CSE 331
Software Design and Implementation

Lecture 21
Design Patterns 2

Leah Perlmutter / Summer 2018

Announcements

Announcements

- Guest Speaker Friday!!! <3
  - Kendra Yourtee
- Quiz 7 due Thursday 8/9
- Homework 8 due Thursday 8/9
  - HW8 has a regression testing component: HW5, 6, 7 tests must pass.

Outline

✓ Introduction to design patterns

✓ Creational patterns (constructing objects)

⇒ Structural patterns

- Behavioral patterns (affecting object semantics)

Wrappers

Structural patterns: Wrappers

Wrappers are a thin veneer over an encapsulated object

- Modify the interface
- Extend behavior
- Restrict access

We've seen this before!

- Wrappers can serve as an alternative to subtyping!

Terminology:

- Also called composition or delegation
- The wrapped or encapsulated object is called the delegate

Structural patterns: Wrappers

Wrappers can change the interface or functionality of the encapsulated object.

- The encapsulated object does most of the work

Different kinds of wrappers:

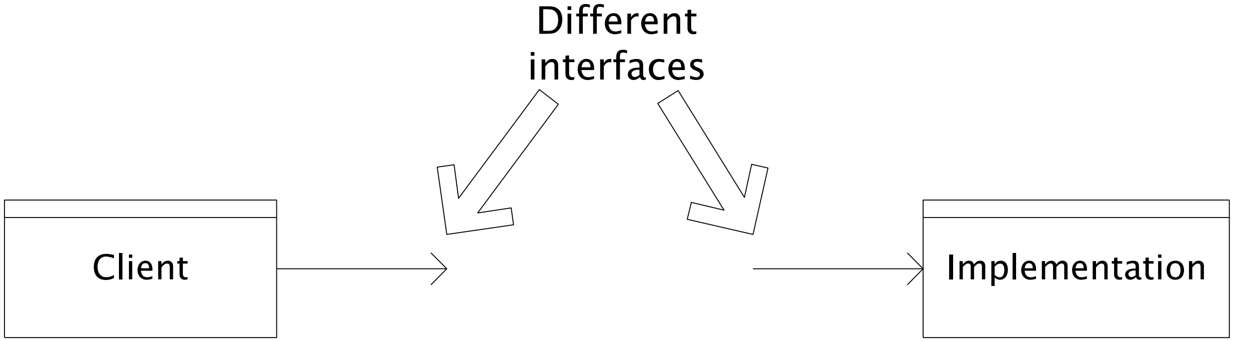
Pattern	**Functionality**	**Interface**
Adapter	**same**	**different**
Decorator	**different**	**same**
Proxy	**same***	**same**

*from client perspective

Some wrappers have qualities of more than one of adapter, decorator, and proxy

Adapter

- An adapter translates between incompatible interfaces
- Change an interface without changing functionality
  - Rename a method
  - Convert units
  - Implement a method in terms of another
- Example: angles passed in radians vs. degrees
- Example: use "old" method names for legacy code



Adapter example: scaling rectangles

We have this **`Rectangle`** interface

```
interface Rectangle {
   // grow or shrink this by the given factor
   void scale(float factor);
   ...
   float getWidth();
   float area();
}
```

Client interface

Goal: client code wants to use this library to "implement" **`Rectangle`** without rewriting code that uses **`Rectangle`**:

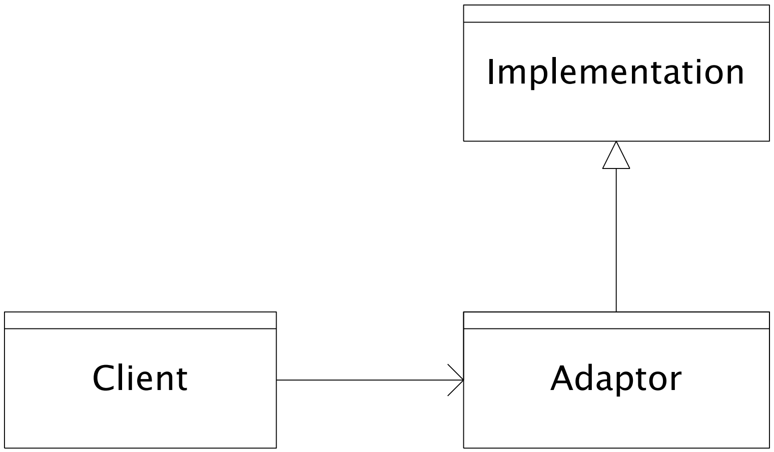
```
// doesn't implement Rectangle interface
class SimpleRectangle {
   void setWidth(float width) { ... }
   void setHeight(float height) { ... }
       // no scale method
   ...
   float getWidth() { ... }
   float area() { ... }
}
```

Implementation

needs adapter!

Adapter: Use subclassing

```
class ScaleableRectangle1
            extends SimpleRectangle
            implements Rectangle {
  void scale(float factor) {
    setWidth(factor * getWidth());
    setHeight(factor * getHeight());
  }
}
```



Adapter: use delegation

Delegation: forward requests to another object

```
class ScaleableRectangle2 implements Rectangle {
  SimpleRectangle r;
  ScaleableRectangle2(float w, float h) {
    this.r = new SimpleRectangle(w,h);
  }
  void scale(float factor) {
    r.setWidth(factor * r.getWidth());
    r.setHeight(factor * r.getHeight());
  }
  float getWidth() { return r.getWidth(); }
  float area() { return r.area(); }
  ...
}
```

Adapter: subclassing vs. delegation

Subclassing

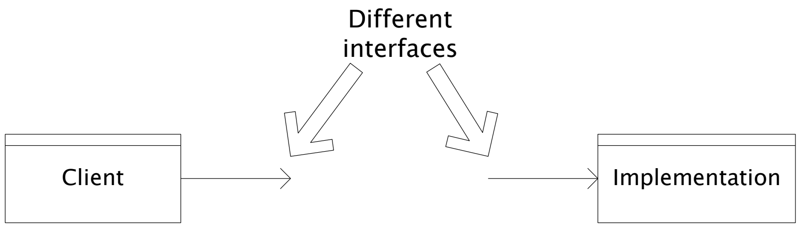
- automatically gives access to all methods of superclass
- built in to the language (syntax, efficiency)

Delegation

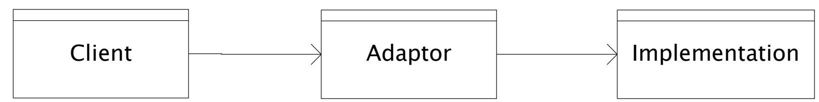
- also known as composition
- permits “removal” of methods
- objects of arbitrary concrete classes can be wrapped
  - don’t need to worry about true subytpes
- multiple wrappers can be composed

Types of adapter

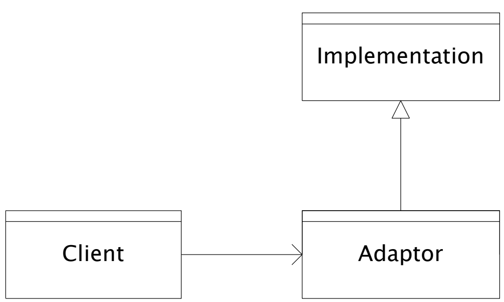
Goal of adapter:
connect incompatible interfaces



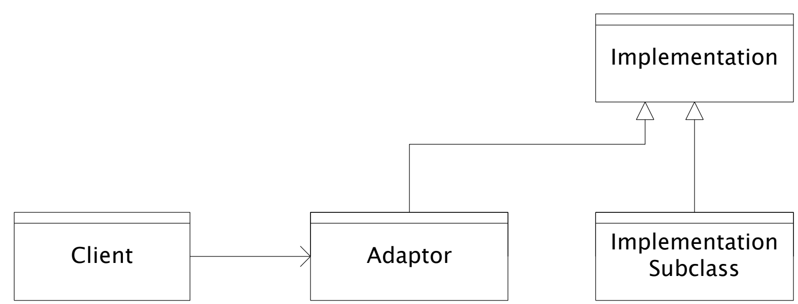
Adapter with delegation



Adapter with subclassing

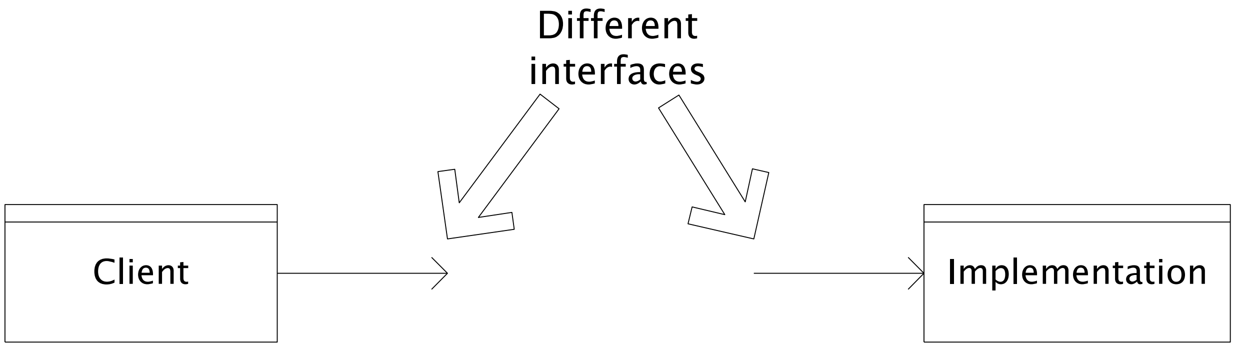


Adapter with subclassing:
no extension is permitted



Summary: Adapter

- An adapter translates between incompatible interfaces
- Can do it by subclassing the original implementation or by delegating to an object of the original implementation



Decorator

- Add functionality without changing the interface
- Add to existing methods to do something additional
  - (while still preserving the previous specification)
- More flexible than subclassing alone
- Note: decorator is a lot of things.
  - what we say about one kind of decorator might not apply to other kinds of decorator

Decorator from lecture on subtyping...

```
public class InstrumentedHashSet<E>
                                   extends HashSet<E> {
  private int addCount = 0;  // count # insertions
  public InstrumentedHashSet(Collection<? extends E> c){
     super(c);
  }
  public boolean add(E o) {
     addCount++;
     return super.add(o);
  }
  public boolean addAll(Collection<? extends E> c) {
     addCount += c.size();
     return super.addAll(c);
  }
  public int getAddCount() { return addCount; }
}
```

Problem: Unspecified whether **addAll** will call **add**. Don't know whether to increment count here.

Decorator from lecture on subtyping...

```
public class InstrumentedHashSet<E> {
  private final HashSet<E> s = new HashSet<E>();
  private int addCount = 0;
  public InstrumentedHashSet(Collection<? extends E> c){
      this.addAll(c);
  }
  public boolean add(E o) {
     addCount++;   return s.add(o);
  }
  public boolean addAll(Collection<? extends E> c) {
     addCount += c.size();
     return s.addAll(c);
  }
  public int getAddCount() {  return addCount; }
  // ... and every other method specified by HashSet<E>
}
```

Delegate

No longer calls InstrumentedHashSet's add method

Decorator example: Bordered windows

```
interface Window {
  // rectangle bounding the window
  Rectangle bounds();
  // draw this on the specified screen
  void draw(Screen s);
  ...
}

class WindowImpl implements Window {
  ...
}
```

Bordered window implementations

Via subclasssing:

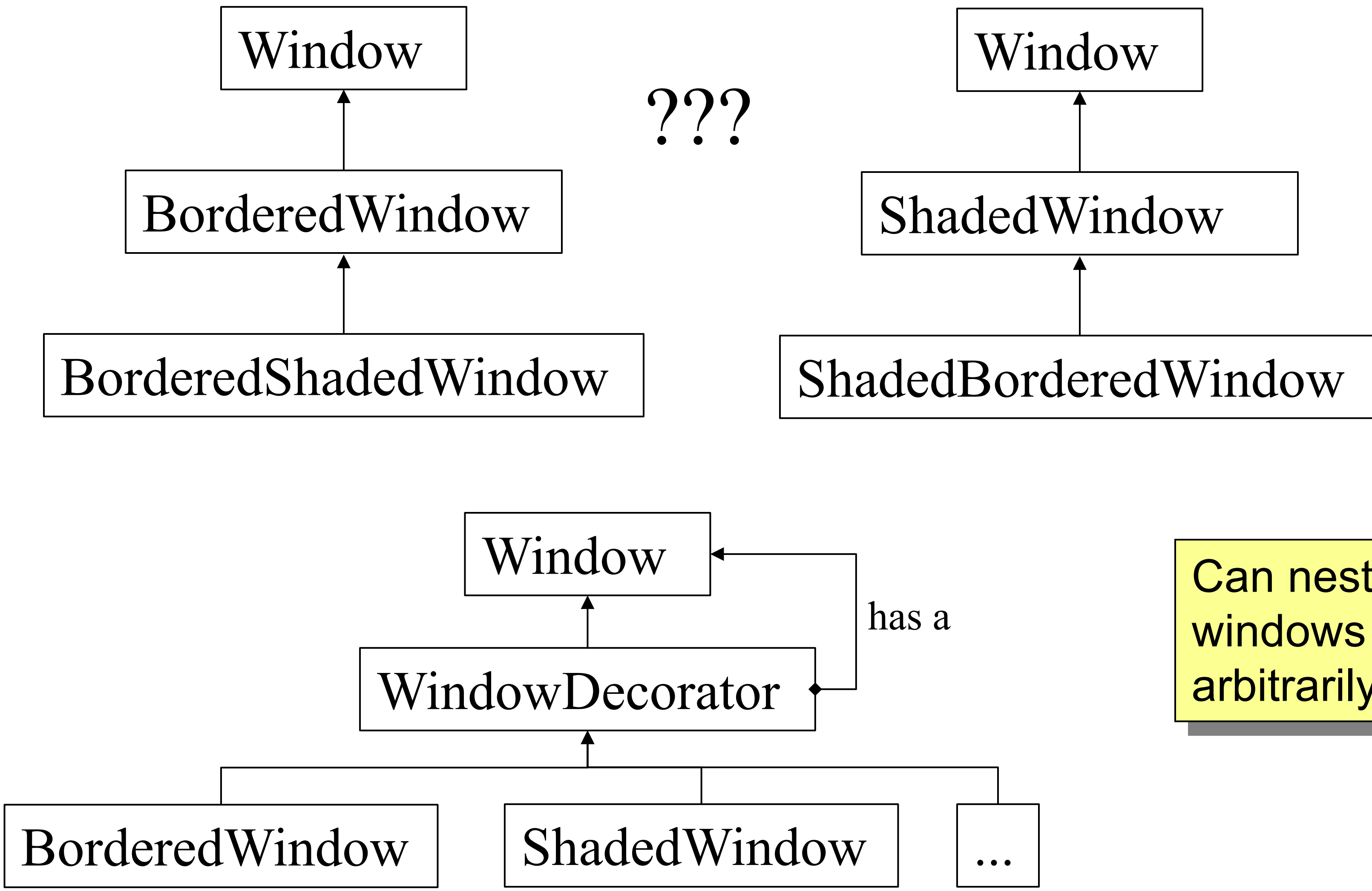
```
class BorderedWindow1 extends WindowImpl {
  void draw(Screen s) {
    super.draw(s);
    bounds().draw(s);
  }
}
```

Delegation permits multiple borders on a window, or a window that is both bordered and shaded

Via delegation:

```
class BorderedWindow2 implements Window {
  Window innerWindow;
  BorderedWindow2(Window innerWindow) {
    this.innerWindow = innerWindow;
  }
  void draw(Screen s) {
    innerWindow.draw(s);
    innerWindow.bounds().draw(s);
  }
}
```

Decorators can solve inheritance issues



Can nest windows arbitrarily!

A decorator can remove functionality

Remove functionality without changing method signatures

- it does change the spec though

Example: **`UnmodifiableList`**

- What does it do about methods like **`add`** and **`put`**?

Problem: **`UnmodifiableList`** is a Java subtype, but not a true subtype, of **`List`**

Decoration via delegation can create a class with no Java subtyping relationship, which is often desirable

Proxy

- Same interface *and* functionality as the wrapped class
  - So, uh, why wrap it?...
  - Note that functionality here means from the client perspective.

- Control access to other objects
  - Communication: manage network details when using a remote object
  - Locking: serialize access by multiple clients
  - Security: permit access only if proper credentials
  - Creation: object might not yet exist (creation is expensive)
    - Hide latency when creating object
    - Avoid work if object is never used

Summary: Wrappers

Wrappers can change the interface or functionality of the encapsulated object.

- The encapsulated object, or delegate, does most of the work

Adapter – make client interface compatible with an implementation

Decorator – add mix-in features

Proxy – add hidden functionality so the client can use the same abstraction for different objects

Pattern	Functionality	Interface
Adapter	same	different
Decorator	different	same
Proxy	same*	same

*from client perspective

Composite

Composite pattern

- Composite permits a client to manipulate either an *atomic* unit or a *hierarchical collection* of units in the same way
  - So no need to “always know” if an object is a collection of smaller objects or not
- Good for dealing with “part-whole” relationships
- **Different from composition!**
  - composition is simply where any object (the wrapper) has another object (the delegate) as an instance field and calls on the delegate to execute certain operations
  - composite pattern involves dealing with object hierarchies as though they were atomic units

- An extended example…

Composite example: Bicycle

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

Composite example: Bicycle

Supports polymorphism for individual parts and hierarchies of parts!

```
abstract class BicycleComponent {
  public int weight();
  public float cost();
}
class Skewer extends BicycleComponent {
  private float price;
  public float cost() { return price; }
  ...
}
class Wheel extends BicycleComponent {
  private float assemblyCost;
  private Skewer skewer;
  private Hub hub;
  ...
  public float cost() {
    return assemblyCost + skewer.cost()
           + hub.cost() + ...;
  }
}
```

Composite example: Libraries

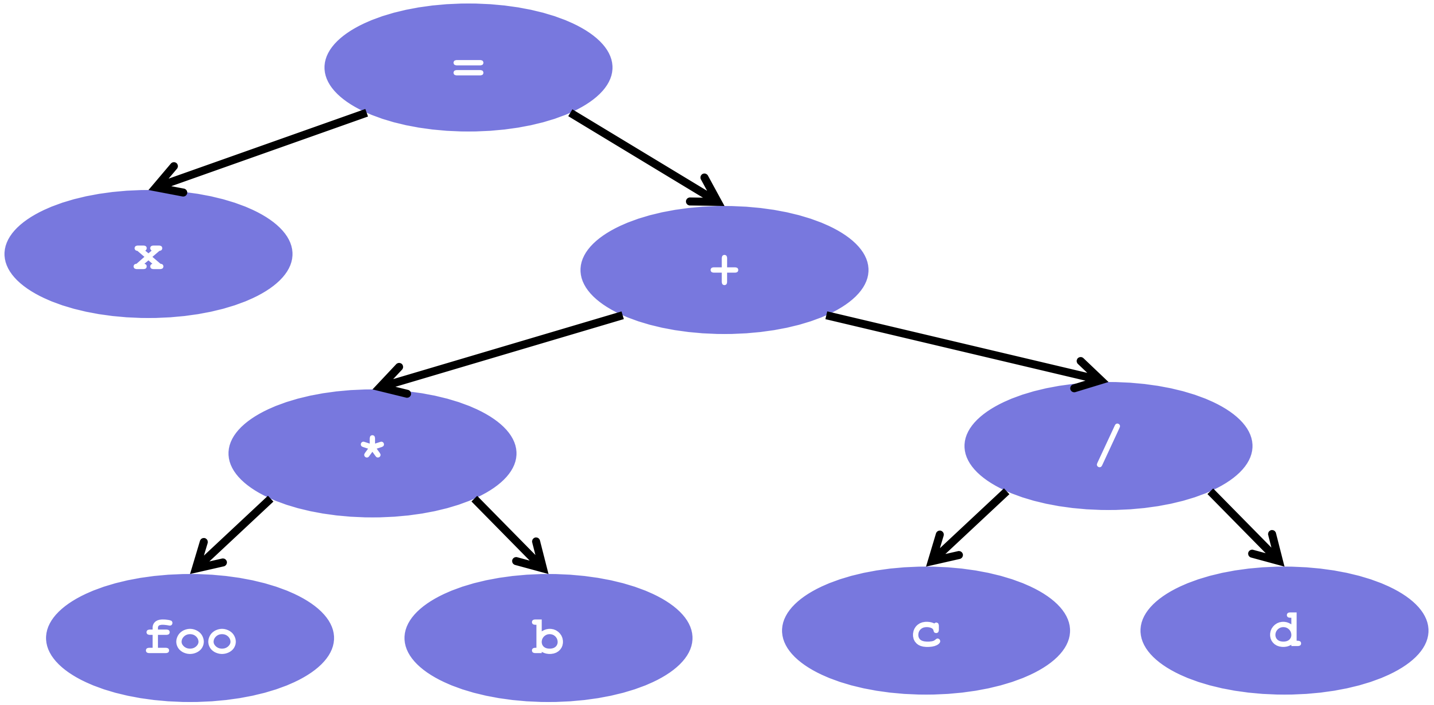
Library
- Section (for a given genre)
  - Shelf
    - Volume
      - Page
        - Column
          - Word
            - Letter

Supports polymorphism for individual parts and hierarchies of parts!

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

Composite example: Abstract Syntax Tree

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

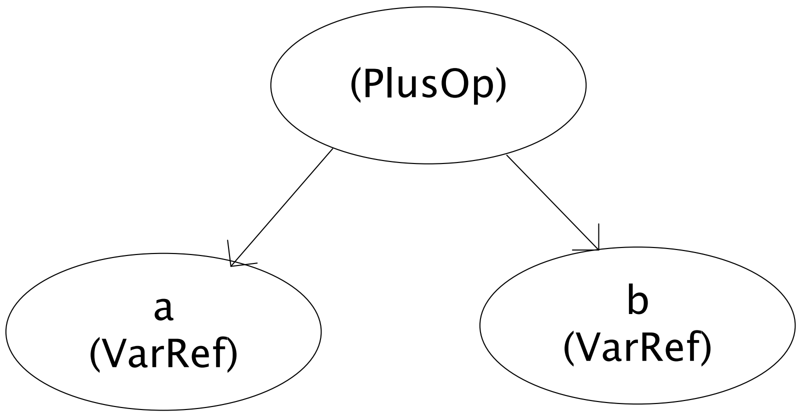


Abstract syntax tree for Java code

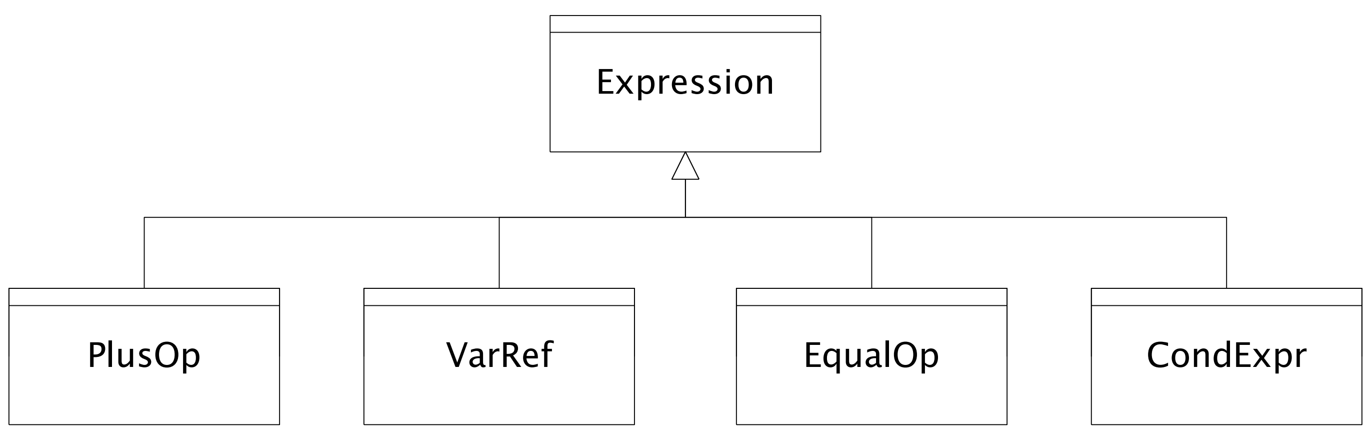
```
class PlusOp extends Expression { // + operation
  Expression leftExp;
  Expression rightExp;
}
class VarRef extends Expression { // variable use
  String varname;
}
class GreaterOp extends Expression {  // test a > b
  Expression leftExp;
  Expression rightExp;
}
class CondOp extends Expression {  // a ? b:c
  Expression testExp;
  Expression thenExp;
  Expression elseExp;
}
```

Object model vs. type hierarchy

- AST for **`a + b`**:



- Class hierarchy for **`Expression`**:



Summary: Composite pattern

- Manipulate either an *atomic* unit or a *hierarchical collection* of units in the same way

- **Different from composition!**
  - composition is simply where any object (the wrapper) has another object (the delegate) as an instance field and calls on the delegate to execute certain operations
  - composite pattern involves dealing with object hierarchies as though they were atomic units

Outline

✓ Introduction to design patterns

✓ Creational patterns (constructing objects)

✓ Structural patterns

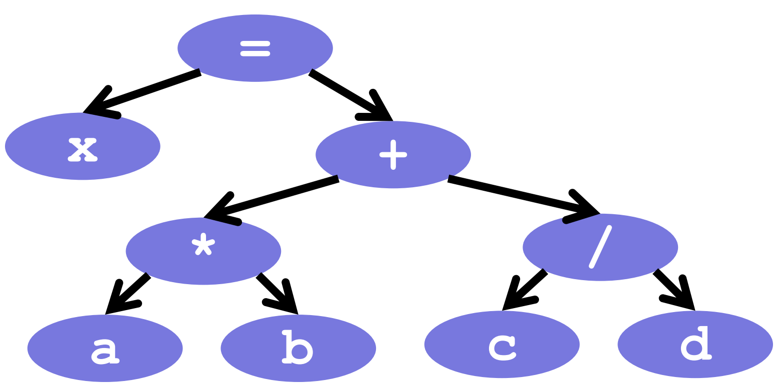
⇒ Behavioral patterns (affecting object semantics)

- Already seen: Observer / Listeners
- Will just do 2-3 related ones

Behavioral Patterns

Traversing composites

- Goal: perform operations on all parts of a composite
- Idea: generalize the notion of an iterator – process the components of a composite in an order appropriate for the application
- Example: arithmetic expressions in Java (abstract syntax trees)
  - How do we traverse/process these expressions?



Operations on abstract syntax trees

Suppose we are writing a compiler!

Need to write code for each entry in this table

Types of Objects

Operations		**CondExpr**	**EqualOp**
	typecheck		
	print		

- Question: Should we group together the code for a particular operation or the code for a particular expression?
  - That is, do we group the code into rows or columns?
- Given an operation and an expression, how do we “find” the proper piece of code?

Interpreter and procedural patterns

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

- Makes it easy to add types of objects, hard to add operations
- An instance of the Composite pattern

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

- Makes it easy to add operations, hard to add types of objects
- The Visitor pattern is a variety of the procedural pattern

(See also many offerings of CSE341 for an extended take on this question

- Statically typed functional languages help with procedural whereas statically typed object-oriented languages help with interpreter)

Interpreter pattern

	Objects	
	CondExpr	EqualOp
typecheck		
print		

Add a method to each class for each supported operation

```
abstract class Expression {
  ...
  Type typecheck();
  String print();
}
class EqualOp extends Expression
  ...
  Type typecheck() { ... }
  String print() { ... }
}
class CondOp extends Expression {
  ...
  Type typecheck() { ... }
  String print() { ... }
}
```

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

Overall type-checker spread across classes

Objects

	CondExpr	EqualOp
typecheck		
print		

Procedural pattern

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

```
class Typecheck {
  Type typeCheckCondExpr(CondExpr e) {
    Type condType = typeCheckExpr(e.condition);
    Type thenType = typeCheckExpr(e.thenExpr);
    Type elseType = typeCheckExpr(e.elseExpr);
    if (condType.equals(BoolType) &&
            thenType.equals(elseType))
      return thenType;
    else
      return ErrorType;
  }
  Type typeCheckEqualOp(EqualOp e) {
    ...
  }
}
```

How to invoke the right method for an expression `e`?

Definition of **`typeCheckExpr`** (using procedural pattern)

```
class Typecheck {
  ...
  Type typeCheckExpr(Expression e) {
    if (e instanceof PlusOp) {
      return typeCheckPlusOp((PlusOp)e);
    } else if (e instanceof VarRef) {
      return typeCheckVarRef((VarRef)e);
    } else if (e instanceof EqualOp) {
      return typeCheckEqualOp((EqualOp)e);
    } else if (e instanceof CondExpr) {
      return typeCheckCondExpr((CondExpr)e);
    } else ...
    ...
  }
}
```

Definition of `typeCheckExpr` (using procedural pattern)

```
class Typecheck {
  ...
  Type typeCheckExpr(Expression e) {
    if (e instanceof PlusOp) {
      return typeCheckPlusOp((PlusOp)e);
    } else if (e instanceof VarRef) {
      return typeCheckVarRef((VarRef)e);
    } else if (e instanceof EqualOp) {
      return typeCheckEqualOp((EqualOp)e);
    } els
      ret
    } els
    ...
  }
}
```

Maintaining this code is tedious and error-prone

- No help from type-checker to get all the cases (unlike in functional languages)

Cascaded if tests are likely to run slowly (in Java)

Need similar code for each operation

Visitor pattern:
A variant of the procedural pattern

- Nodes (objects in the hierarchy) accept visitors for traversal
- 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 extends Visitor {
  void visit(SomeExpression n) {
    perform work on n
  }
}
```

n.accept(v) traverses the structure rooted at **n**, performing **v**'s operation on each element of the structure

Example: accepting visitors

```
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);
  }
}
```

First visit all children

Then pass "self" back to visitor

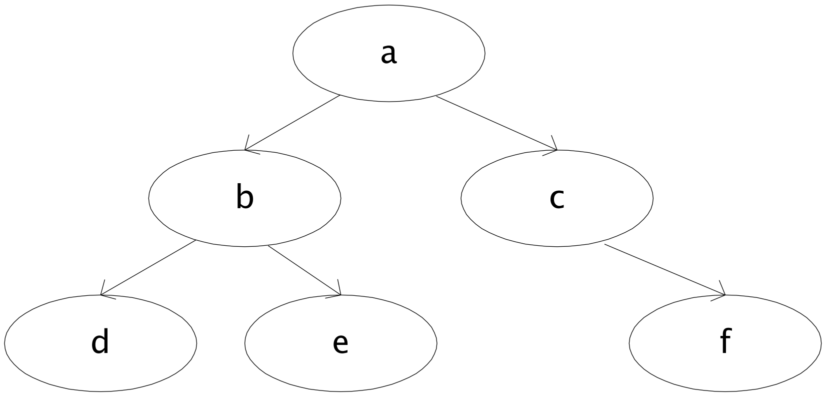
The visitor has a **`visit`** method for each kind of expression, thus picking the right code for this kind of expression

- Overloading makes this look more magical than it is…

Lets clients provide unexpected visitors

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)



Sequence of calls to visit: d, e, b, f, c, a

Example: Implementing visitors

```
class TypeCheckVisitor
   implements Visitor {
  void visit(VarOp e) { … }
  void visit(EqualsOp e) { … }
  void visit(CondOp e) { … }
}

class PrintVisitor implements
   Visitor {
  void visit(VarOp e) { … }
  void visit(EqualsOp e) { … }
  void visit(CondOp e) { … }
}
```

Example: Implementing visitors

```
class TypeCheckVisitor
   implements Visitor {
  void visit(VarOp e) { … }
  void visit(EqualsOp e) { …
  void visit(CondOp e) { … }
}

class PrintVisitor implement
   Visitor {
  void visit(VarOp e) { … }
  void visit(EqualsOp e) { …
  void visit(CondOp e) { … }
}
```

Now each operation has its cases back together

And type-checker should tell us if we fail to implement an abstract method in Visitor

Again: overloading just a nicety

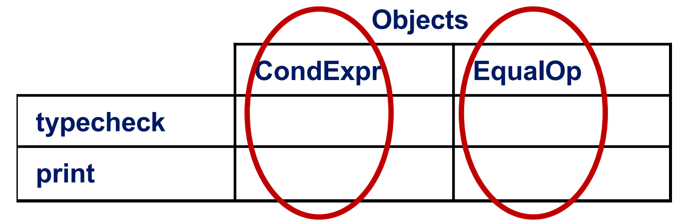
Again: An OOP workaround for procedural pattern

- Because language/type-checker is not instance-of-test friendly

Behavioral patterns: Summary

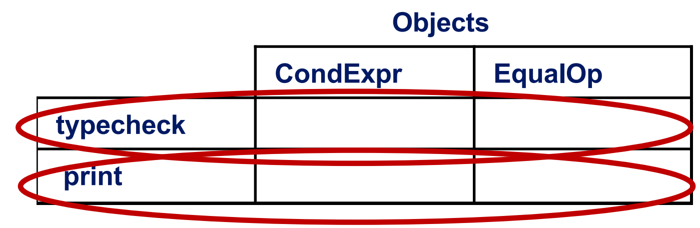
Interpreter Pattern: group operations by object

- Java is well suited for it



Procedural Pattern: group operations by operation

- awkward in Java



Visitor Pattern: Variant of procedural pattern that uses double dispatch to alleviate some of the awkwardness

- code is grouped by operation in visitors
- each class in the hierarchy implements accept method