The Visitor Design Pattern

Jeonggeun Kim
Kyungpook National University

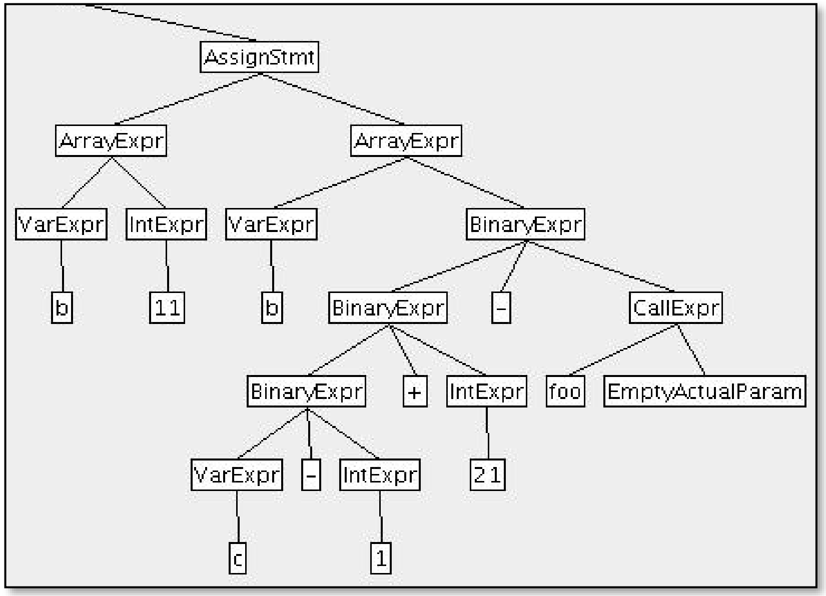
This lecture's materials are based on Prof. Bernd Burgstaller's slides (CSI4104 Compiler Design, Yonsei University)

We need operations on MiniC ASTs!

Class hierarchy:

```
+ AST
  =Program(Decl D)
  + Decl
     = FunDecl(Type tAST, ID idAST, Decl parmsAST,
     Stmt stmtAST)
     = VarDecl(Type tAST, ID idAST, Expr eAST)
     = FormalParamDecl(Type astType, ID astIdent)
  + Stmt
     = CompoundStmt(Decl astDecl, Stmt astStmt)
     = IfStmt(Expr eAST, Stmt s1AST (, Stmt s2AST)?)
     = WhileStmt(Expr eAST, Stmt stmtAST)
     = ForStmt (Expr e1AST, Expr e2AST, Expr e3AST,
     Stmt stmtAST)
     = ReturnStmt(Expr eAST)
     = AssignStmt(Expr lAST, Expr rAST)
  + Expr
     =BinaryExpr(Expr lAST, Operator oAST, Expr rAST)
     =UnaryExpr(Operator oAST, Expr eAST)
     ....
  + Terminal
     =ID(String Lexeme)
     =Operator(String Lexeme)
     =IntLiteral(String Lexeme)
     =FloatLiteral(String Lexeme)
     ...
```

AST:



- ASTs are made of objects from the AST class hierarchy.
- We need **operations** that work **on ASTs**:
  - to compute the value of an AST expression,
  - to determine the type of an AST expression,
  - to generate byte-code from ASTs,
  - to print ASTs (like in the figure above), ...

How do we implement operations on ASTs?

Main question: given a class hierarchy like the MiniC AST hierarchy, how can we add operations to this hierarchy?

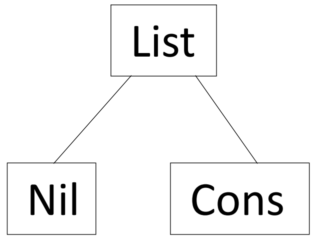
On the next slides, we will discuss 3 ways to do this:

1) in a non-object-oriented way.
2) in an object-oriented way, by changing every class in the hierarchy.
3) in an object-oriented way, without touching any class in the hierarchy.
   - using a so-called design pattern.
   - specifically, the Visitor design pattern
   - the Visitor design pattern will allow us to implement operations on ASTs without touching the AST class hierarchy.

But first we discuss *dispatching* method calls...

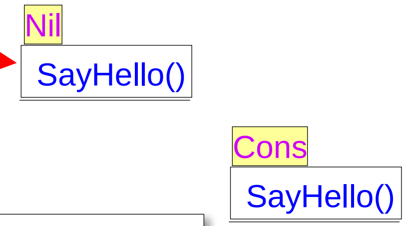
```
interface List {
 void SayHello();
}

class Nil implements List{
   public void SayHello() {
     System.out.println("I am a Nil...");
   }
}

class Cons implements List {
   public void SayHello() {
     System.out.println("I am a Cons...");
   }
}
```

Class hierarchy:



Run-time objects:

- Have a "**tag**" that tells the actual type



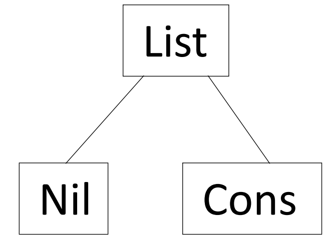

```
List L;
L = new Nil();
L.SayHello();  // "I am a Nil..."
L = new Cons();
L.SayHello();  // "I am a Cons..."
```

L has the type of the base class, i.e., "List". Depending on the concrete object, the call to SayHello dispatches to the method of the actual subclass (Nil or Cons)!

If you don't think this is terrific...

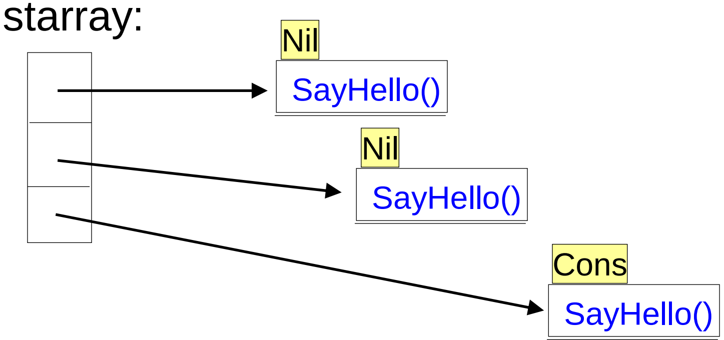
```
List[] listarray = new List[3];

for (int i=0; i<3; i++) {
  if (Math.random() > 0.5)
    listarray[i] = new Nil();
  else
    listarray[i] = new Cons();
}

for (int i=0; i<3; i++) {
  listarray[i].SayHello();
}
```

Class hierarchy:



Run-time objects:

listarray:



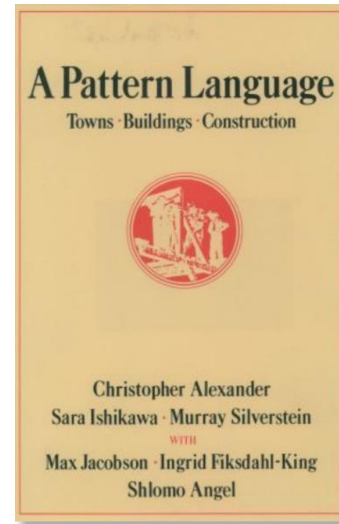
Listarray gets initialized at run-time. Based on a random value, either a Nil or a Cons object is created. The Java run-time system determines at <u>run-time</u> whether listarray[i] points to a Nil or Cons object. The call to SayHello() is dispatched accordingly.

Output:

```
I'm a Nil...
I'm a Nil...
I'm a Cons
```

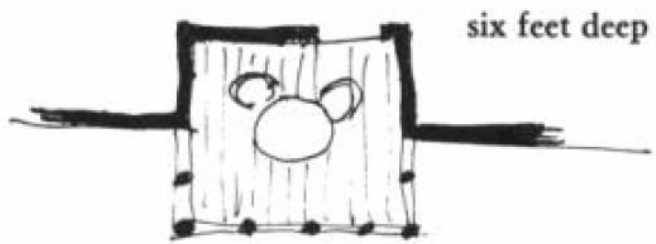
Patterns in Architecture

- Berkeley Professor Christopher Alexander
- In 1977, patterns for city planning, landscaping and architecture.
- To capture principles of “living” design.



Therefore:

Whenever you build a balcony, a porch, a gallery, or a terrace always make it at least six feet deep. If possible, recess at least a part of it into the building so that it is not cantilevered out and separated from the building by a simple line, and enclose it partially.



Design Patterns...

- ... provide general reusable solutions to commonly occurring problems in software design.
- ... are a description or template for how to solve a certain kind of problem.
- Object-oriented design patterns show relationships and interactions between classes or objects.
- Introduction to design patterns:

- http://en.wikipedia.org/wiki/Design_pattern_(computer_science)

The Visitor Design Pattern

``For object-oriented programming,

the Visitor pattern enables

the definition of a ***new operation***

on an object structure

without changing the classes

of the objects.''

Gamma, Helm, Johnson, Vlissides:

Design Patterns, 1995

Sneak Preview:

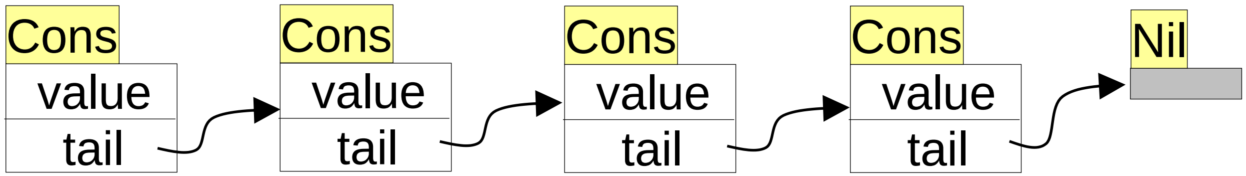
When using the **Visitor** pattern,

- the set of classes must be fixed in advance, and
- each class must have an accept() method.

Java example: summing an integer list.

```
interface List {}

class Nil implements List{}

class Cons implements List {
   public int value;
   public List tail;
}
```

An interface is an abstract type that specifies the methods that classes have to support.

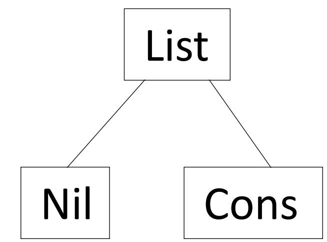


On the following slides, we'll discuss three approaches to compute the sum over this integer list.

First Approach: Instanceof and TypeCasts

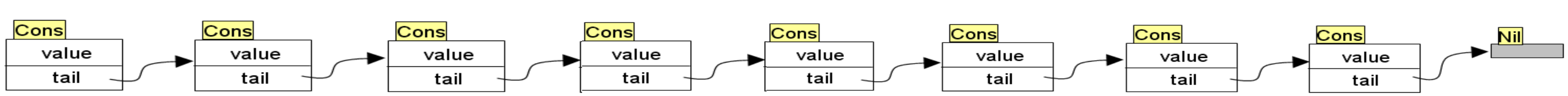
```
List L = ...;
int sum = 0;
boolean proceed = true;
while (proceed) {
 if (L instanceof Nil)
    proceed = false;
 else if (L instanceof Cons) {
    sum = sum + ((Cons) L).value; // Type-cast 1
    L = ((Cons) L).tail;          // Type-cast 2
 }
}
System.out.println("Sum: " + sum);
```

Class hierarchy:



Advantage: code is written without touching the classes Nil and Cons.

Drawback: code uses typecasts and instanceof to determine what class of object it is considering.



Second Approach: Dedicated Methods

- The first approach is not object-oriented (typecasts!).
- To access parts of an object, the “classical” approach is to use dedicated methods which both access and act on sub-classes.

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

- We can now compute the sum of a given List-object L by writing

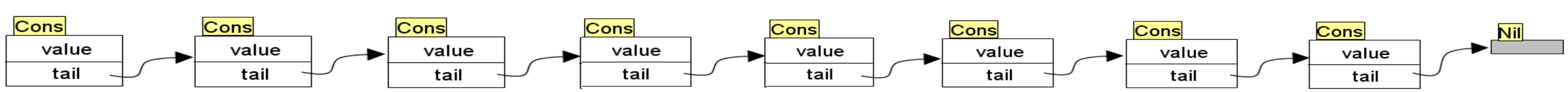
```
L.sum();
```

Second Approach: Dedicated Methods

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

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

List L = ...;
int sum = L.sum();
```

tail.sum() invokes the sum() method of the next list element. It will return the sum of the remainder of the list.

- **Advantage:** the typecasts and instanceof operations have disappeared. The code is written in a systematic way.
- **Disadvantage:** For each new operation on List objects, a new dedicated method has to be added to each class, and all classes must be recompiled.

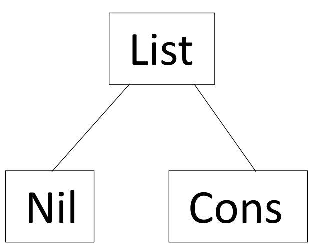


Third Approach: The Visitor Pattern

The idea:

- Divide the code into class hierarchy and a **visitor** (see Slide 17).
  - The visitor will implement the new operation to be added to the class hierarchy.
- Insert `accept()` method in each class.
  - Each `accept()` method takes a **visitor** as argument.
  - Note: a **single** accept method per class suffices for all possible visitors!
- A visitor contains a `visit()` method for each class (overloading!).
  - `visit()` methods implement functionality that we add to class hierarchy
    - Example: `sum()`
  - A `visit()` method for class foo contains an argument of type foo.

Class hierarchy:




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

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

Third Approach: The Visitor Pattern

- The purpose of the **`accept()`** methods is to invoke the **`visit()`** method in the visitor which knows how to handle the current object.

static overload resolution on visit()

dispatch on v

```
class Nil implements List {
   public void accept (Visitor v) {
      v.visit (this); //dispatches on v to v.visit(Nil x)
   }
}

class Cons implements List {
   public int value;
   public List tail;

   public void accept (Visitor v) {
      v.visit (this); //dispatches to v.visit(Cons x)
   }
}
```

'this' is a reference to the *current object*

Note: even if we add N operations to our class hierarchy, a single accept() method per class is sufficient!

```
class Cons implements List {
   public int value;
   public List tail;

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

```
;; JVM bytecode for Cons.accept:
;;
.method public accept(LVisitor;)V
  .limit stack 2
  .limit locals 2
  .line 11
  0: aload_1          ; will dispatch on visitor argument
  1: aload_0          ; this
  2: invokeinterface Visitor/visit(LCons;)V 2
  7: return
.end method
```

visitor argument

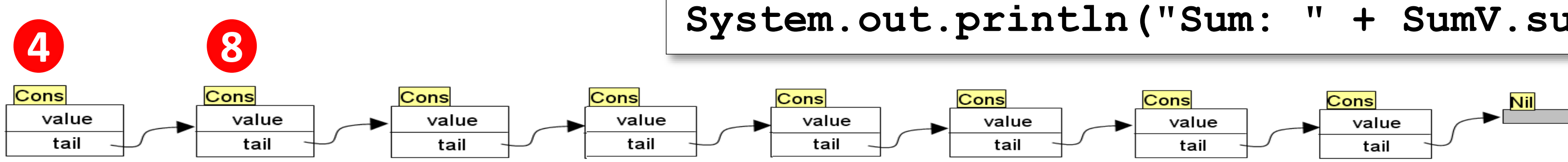
Third Approach: The Visitor Pattern

- The control flow goes back and forth between the **`visit()`** methods in the Visitor and the **`accept()`** methods in the object structure.

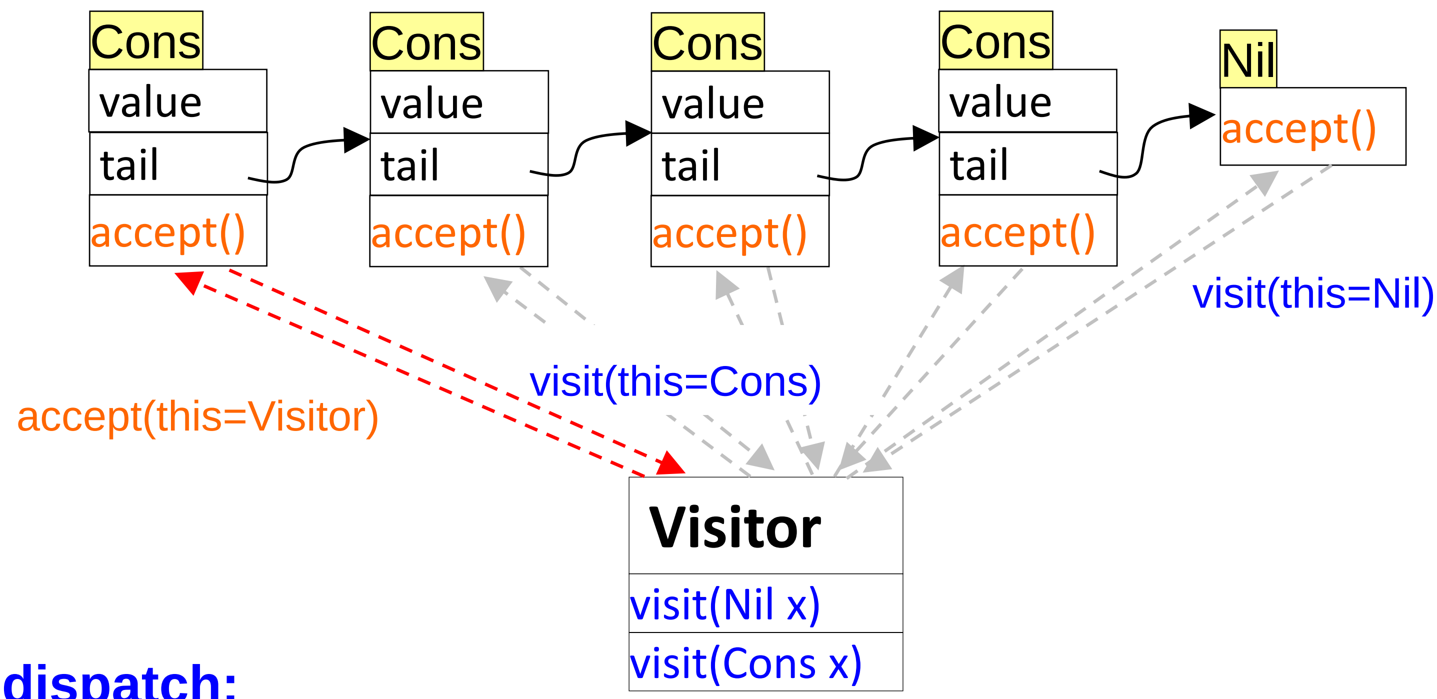
```
class Nil implements List {
  public void accept (Visitor v) {
    //dispatch to v.visit(Nil x)
    v.visit (this); }
}

class Cons implements List {
  public int value;
  public List tail;

  public void accept (Visitor v) {
    //dispatch to v.visit(Cons x)
    v.visit (this); }
}
```

```
class SumVisitor implements Visitor {

   int sum = 0;

   public void visit (Nil x)   {}

   public void visit (Cons x) {
      sum = sum + x.value; // do the work
      x.tail.accept(this) //go to next element
   }
}
```

```
List L = ...; // some list again.

SumVisitor SumV = new SumVisitor();
L.accept (SumV);
System.out.println("Sum: " + SumV.sum);
```



Third Approach: The Visitor Pattern



Double dispatch:

- The visitor calls the **`accept()`** method which dispatches to the proper object in the List class-hierarchy.
- The **`List`** object uses the visitor argument provided with **`accept()`** to call **`visit()`**.
  - **`visit()`** dispatches to the proper object in the visitor class-hierarchy.

Third Approach: The Visitor Pattern

- We can implement **further visitor classes** which perform additional operations on our class hierarchy:

```
class PrintVisitor implements Visitor {

   public void visit (Nil x) {
      System.out.println ("I'm a Nil list node...");
   }

   public void visit (Cons x) {
      System.out.println ("I'm a Cons list node, value "
                                    + x.value);
      x.tail.accept(this);
   }

}
```

```
List L = ...;

PrintVisitor PrintV = new PrintVisitor();
L.accept (PrintV);
```

Comparison

The Visitor pattern combines the advantages of the other two approaches:

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

The **advantage** of visitors: new methods without recompilation!

Requirements for using visitors: all classes must have an accept() method.

Visitors: Summary

- **Visitors make adding new operations easy.** Simply write a new visitor.
- **A visitor gathers related operations.** It separates unrelated operations.
- **Adding new classes to the class hierarchy is hard** (all visitors must be updated as well).
  - Key consideration: are you likely to change the algorithms applied over a class hierarchy, or are you more likely to change the classes of your class hierarchy?
- **Visitors can accumulate state** (like the sum variable in the running example).
- **Visitors 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. *This may compromise encapsulation.*
- **Double dispatch ~ double run-time overhead.**
  Only relevant if SW is performance critical.

MiniC Visitors

Our MiniC framework makes extensive use of the Visitor pattern:

- All non-abstract AST classes provide own accept() method.
  - TreeDrawer,
  - Unparser, and
  - TreePrinter

  are Visitors of the AST class hierarchy. They work on an AST object structure to do their work.

We will use Visitors

- for semantic analysis
- for code generation.