Object-Oriented Versus Functional Programming; The Visitor Pattern

April 14–16, 2014

Functional vs OO

```
interface Animal {
    String happyNoise();
    String excitedNoise();
}

class Dog implements Animal {
    String happyNoise() { return "pant pant"; }
    String excitedNoise() { return "bark"; }
}

class Cat implements Animal {
    String happyNoise() { return "purrrrr"; }
    String excitedNoise() { return "meow"; }
}
```

Functional vs OO

```
class Chicken implements Animal {
    String happyNoise() { return "cluck cluck"; }
    String excitedNoise() { return "cockadoodledoo"; }
}
```

Functional vs OO

```
interface Animal {
    String happyNoise();
    String excitedNoise();
    String angryNoise();
}

class Dog implements Animal {
    String happyNoise() { return "pant pant"; }
    String excitedNoise() { return "bark"; }
    String angryNoise() { return "grrrrr"; }
}

class Cat implements Animal {
    String happyNoise() { return "purrrrr"; }
    String excitedNoise() { return "meow"; }
    String angryNoise() { return "hissss"; }
}
```

Functional vs OO

```
datatype Animal = Dog | Cat ;

fun happyNoise(Dog) = "pant pant"
  | happyNoise(Cat) = "purrrr"

fun excitedNoise(Dog) = "bark"
  | excitedNoise(Cat) = "meow"
```

Functional vs OO

```
fun angryNoise(Dog) = "grrrrr"
  | angryNoise(Cat) = "hissss"
```

Functional vs OO

```
datatype Animal = Dog | Cat | Chicken;

fun happyNoise(Dog) = "pant pant"
    | happyNoise(Cat) = "purrrr"
    | happyNoise(Chicken) = "cluck cluck";

fun excitedNoise(Dog) = "bark"
    | excitedNoise(Cat) = "meow"
    | excitedNoise(Chicken) = "cockadoodledoo";

fun angryNoise(Dog) = "grrrrr"
    | angryNoise(Cat) = "hisssss"
    | angryNoise(Chicken) = "squaaaack";
```

Functional vs OO

	Dog	Cat	Chicken
happyNoise	pant pant	purrrr	cluck cluck
excitedNoise	bark	meow	cockadoodledoo
angryNoise	grrrrr	hisssss	squaaaaack

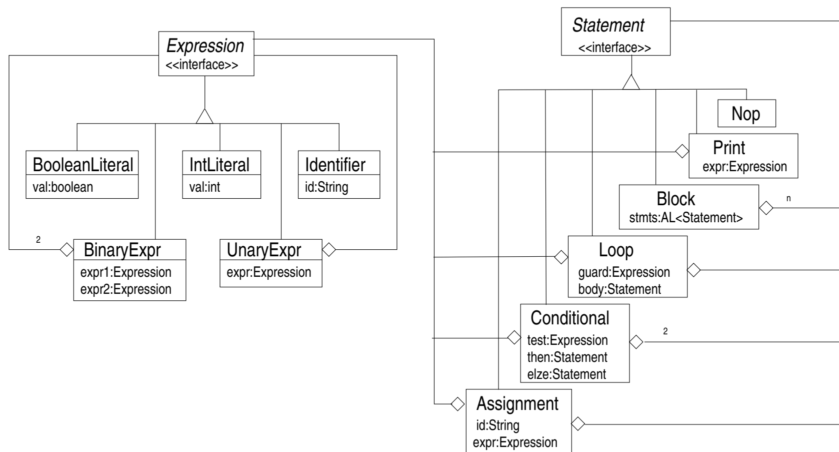
MilliJava grammar

$$
\begin{array}{rcl}
\mathit{Program} & \rightarrow & \text{public class } \mathit{ID} \\
& & \{ \text{ public static void main(String[] args)} \\
& & \{ \ \mathit{Declaration}* \quad \mathit{Statement}* \ \} \ \} \\
\mathit{Declaration} & \rightarrow & \mathit{Type} \ \mathit{ID} \ ; \\
\mathit{Type} & \rightarrow & \text{int} \ \mid \text{boolean} \\
\mathit{Statement} & \rightarrow & \mathit{Assignment} \ \mid \mathit{Conditional} \ \mid \mathit{Loop} \ \mid \mathit{Block} \\
& & \mid \mathit{Print} \ \mid \mathit{Nop} \\
\mathit{Assignment} & \rightarrow & \mathit{ID} = \mathit{Expression} \ ; \\
\mathit{Conditional} & \rightarrow & \text{if (} \mathit{Expression} \text{) } \mathit{Statement} \text{ else } \mathit{Statement} \\
\mathit{Loop} & \rightarrow & \text{while (} \mathit{Expression} \text{) } \mathit{Statement} \\
\mathit{Block} & \rightarrow & \{ \text{ Statement } *\} \\
\mathit{Print} & \rightarrow & \text{System.out.println(} \mathit{Expression} \text{) ;} \\
\mathit{Nop} & \rightarrow & ;
\end{array}
$$

MilliJava grammar

$$
\begin{array}{rcl}
\mathit{Expression} & \rightarrow & \mathit{BoolLiteral} \mid \mathit{IntLiteral} \mid \mathit{Variable} \\
 & & \mid \mathit{BinaryExpression} \mid \mathit{UnaryExpression} \\
\mathit{Variable} & \rightarrow & \mathit{Identifier} \\
\mathit{BinaryExpression} & \rightarrow & (\ \mathit{Expression}\ \mathit{BinaryOperator}\ \mathit{Expression}\ ) \\
\mathit{UnaryExpression} & \rightarrow & (\ \mathit{UnaryOperator}\ \mathit{Expression}\ ) \\
\mathit{BinaryOperator} & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{||} \mid \texttt{\&\&} \\
 & & \mid \texttt{<} \mid \texttt{>} \mid \texttt{==}
\end{array}
$$

Interpreter/Composite Pattern



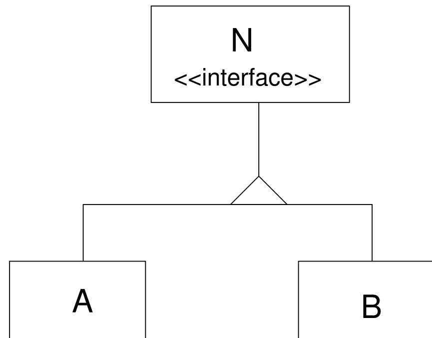
Structures and operations

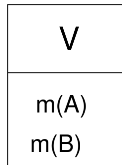
	Declaration	Assignment	Conditional	...	BinaryExpr	IntLit	...
TypeCheck							
Compile							
Optimize							
PrettyPrint							

Our goal

Separate the structure/data from functionality so that we can add new functionality by writing one new "module" rather than modifying several.

Visitor: General problem



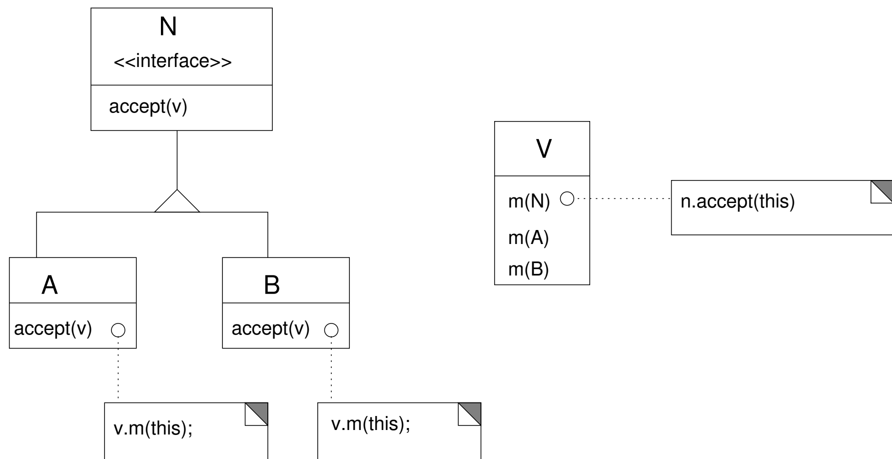


Visitor: Intent

> *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.*

DP, pg 331.

Visitor: Structure



Subtying the Visitors