

Design patterns (part 3)

CSE 403

University of Washington

Michael Ernst

Outline

- ✓ Introduction to design patterns
- ✓ Creational patterns (constructing objects)
- ✓ Structural patterns (controlling heap layout)
- ⇒ Behavioral patterns (affecting object semantics)

Review: Composite pattern

Composite permits a client to manipulate either an **atomic** unit or a **collection** of units in the same way

Good for dealing with part-whole relationships

Example behavior: traversing composites

- Goal: perform operations on all parts of a composite
- Idea: generalize the notion of an iterator

Composite example 1: Bicycle

- Bicycle
 - Frame
 - Drivetrain
 - Wheel
 - Tire
 - Tube
 - Tape
 - Rim
 - Nipples
 - Spokes
 - Hub
 - Skewer
 - Lever
 - Body
 - Cam
 - Rod
 - Acorn nut
 - ...

Methods on components

```
abstract class BicycleComponent {  
    int weight();  
    float cost();  
}
```

```
class Wheel  
    extends BicycleComponent {  
    float assemblyCost;  
    Skewer skewer;  
    Hub hub;  
    ...  
    float cost() {  
        return assemblyCost  
            + skewer.cost()  
            + hub.cost()  
            + ...;  
    }  
}
```

```
class Bicycle  
    extends BicycleComponent {  
    float assemblyCost;  
    Frame frame;  
    Drivetrain drivetrain;  
    Wheel frontWheel;  
    ...  
    float cost() {  
        return assemblyCost  
            + frame.cost()  
            + drivetrain.cost()  
            + frontWheel.cost()  
            + ...;  
    }  
}
```

Composite example 2: Libraries

Library

Section (for a given genre)

Shelf

Volume

Page

Column

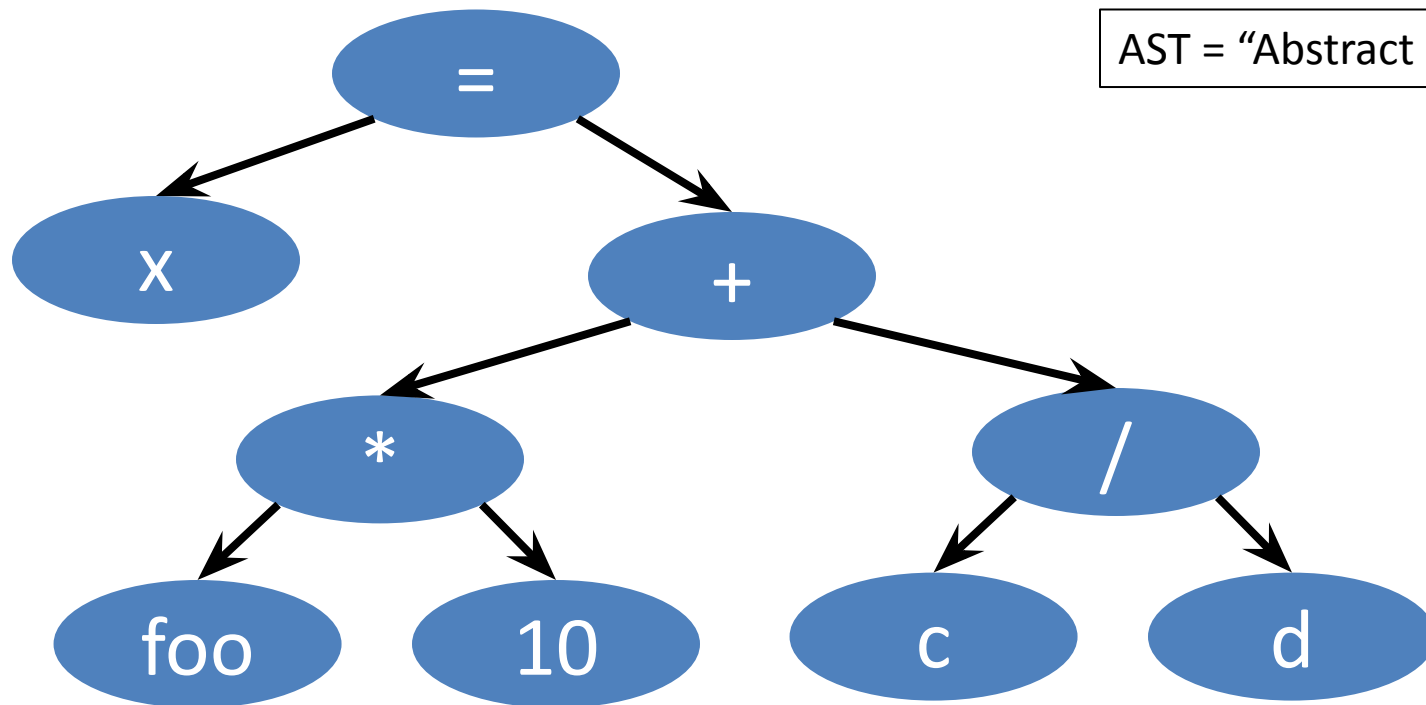
Word

Letter

```
interface Text {  
    String getText();  
}  
class Page implements Text {  
    String getText() {  
        ... return the concatenation of the column texts ...  
    }  
}
```

Composite example 3: Representing Java code

```
x = foo * 10 + c / d;
```



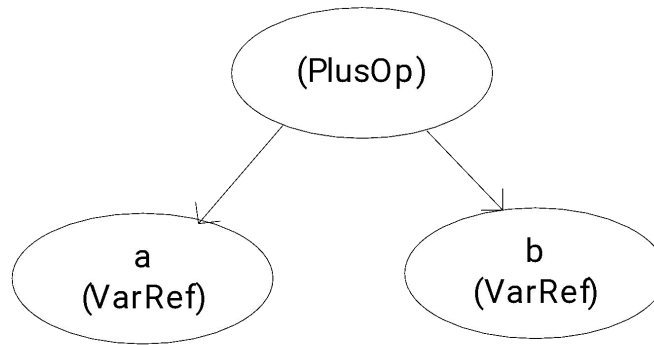
AST = “Abstract Syntax Tree”

Abstract syntax tree (AST) for Java code

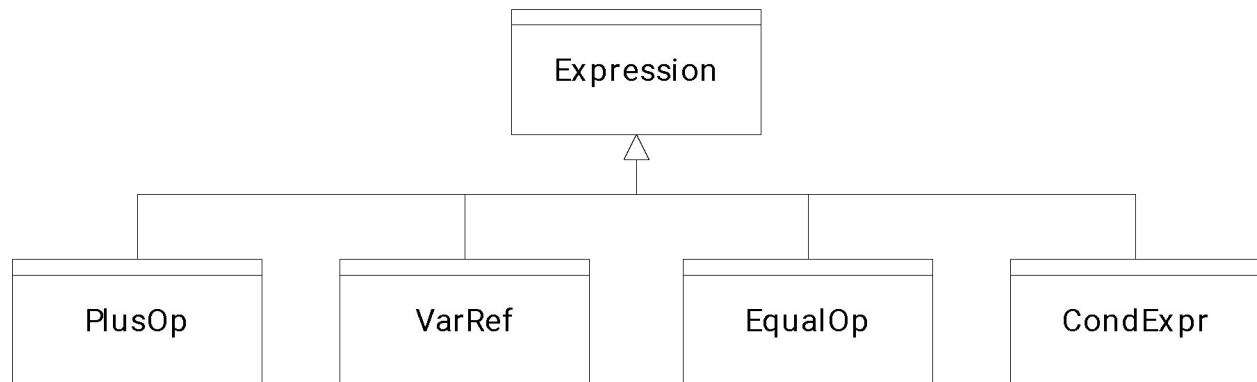
```
class PlusOp extends Expression {    // + operation
    Expression leftExp;
    Expression rightExp;
}
class VarRef extends Expression {    // variable reference
    String varname;
}
class EqualOp extends Expression {    // equality test a==b
    Expression lvalue;    // left-hand side: "a" in "a==b"
    Expression rvalue;    // right-hand side: "b" in "a==b"
}
class CondExpr extends Expression {    // a?b:c
    Expression condition;
    Expression thenExpr;    // value of expression if a is true
    Expression elseExpr;    // value of expression if a is false
}
```

Object model (heap layout) vs. type hierarchy (subtyping)

- AST for "a + b":



- Class hierarchy for **Expression**:



Operations on Java expressions

Java expressions

- Plus: `20 + 22`
- Variable reference: `x`
- Method call: `foo(5)`
- Conditional: `b ? 3 : 4`
- Equals: `x == y`
- ...

Operations

- Typecheck
- Pretty-print
- Optimize
- ...

Type-checking an equality operation

```
// typecheck "a == b" where a and b are arbitrary
expressions
Type tcEqualsOp(EqualOp e) {
    Type leftType = tcExpression(e.lvalue); // type of a
    Type rightType = tcExpression(e.rvalue); // type of b
    if (leftType != ERROR_TYPE
        && rightType != ERROR_TYPE
        && leftType == rightType) {
        return BOOLEAN_TYPE;
    } else {
        return ERROR_TYPE;
    }
}
```

Type-checking a conditional expr

```
// typecheck "a ? b : c"
```

```
Type tcCondExpr(CondExpr e) {  
    Type condType = tcExpression(e.condition); // type of a  
    Type thenType = tcExpression(e.thenExpr);  // type of b  
    Type elseType = tcExpression(e.elseExpr);  // type of c  
    if (condType != ERROR_TYPE  
        && thenType != ERROR_TYPE  
        && elseType != ERROR_TYPE  
        && condType == BOOLEAN_TYPE  
        && thenType == elseType) {  
        return thenType;  
    } else {  
        return ERROR_TYPE;  
    }  
}
```

Organizing operations on abstract syntax trees

We know how to write code in each cell of this table:

		Objects	
		CondExpr	EqualOp
Operations	typecheck		
	pretty-print		

Question: Should we group together the code for a particular operation or the code for a particular expression?

(A separate issue: given an operation and an expression, how to select the proper piece of code to execute?)

Interpreter and procedural patterns

Interpreter: collects code for similar **objects**, spreads apart code for similar operations

Easy to add objects

Hard to add operations

An instance of the composite pattern

Procedural: collects code for similar **operations**, spreads apart code for similar objects

Easy to add operations

Hard to add objects

The visitor pattern is a variety of the procedural pattern

OO style

	Objects	
	CondExpr	EqualOp
typecheck		
pretty-print		

Functional style

	Objects	
	CondExpr	EqualOp
typecheck		
pretty-print		

Interpreter pattern

Objects		
	CondExpr	EqualOp
typecheck		
pretty-print		

Add a method to each class for each supported operation

```
class Expression {  
    ...  
    Type typecheck();  
    String prettyPrint();  
}
```

```
class EqualOp extends Expression { // "a == b"  
    ...  
    Type typecheck() { ... }  
    String prettyPrint() { ... }  
}
```

```
class CondExpr extends Expression { // "a?b:c"  
    ...  
    Type typecheck() { ... }  
    String prettyPrint() { ... }  
}
```

Dynamic dispatch chooses the right implementation, for a call like `myExpr.typeCheck()`

Procedural pattern

Objects		
	CondExpr	EqualOp
typecheck		
pretty-print		

Create a class per operation, with a method per operand type

```
class Typecheck {
  // typecheck "a?b:c"
  Type tcCondExpr(CondExpr e) {
    Type condType = tcExpression(e.condition); // type of "a"
    Type thenType = tcExpression(e.thenExpr);  // type of "b"
    Type elseType = tcExpression(e.elseExpr);  // type of "c"
    if ((condType == BoolType) && (thenType == elseType)) {
      return thenType;
    } else {
      return ErrorType;
    }
  }

  // typecheck "a==b"
  Type tcEqualOp(EqualOp e) {
    ...
  }
}
```

Given an expression,
how to invoke the
right implementation?

Definition of tcExpression

(in procedural pattern)

```
class Typecheck {  
    ...  
    Type tcExpression(Expression e) {  
        if (e instanceof PlusOp) {  
            return tcPlusOp((PlusOp)e);  
        } else if (e instanceof VarRef) {  
            return tcVarRef((VarRef)e);  
        } else if (e instanceof EqualOp) {  
            return tcEqualOp((EqualOp)e);  
        } else if (e instanceof CondExpr) {  
            return tcCondExpr((CondExpr)e);  
        } else ...  
        ...  
    }  
}
```

Maintaining this code is tedious and error-prone.

The cascaded if tests may to run slowly.

Code duplication: this code is repeated in PrettyPrint and every other operation class.

(Languages with pattern matching mitigate these problems.)

Visitor pattern:

a variant of the procedural pattern

Visitor encodes a traversal of a hierarchical data structure

Nodes (objects in the hierarchy) accept visitors

Visitors visit nodes (objects)

```
class SomeExpression extends Expression {  
    void accept(Visitor v) {  
        for each child of this node {  
            child.accept(v);  
        }  
        v.visit(this);  
    }  
}  
class SomeVisitor {  
    void visit(SomeExpression n) {  
        perform work on n  
    }  
    void visit(OtherExpression n) {  
        perform work on n  
    }  
}
```

`n.accept(v)` traverses the structure rooted at `n`, invoking `v`'s operation

`visit` does the work on one element

What happened to all the
instanceof operations?

Visitor pattern: example

Objects

```
class VarOp extends Expression {
    void accept(Visitor v) {
        v.visit(this);
    }
}

class EqualsOp extends Expression {
    void accept(Visitor v) {
        leftExp.accept(v);
        rightExp.accept(v);
        v.visit(this);
    }
}

class CondOp extends Expression {
    void accept(Visitor v) {
        testExp.accept(v);
        thenExp.accept(v);
        elseExp.accept(v);
        v.visit(this);
    }
}
```

Operations

```
class TypeCheckVisitor implements
    Visitor {
    void visit(EqualsOp e) {
        assert e.leftExp and e.rightExp
        have the same type
    }
    void visit(PlusOp e) {
        assert e.leftExp and e.rightExp
        are numeric
    }
}
```

Each operation has its cases together.

Compiler error if a case is missing

```
class PrettyPrintVisitor implements
    Visitor {
    void visit(EqualsOp e) {
        print e
    }
    void visit(PlusOp e) {
        print e
    }
}
```

Visitor pattern: example

Objects

```
class VarOp extends Expression {
    void accept(Visitor v) {
        v.visit(this);
    }
}

class EqualsOp extends Expression {
    void accept(Visitor v) {
        leftExp.accept(v);
        rightExp.accept(v);
        v.visit(this);
    }
}

class CondOp extends Expression {
    void accept(Visitor v) {
        testExp.accept(v);
        thenExp.accept(v);
        elseExp.accept(v);
        v.visit(this);
    }
}
```

Overloading

Operations

```
class TypeCheckVisitor implements
    Visitor {
    void visit(EqualsOp e) {
        assert e.leftExp and e.rightExp
            have the same type
    }
    void visit(PlusOp e) {
        assert e.leftExp and e.rightExp
            are numeric
    }
}

class PrettyPrintVisitor implements
    Visitor {
    void visit(EqualsOp e) {
        print e
    }
    void visit(PlusOp e) {
        print e
    }
}
```

Visitor pattern: example

Objects

```
class VarOp extends Expression {
    void accept(Visitor v) {
        v.visitVar(this);
    }
}

class EqualsOp extends Expression {
    void accept(Visitor v) {
        leftExp.accept(v);
        rightExp.accept(v);
        v.visitEquals(this);
    }
}

class CondOp extends Expression {
    void accept(Visitor v) {
        testExp.accept(v);
        thenExp.accept(v);
        elseExp.accept(v);
        v.visitCond(this);
    }
}
```

Same behavior

Operations

```
class TypeCheckVisitor implements
    Visitor {
    void visitEquals(EqualsOp e) {
        assert e.leftExp and e.rightExp
            have the same type
    }
    void visitPlus(PlusOp e) {
        assert e.leftExp and e.rightExp
            are numeric
    }
}

class PrettyPrintVisitor implements
    Visitor {
    void visitEquals(EqualsOp e) {
        print e
    }
    void visitPlus(PlusOp e) {
        print e
    }
}
```

Sequence of calls to accept and visit

a.accept(v)

 b.accept(v)

 d.accept(v)

 v.visit(d)

 e.accept(v)

 v.visit(e)

 v.visit(b)

 c.accept(v)

 f.accept(v)

 v.visit(f)

 v.visit(c)

 v.visit(a)

d

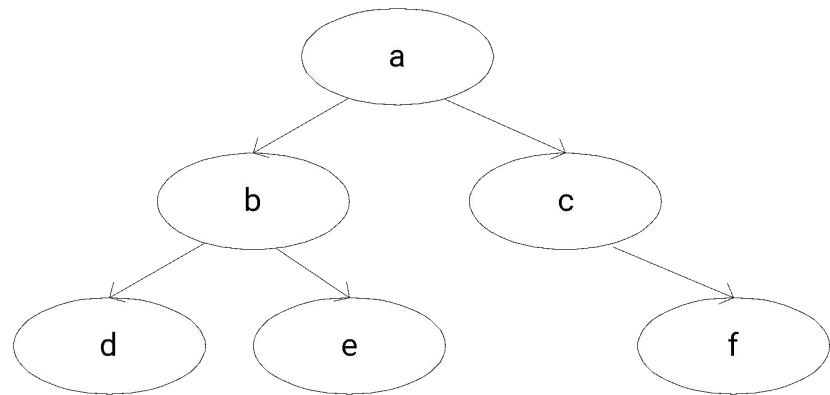
e

b

f

c

a



Sequence of calls to visit = depth-first (children before parents)

How could we make it breadth-first (parents before children)?

Requires a different visitor

Visitors are like iterators

- Each element of the data structure is presented in turn to the `visit` method
- Visitors have knowledge of the structure, not just the sequence

Calls to `visit` cannot communicate with one another

Ways to address this:

1. Use an auxiliary data structure
2. `visit` returns a value
3. Move more work into the visitor itself

```
class Node {  
    void accept(Visitor v) {  
        v.visit(this);  
    }  
}  
  
class Visitor {  
    void visit(Node n) {  
        for each child of this node {  
            child.accept(v);  
        }  
        perform work on n  
    }  
}
```

Information flow is clearer (if visitor depends on children)

Traversal code repeated in all visitors (acceptor is extraneous)