(Visitor visitor) {
        visitor.visitWheel(this);
    }
}

class Engine {
    public void accept(Visitor visitor) {
        visitor.visitEngine(this);
    }
}

class Body {
    public void accept(Visitor visitor) {
        visitor.visitBody(this);
    }
}
class Car implements Visitable {

    private Engine   engine;
    private Body     body;
    private Wheel[]  wheels;

    public void accept(Visitor visitor) {
        visitor.visitCar(this);
        engine.accept(visitor);
        body.accept(visitor);
        for(int i = 0; i < wheels.length; ++i) {
            wheels[i].accept(visitor);
        }
    }
}
class PrintVisitor implements Visitor {
    private static count = 0;

    public void visit(Wheel wheel) {
        count++;
        System.out.println("Visiting wheel "+ count);
    }

    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }

    public void visit(Body body) {
        System.out.println("Visiting body");
    }

    public void visit(Car car) {
        System.out.println("Visiting car");
    }
}