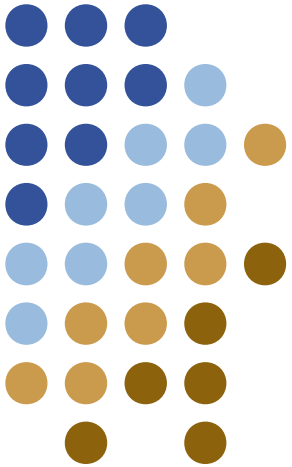
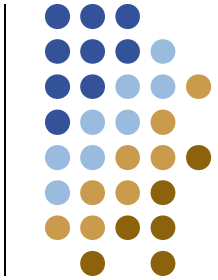
Compilers

Visitor pattern for compilers

Yannis Smaragdakis, U. Athens

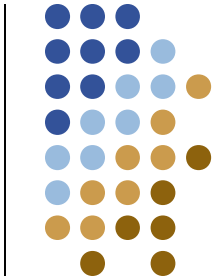


For Your Project



- You will use a parser generator: javacc
  - the Java Compiler Compiler
  - LL(k)
  - single file with lexer and parser specification
- You will also use the Java Tree Builder (JTB)
  - JTB is a front-end for javacc
    - first pass a grammar file through JTB, get back Java code that forms the skeleton of a compiler
      - Java grammar that builds a syntax tree
      - one class for every form of syntax tree node
      - a default visitor for traversals of the tree

What Is a Visitor?



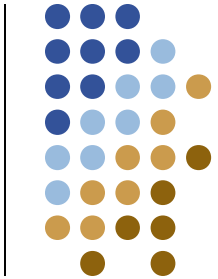
- Visitor pattern: a well-known design pattern
  - “programming functionally in an OO language”
- For encoding operations as independent entities, without distributing them throughout classes
  - useful for adding functionality without editing classes
- In OO designs if an “operation” is applicable to multiple types (classes), it is defined as a method in all the corresponding classes. With visitor, the “operation” can be in a class by itself

Example from Compilers

Consider our syntax tree having nodes of type `STNode`

```
class STNode { }
class Meth extends STNode {
  Decl signature;
  Stmt [] stmts;
}
class Stmt extends STNode { }
class IfStmt extends Stmt {
  Expr cond;
  Stmt thenClause, elseClause;
 …
}
class Expr extends STNode {…} …
```

Example from Compilers

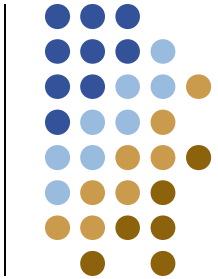


OO approach to adding operations:

```
class STNode { boolean typeCheck() {…} }
class Meth extends STNode { …
  boolean typeCheck() {…}
}
class Stmt extends STNode { …
  boolean typeCheck() {…}
}
class IfStmt extends Stmt { …
  boolean typeCheck() { …
   cond.typeCheck();… thenStmt.typeCheck();
 }
}
…
```

Problem: adding a new operation requires changing all classes

Another (not good) Approach



Type unsafe approach to adding operations:

```
boolean typeCheck(STNode n) { …
  if (n instanceof IfStmt) {
    IfStmt s = (IfStmt) n;
    … typeCheck(s.cond)
    … typeCheck(s.thenStmt)
  } else if (n instanceof Expr) {
    …
  }
  …
}
```

Visitor Pattern

Every class has a stylized “accept” method, there is a separate hierarchy of visitors

```
class STNode {
  void accept(Visitor v) {
    v.visit(this);
  }
}
class Meth extends STNode { …
  void accept(Visitor v) {
    v.visit(this);
  }
}
…
```

Visitor Pattern

Every class has a stylized “accept” method, there is a separate hierarchy of visitors

```
abstract class Visitor {
  abstract void visit(STNode v);
  abstract void visit(Meth m);
  abstract void visit(Stmt s); …
}
class TypeCheckVisitor extends Visitor {
  void visit(STNode v) { … }
  void visit(IfStmt is) { …
    is.cond.accept(this); …
  } …
}
```