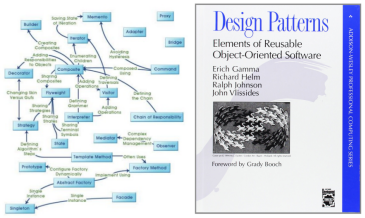
Design Patterns So Far



- Creational patterns: Factories, Prototype, Singleton, Interning
  - Problem: constructors in Java (and other OO languages) are inflexible
    - 1. Can't return a subtype of the type they belong to. "Factory" patterns address the issue: Factory method (e.g. **`createBicycle()`**), Factory class/object, Prototype
    - 2. Always return a fresh new object, can't reuse.
      - "Sharing" patterns address the issue: Singleton, Interning

Design Patterns

- FlyWeight
  - Many objects are similar
  - Objects have many common components
- Wrappers: Adapter, Decorator, Proxy
  - Structural patterns: when we want to change interface or functionality of an existing class, or restrict access to an object
- Composite
  - A structural pattern: expresses whole-part structures, gives uniform interface to client
- Patterns for traversal of composites: Interpreter, Procedural and Visitor

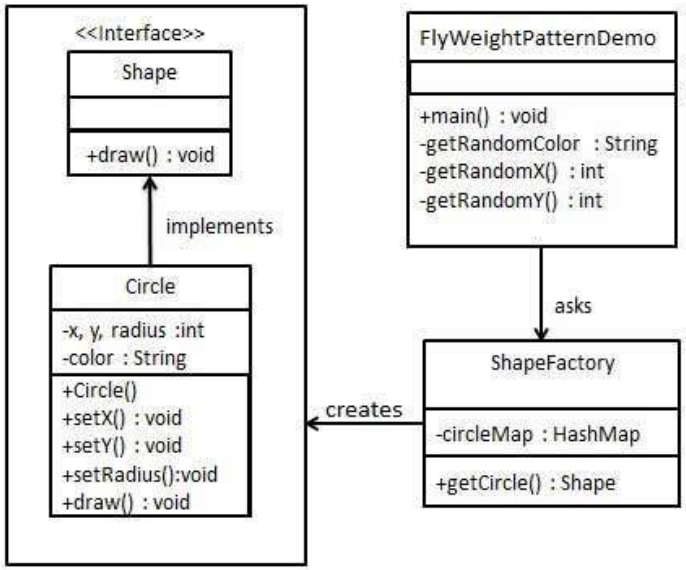
Flyweight Pattern

- Good when many objects are mostly the same
  - Interning works only if objects fall into groups and objects in each group are completely the same and immutable
- If there is an intrinsic state that is the same across all objects
  - Intern it
- If there is an extrinsic state that is different for different objects
  - Extrinsic - not part of the essential nature of someone or something; coming or operating from outside.
  - Represent it explicitly

Flyweight Pattern

- The flyweight pattern is primarily used to reduce the number of objects created
  - decrease memory footprint
  - increase performance
  - decrease object count
  - creates new object when no matching object is found.
  - Flyweight pattern often involves a Factory or composition

Flyweight Example



We will demonstrate this pattern by drawing 20 circles at different locations but we will create only 5 objects. Only 5 colors are available, so the color property is used to check already existing *Circle* objects.

Similar to memoization.

https://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm

```
public interface Shape {
  void draw();
}

public class Circle implements Shape {
  private String color; // intrinsic
  private int x;
  private int y;
  private int radius;

  public Circle(String color){
    this.color = color;
  }

  @Override
  public void draw() {
    System.out.println("Circle: Draw() [Color : " + color +
        ", x : " + x + ", y :" + y + ", radius :" + radius);
  }
  …
}
```

```
public class ShapeFactory {
  private static final HashMap<String, Shape> circleMap =
                new HashMap<String, Shape>();

  public static Shape getCircle(String color) {
    Circle circle = (Circle)circleMap.get(color);

    if(circle == null) {
      circle = new Circle(color);
      circleMap.put(color, circle);
      System.out.println("Creating circle of color : " + color);
    }
    return circle;
  }
}
```

```
public class FlyweightPatternDemo {
  private static final String colors[] = { "Red", "Green", "Blue", "White", "Black" };
  public static void main(String[] args) {

    for(int i=0; i < 20; ++i) {
      Circle circle = (Circle)ShapeFactory.getCircle(getRandomColor());
      circle.setX(getRandomX());
      circle.setY(getRandomY());
      circle.setRadius(100);
      circle.draw();
    }
  }
  private static String getRandomColor() {
    return colors[(int)(Math.random()*colors.length)];
  }
  private static int getRandomX() {
    return (int)(Math.random()*100 );
  }
  private static int getRandomY() {
    return (int)(Math.random()*100);
  }
}
```

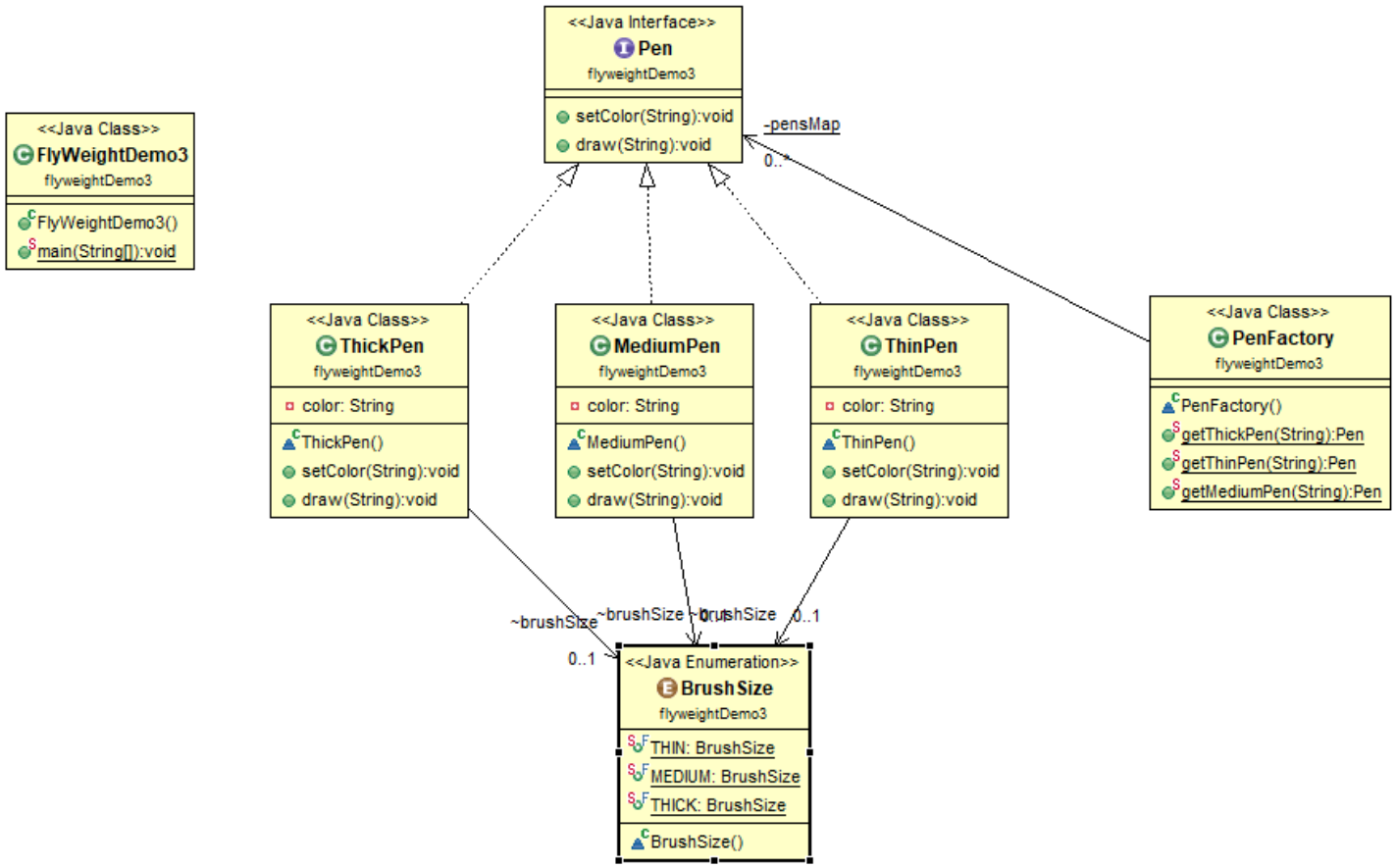
```
Creating circle of color : Black
Circle: Draw() [Color : Black, x : 36, y :71, radius :100
Creating circle of color : Green
Circle: Draw() [Color : Green, x : 27, y :27, radius :100
Creating circle of color : White
Circle: Draw() [Color : White, x : 64, y :10, radius :100
Creating circle of color : Red
Circle: Draw() [Color : Red, x : 15, y :44, radius :100
Circle: Draw() [Color : Green, x : 19, y :10, radius :100
Circle: Draw() [Color : Green, x : 94, y :32, radius :100
Circle: Draw() [Color : White, x : 69, y :98, radius :100
Creating circle of color : Blue
Circle: Draw() [Color : Blue, x : 13, y :4, radius :100
Circle: Draw() [Color : Green, x : 21, y :21, radius :100
Circle: Draw() [Color : Blue, x : 55, y :86, radius :100
Circle: Draw() [Color : White, x : 90, y :70, radius :100
Circle: Draw() [Color : Green, x : 78, y :3, radius :100
Circle: Draw() [Color : Green, x : 64, y :89, radius :100
Circle: Draw() [Color : Blue, x : 3, y :91, radius :100
Circle: Draw() [Color : Blue, x : 62, y :82, radius :100
Circle: Draw() [Color : Green, x : 97, y :61, radius :100
Circle: Draw() [Color : Green, x : 86, y :12, radius :100
Circle: Draw() [Color : Green, x : 38, y :93, radius :100
Circle: Draw() [Color : Red, x : 76, y :82, radius :100
Circle: Draw() [Color : Blue, x : 95, y :82, radius :100
```

Flyweight and Interning

- The previous example looks a lot like interning
- It is.
  - Only a single instance of a circle with a color is created
- Flyweight often uses interning.
  - To store the common parts of object
  - In example, color is common part
  - Other parts are set by getCircle()

Another Flyweight Example

- Paint Brush application where client can use brushes on three types – THICK, THIN and MEDIUM.
- All the thick (thin or medium) brushes will draw the content in a similar fashion
  - only the content color will be different.
- Brush color is the extrinsic attribute which will be supplied by client
  - Everything else will remain the same for the Pen.
- Client creates three THIN pens
  - 2 YELLOW, 1 BLUE
- Runtime there is only one pen object of each color type
  - Looks a lot like interning
- See FlyWeightDemo3.java


<<Java Interface>>
Pen
flyweightDemo3
setColor(String):void
draw(String):void
-pensMap
0..*
<<Java Class>>
FlyWeightDemo3
flyweightDemo3
FlyWeightDemo3()
main(String[]):void
<<Java Class>>
ThickPen
flyweightDemo3
color: String
ThickPen()
setColor(String):void
draw(String):void
<<Java Class>>
MediumPen
flyweightDemo3
color: String
MediumPen()
setColor(String):void
draw(String):void
<<Java Class>>
ThinPen
flyweightDemo3
color: String
ThinPen()
setColor(String):void
draw(String):void
<<Java Class>>
PenFactory
flyweightDemo3
PenFactory()
getThickPen(String):Pen
getThinPen(String):Pen
getMediumPen(String):Pen
~brushSize
0..1
~brushSize
~brushSize
0..1
<<Java Enumeration>>
BrushSize
flyweightDemo3
THIN: BrushSize
MEDIUM: BrushSize
THICK: BrushSize
BrushSize()

Example: bicycle spokes

- 32 to 36 spokes per wheel
  - Only 3 varieties per bike model
- In a bike race, hundreds of spoke varieties
  - Thousands of instances

```
class Wheel {
        FullSpoke[] spokes;
        ...
}
class FullSpoke {
        int length;
        int diameter;
        bool tapered;
        Metal material;
        float weight;
        float threading;
        bool crimped;
        int location; // rim and hub holes this is installed in
```

Alternatives to FullSpoke

Intern IntrinsicSpoke

```
class IntrinsicSpoke {
        int length;
        int diameter;
        boolean tapered;
        Metal material;
        float weight;
        float threading;
        boolean crimped;
    }
```

Doesn't save space. Same as FullSpoke

```
class InstalledSpokeFull extends IntrinsicSpoke {
    int location;
}
```

Composition - Saves space because of interning

```
class InstalledSpokeWrapper {
        IntrinsicSpoke s; // refer to interned object
        int location;
}
```

Align (true) a Wheel

```
class FullSpoke {
        // Tension the spoke by turning the nipple the
        // specified number of turns.
        void tighten(int turns) {
          ... location ... // location is a field
        }
}
class Wheel {
        FullSpoke[] spokes;
        void align() {
                while ( wheel is misaligned) {
                        // tension the ith spoke
                        ... spokes[i].tighten(numturns) ...
                }
        }
}
```

What is value of location in spokes[i]

Flyweight code to true (align) a wheel

```
class IntrinsicSpoke {
        void tighten(int turns, int location) {
                ... location ... // location is a parameter
        }
}
class Wheel {
        IntrinsicSpoke[] spokes;
        void align() {
                while (wheel is misaligned) {
                // tension the ith spoke
                        ... spokes[i].tighten(numturns, i) ... // pass the location to tighten()
                }
        }
}
```

Flyweight Pattern

- What if FullSpoke contains a wheel field pointing at the Wheel containing it?
  - Wheel methods pass *this* to the methods that use the wheel field.
- What if Fullspoke contains a boolean indication of a broken spoke
  - Add an array of booleans parallel to spokes
- Flyweight used when there are very few mutable (extrinsic) fields
  - Complicates code
  - Use when profiling has determined that space is a serious problem

When to use the Flyweight Pattern

Thousands of new instantiation occurrences of a memory intensive heavy object

The objects created via the heavy object have very similar features
in every instantiation occurrence

The object pool contains many similar objects and two objects from
the pool don’t differ significantly

The object creation consumes high memory usage

Objects have the shareable ability

Wrappers

- A wrapper uses composition/delegation
- A wrapper is a thin layer over an encapsulated class
  - Modify the interface
    - GraphWrapper
  - Extend behavior of encapsulated class
  - Restrict access to encapsulated object
- The encapsulated object (delegate) does most work
- Wrapper is not a GoF pattern
  - Similar to Adapter and Façade GoF patterns

Structural Patterns

Pattern	Function	Interface
Adapter	Same	Different
Decorator	Different	Same/Different
Proxy	Same	Same

Adapter Pattern

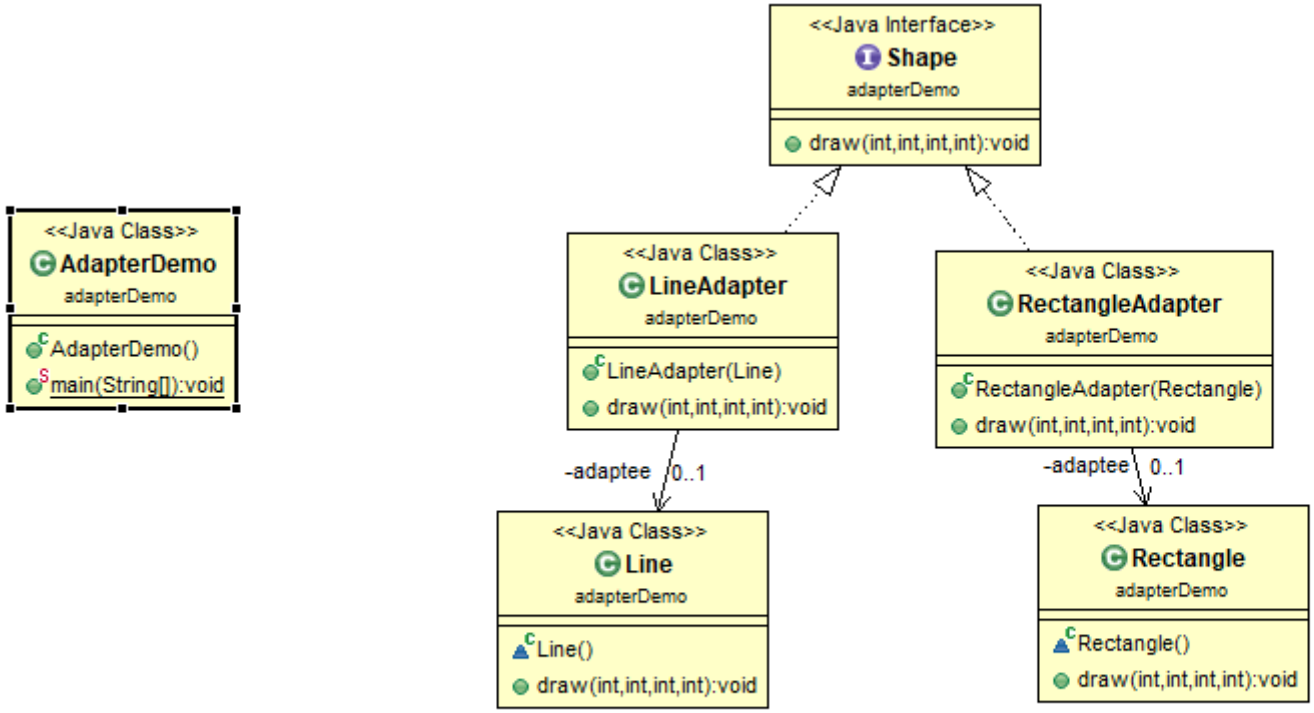
- The purpose of the Adapter is:
  - change an interface, without changing the functionality of the encapsulated class
  - Allows reuse of functionality
  - Protects client from modification of encapsulated class
- Reasons
  - Rename methods
  - Convert units
  - Reformat output
- Example
  - Angles passed in radians instead of degrees

Adapter Pattern

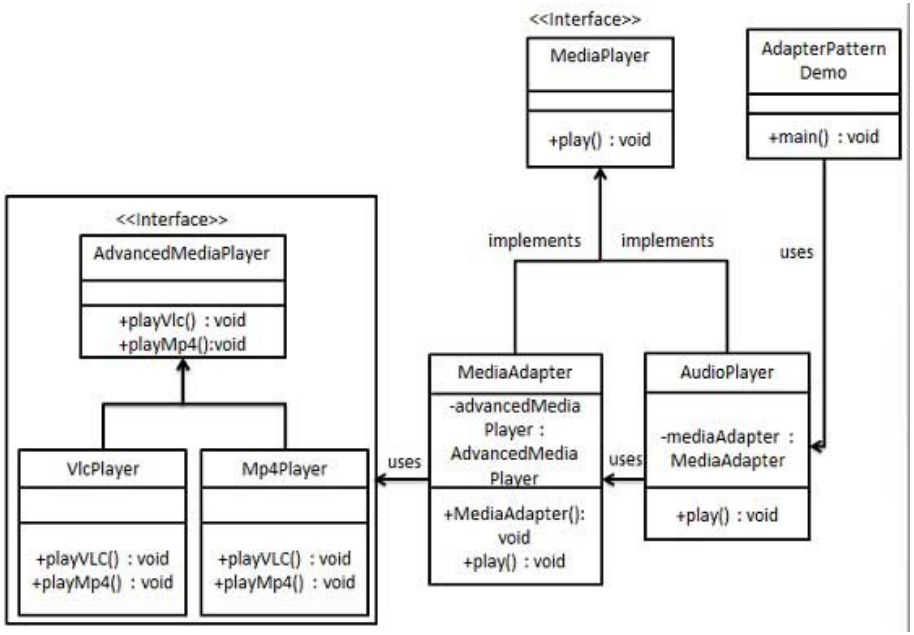
- Bridge between two incompatible interfaces
- Single class which is responsible to join functionalities of independent or incompatible interfaces.
  - Example: SD card reader which acts as an adapter between memory card and a laptop.

Adapter Example

- Mapping a user-friendly common interface to legacy-specific peculiar interfaces.
  - Line and Rectangle both have draw methods
    - Different classes, not subtypes of a common supertype
  - Adapter methods are subtypes of Shape
    - Provide a draw methods
    - Client codes against Shape
    - Client can use common methods
  - See AdapterDemo.java


<<Java Interface>>
Shape
adapterDemo
draw(int,int,int,int):void
<<Java Class>>
AdapterDemo
adapterDemo
AdapterDemo()
main(String[]):void
<<Java Class>>
LineAdapter
adapterDemo
LineAdapter(Line)
draw(int,int,int,int):void
<<Java Class>>
RectangleAdapter
adapterDemo
RectangleAdapter(Rectangle)
draw(int,int,int,int):void
-adaptee
0..1
-adaptee
0..1
<<Java Class>>
Line
adapterDemo
Line()
draw(int,int,int,int):void
<<Java Class>>
Rectangle
adapterDemo
Rectangle()
draw(int,int,int,int):void

Adapter Example



https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm

```
public interface MediaPlayer {
  public void play(String audioType, String fileName);
}

public interface AdvancedMediaPlayer {
  public void playVlc(String fileName);
  public void playMp4(String fileName);
}
```

```
public class VlcPlayer implements AdvancedMediaPlayer{
  @Override
  public void playVlc(String fileName) {
    System.out.println("Playing aac file. Name: "+ fileName);

  }
  …
  @Override
  public void playMp4(String fileName) {
    //do nothing
  }
}
public class Mp4Player implements AdvancedMediaPlayer{

  @Override
  public void playVlc(String fileName) {
    //do nothing
  }

  @Override
  public void playMp4(String fileName) {
    System.out.println("Playing mp4 file. Name: "+ fileName);
  }
}
```

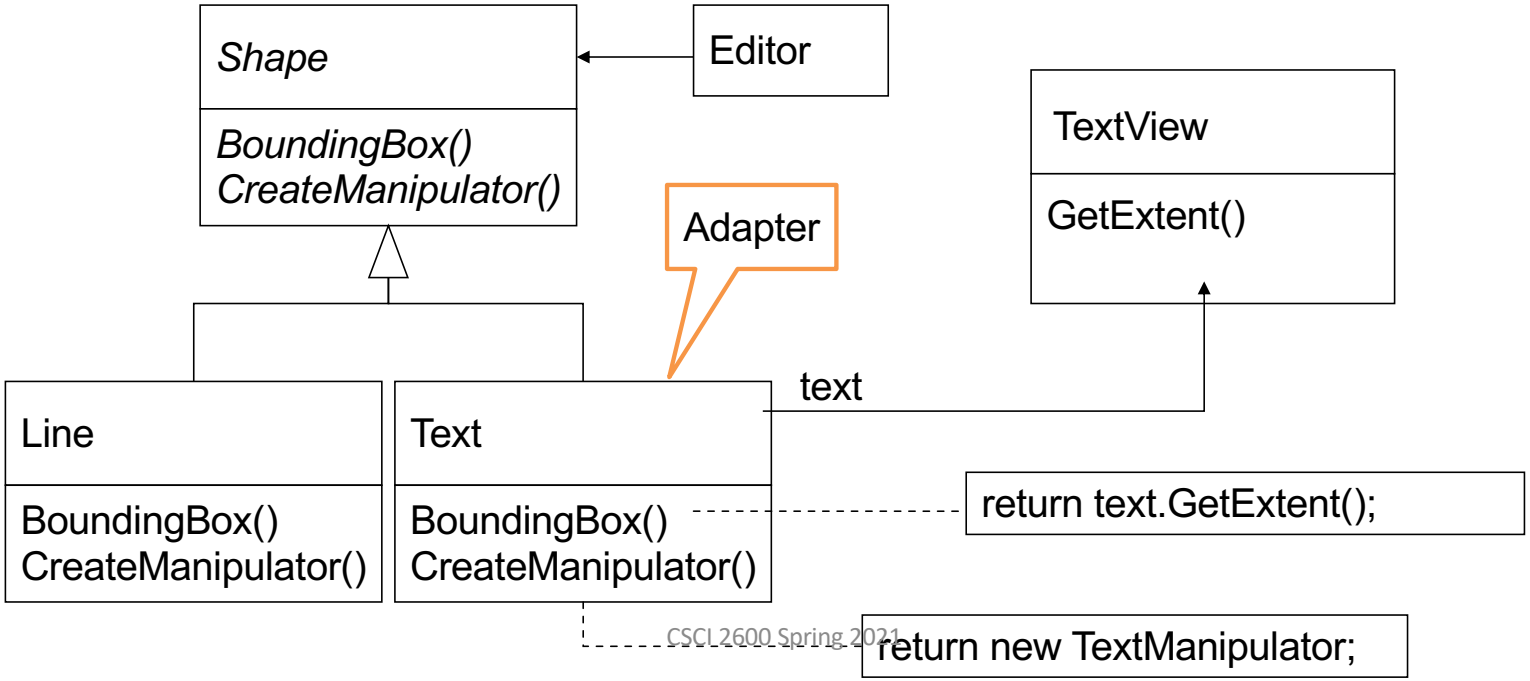
```
public class MediaAdapter implements MediaPlayer {

  AdvancedMediaPlayer advancedMusicPlayer;

  public MediaAdapter(String audioType){
    if(audioType.equalsIgnoreCase("vlc") ){
      advancedMusicPlayer = new VlcPlayer();

    }else if (audioType.equalsIgnoreCase("mp4")){
      advancedMusicPlayer = new Mp4Player();
    }
  }

  @Override
  public void play(String audioType, String fileName) {
    if(audioType.equalsIgnoreCase("vlc")){
      advancedMusicPlayer.playVlc(fileName);
    }
    else if(audioType.equalsIgnoreCase("mp4")){
      advancedMusicPlayer.playMp4(fileName);
    }
  }
}
```

```java
public class AudioPlayer implements MediaPlayer {
  MediaAdapter mediaAdapter;

  @Override
  public void play(String audioType, String fileName) {

    //inbuilt support to play mp3 music files
    if(audioType.equalsIgnoreCase("mp3")){
      System.out.println("Playing mp3 file. Name: " + fileName);
    }

    //mediaAdapter is providing support to play other file formats
    else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
      mediaAdapter = new MediaAdapter(audioType);
      mediaAdapter.play(audioType, fileName);
    }

    else{
      System.out.println("Invalid media. " + audioType + " format not supported");
    }
  }
}
```

```java
public class AdapterPatternDemo {
  public static void main(String[] args) {
    AudioPlayer audioPlayer = new AudioPlayer();

    audioPlayer.play("mp3", "beyond the horizon.mp3");
    audioPlayer.play("mp4", "alone.mp4");
    audioPlayer.play("vlc", "far far away.vlc");
    audioPlayer.play("avi", "mind me.avi");
  }
}
```

Adapter Pattern

- Motivation: reuse a class with an interface different than the class' interface



Adapter Pattern

- We would like to use TextView, which contains a completed implementation.
- However, the client Editor expects the Shape interface.
- If we changed Editor to work with TextView, then we'd have to modify hundreds of places in Editor where Editor uses the Shape interface:
  - e.g., we have to change all calls to BoundedBox() to calls to GetExtent().
- The Adapter pattern allows us to change Editor minimally and still use the implementation of TextView.
- We'll create an Adapter class Text, which encloses TextView and redirects the calls to BoundingBox() to text.GetExtent();
- Editor refers to a Text object, so it keeps calling BoundingBox(), which uses TextView

GoF Adapters

- GoF (Gang of Four) describes two major kinds of adapters:
  - Class adapters
    - Use multiple inheritance to adapt one interface to another.
    - Java doesn't allow multiple inheritance.
    - We need to use interfaces.
  - Object adapters
    - Depend on the object compositions.

Adapter Example: Scaling Rectangles

```
interface Rectangle {
  void scale(int factor); //grow or shrink by factor
  void setWidth();
  float getWidth();
  float area(); …
}
class Client {
  void clientMethod(Rectangle r) {
    … r.scale(2);
  }
}
class NonScalableRectangle {
  void setWidth(); …
  // no scale method!
}
```

Can we use NonScalableRectangle in Client instead?

Class Adapter

- Class adapter adapts via subclassing

```
class ScalableRectangle1
            extends NonScalableRectangle
            implements Rectangle {
   void scale(int factor) {
     setWidth(factor*getWidth());
     setHeight(factor*getHeight());
   }
}
```

Object Adapter

- Object adapter adapts via delegation: it forwards work to delegate

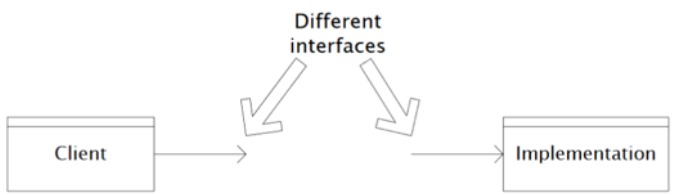
```
class ScalableRectangle2 implements Rectangle {
   NonScalableRectangle r; // delegate
   ScalableRectangle2(NonScalableRectangle r) {
     this.r = r;
   }
   void scale(int factor) {
     r.setWidth(factor * r.getWidth());
     r.setHeight(factor * r.getHeight());
   }
   float getWidth() { return r.getWidth(); }
   …
}
```

Subclassing Versus Delegation

- Subclassing
  - Automatically gives access to all methods in the superclass
  - More efficient
- Delegation
  - Permits removal of methods
  - Multiple objects can be composed
  - More flexible
- Some wrappers have qualities of adapter, decorator, and proxy
  - Differences are subtle

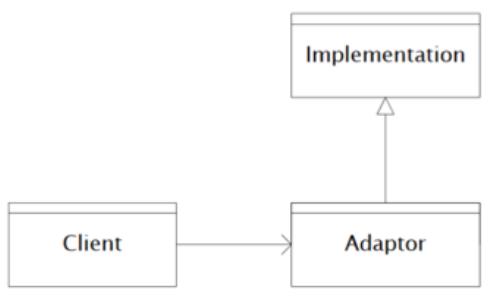
Types of Adapters

Goal of adapter:
connect incompatible interfaces



Adapter with delegation

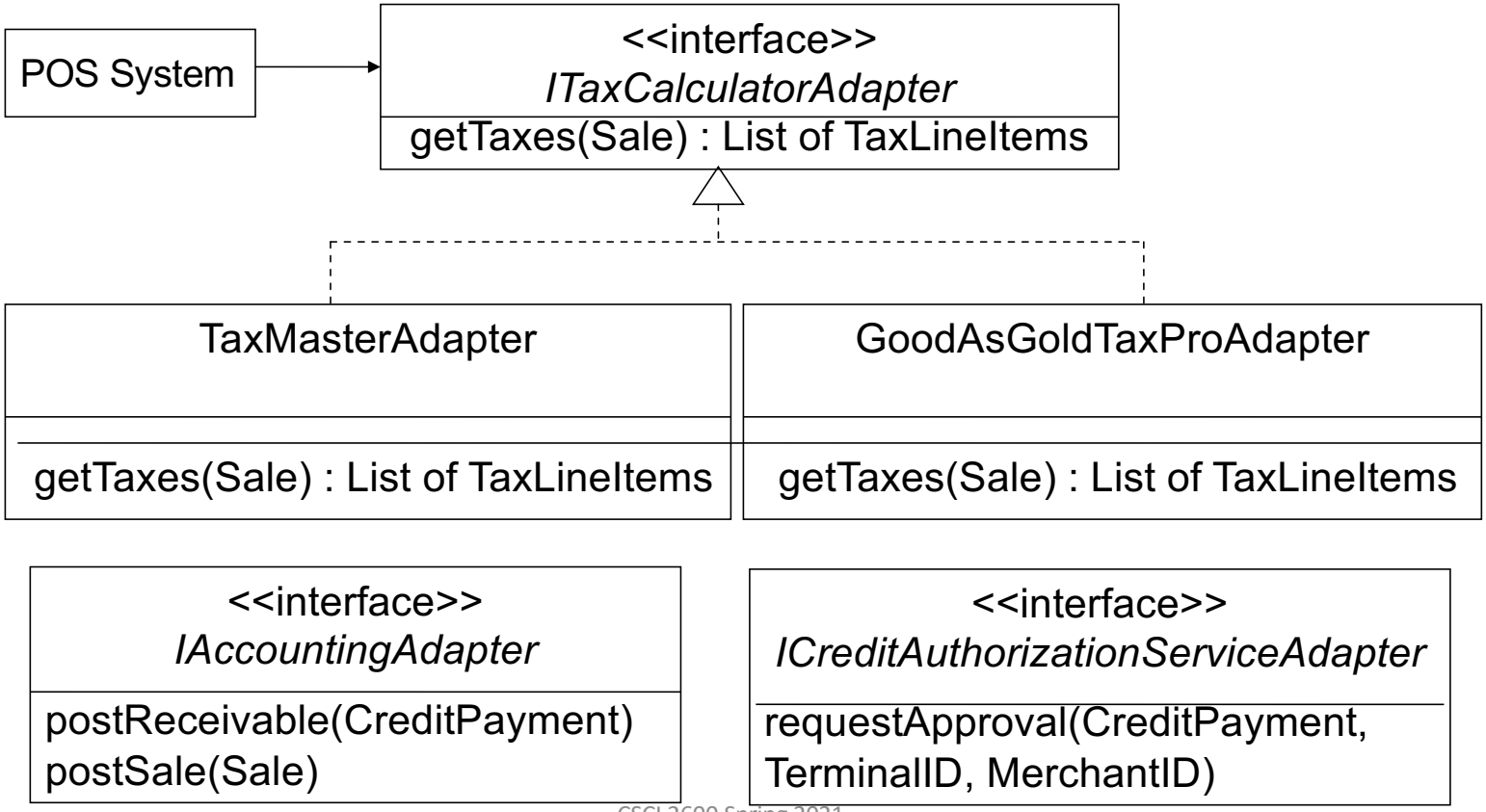
Adapter with subclassing



Another Example

- A Point-of-Sale system needs to support services from different third-party vendors:
  - Tax calculator service from different vendors
  - Credit authorization service from different vendors
  - Inventory systems from different vendors
  - Accounting systems from different vendors
- Each vendor service has its own API, which can’t be changed
- What design pattern helps solve this problem?

The Solution: Object Adapter



Object Adapter

- TaxMasterAdapter adapts the TaxMaster interface to the expected interface.
  - Same for the GoodAsGoldTaxProAdapter.
- The client, POS System uses the interface defined in ITaxCalculatorAdapter.
- If the POS System needs to change from one vendor's system to another, the changes to the POS System client will be minimal to none!

Exercise

- Who creates the appropriate adapter object?
  - Is it a good idea to let some domain object from the Point-of-Sale system (e.g., Register, Sale) create the adapters?
    - That would assign responsibility beyond domain object's logic. We would like to keep domain classes focused, so, this is not a good idea
    - Violates principle of single responsibility for a class
- How to determine what type of adapter object to create? We expect adapters to change.
- What design patterns solve this problem?

The Solution: Factory

ServiceFactory
accountingAdapter : IAccountingAdapter inventoryAdapter : IInventoryAdapter taxCalculatorAdapter : ITaxCalculatorAdapter
getAccountingAdapter() : IAccountingAdapter getInventoryAdapter () : IInventoryAdapter getTaxCalculatorAdapter () : ITaxCalculatorAdapter

Using the Factory

```
public ITaxCalculatorAdapter
                    getTaxCalculatorAdapter() {
  if (taxCalculatorAdapter == null) {
   String className =
     System.getProperty("taxcalculator.classname");
     taxCalculatorAdapter =
      (ITaxCalculatorAdapter)
          Class.forName(className).newInstance();
  }
  return taxCalculatorAdapter;
}
```

- What design pattern(s) do you see here?

Java reflection: creates a brand new object from String className!

Exercise

- Who creates the **`ServiceFactory`**?
- How is it accessed?
- We need a single instance of the **`ServiceFactory`** class
- What pattern solves these problems?

The Solution: Singleton

Special UML notation.

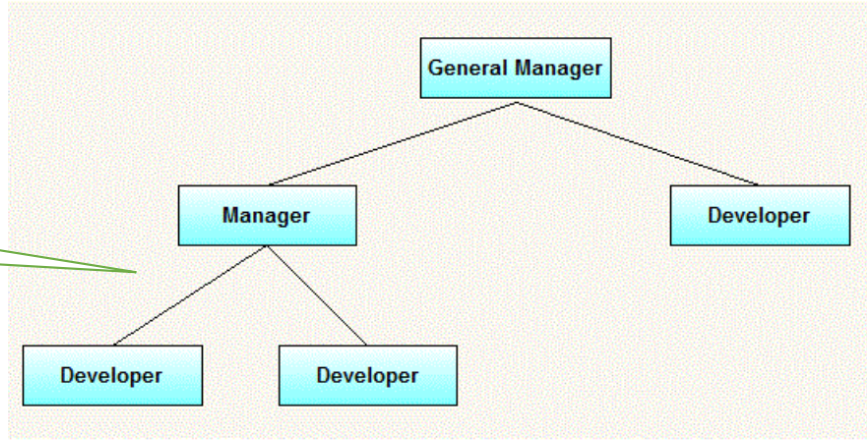
ServiceFactory 1
<u>- instance: ServiceFactory</u> - accountingAdapter : IAccountingAdapter - inventoryAdapter : IInventoryAdapter - taxCalculatorAdapter : ITaxCalculatorAdapter
<u>+ getInstance() : ServiceFactory</u> + getAccountingAdapter() : IAccountingAdapter + getInventoryAdapter() : IInventoryAdapter + getTaxCalculatorAdapter() : ITaxCalculatorAdapter

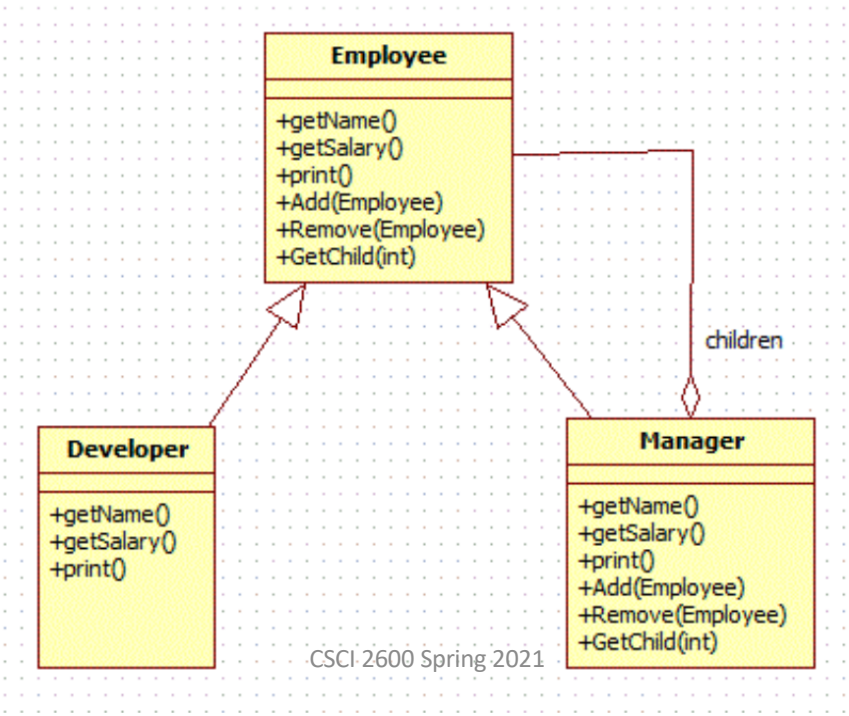
In UML, - means private, + means public. All (shown) fields in **`ServiceFactory`** are private and all methods are public. <u>underline</u> means static. **`instance`** and **`getInstance`** are static. Single instance of **`ServiceFactory`** ensures single instance of adapter objects.

Composite Pattern

- Client treats a composite object (a collection of objects) the same as a simple object (an atomic unit)
- Good for part-whole relationships
  - Can represent arbitrarily complex objects
- Composite pattern composes objects in terms of a tree structure to represent the part as well as the whole hierarchy.
- This pattern creates a class that contains a group of objects.
- The class provides ways to modify its group of objects.

Organization Chart





```
public interface Employee {

  public void add(Employee employee);
  public void remove(Employee employee);
  public Employee getChild(int i);
  public String getName();
  public double getSalary();
  public void print();
}
```

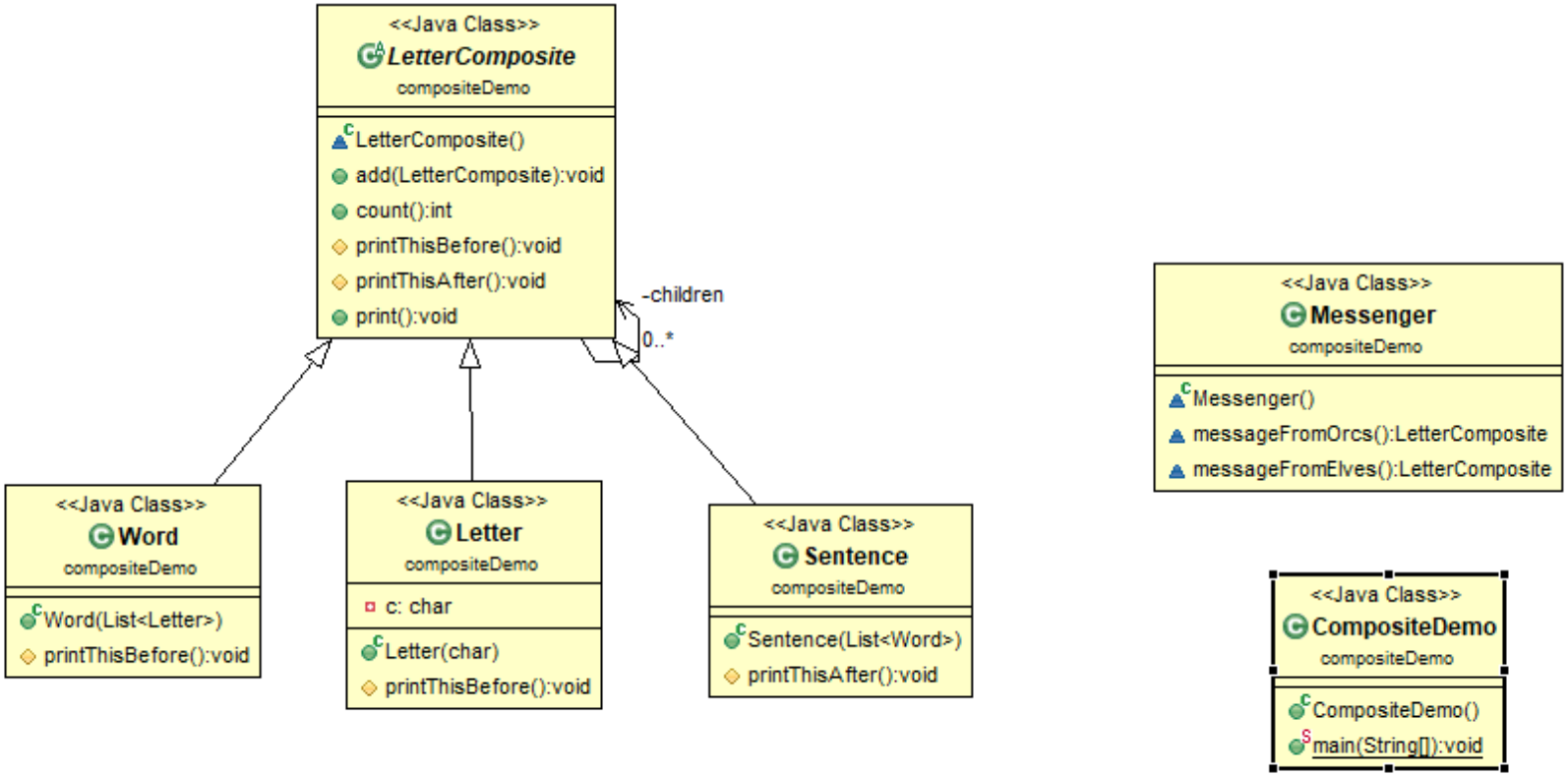
```
public class Manager implements Employee{

   private String name;
   private double salary;

   public Manager(String name, double salary){
      this.name = name;
      this.salary = salary;
   }

 List<Employee> employees = new ArrayList<Employee>();

  public void add(Employee employee) {
     employees.add(employee);
   }

   public Employee getChild(int i) {
     return employees.get(i);
    }
 …// implements print – traverses employees
}
```

```
public class Developer implements Employee{

  private String name;
  private double salary;

  public Developer(String name, double salary){
    this.name = name;
    this.salary = salary;
  }
  public void add(Employee employee) {
    //this is leaf node so this method is not applicable to this class.
  }

  public Employee getChild(int i) {
    //this is leaf node so this method is not applicable to this class.
    return null;
  }
  …
}
```

```
public static void main(String[] args) {
  Employee emp1=new Developer("John", 10000);
  Employee emp2=new Developer("David", 15000);
  Employee manager1=new Manager("Daniel",25000);
  manager1.add(emp1);
  manager1.add(emp2);
  Employee emp3=new Developer("Michael", 20000);
  Manager generalManager=new Manager("Mark", 50000);
  generalManager.add(emp3);
  generalManager.add(manager1);
  generalManager.print();
 }
}
```

Composite Example

- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.
- Example
  - Every sentence is composed of words which are in turn composed of characters.
  - Each of these objects is printable and they can have something printed before or after them
  - Sentence always ends with full stop
  - Word always has space before it
- See CompositeDemo.java


<<Java Class>>
LetterComposite
compositeDemo
LetterComposite()
add(LetterComposite):void
count():int
printThisBefore():void
printThisAfter():void
print():void
-children
0..*
<<Java Class>>
Word
compositeDemo
Word(List<Letter>)
printThisBefore():void
<<Java Class>>
Letter
compositeDemo
c: char
Letter(char)
printThisBefore():void
<<Java Class>>
Sentence
compositeDemo
Sentence(List<Word>)
printThisAfter():void
<<Java Class>>
Messenger
compositeDemo
Messenger()
messageFromOrcs():LetterComposite
messageFromElves():LetterComposite
<<Java Class>>
CompositeDemo
compositeDemo
CompositeDemo()
main(String[]):void

Example: Bicycle

- Bicycle
  - Wheel
    - Skewer
      - Lever
      - Body
      - Cam
      - Rod
      - Acorn nut
    - Hub
    - Spokes
    - …
  - Frame
  - …

Example: Methods on Components

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

Skewer is an atomic unit

Wheel is a collection of objects

Even Better

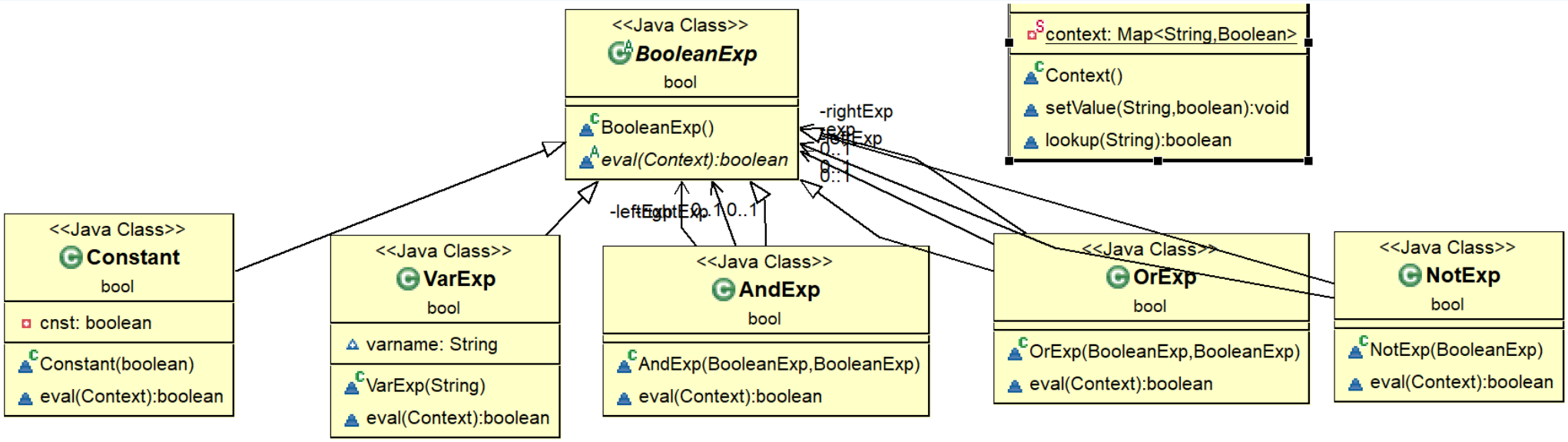
```
abstract class BicycleComponent {
  …
  float cost();
}
class Skewer extends BicycleComponent {
  float price;
  float cost() { return price; }
}
class Wheel extends BicycleComponent {
  float assemblyCost;
  BicycleComponent skewer;
  BicycleComponent hub;
  …
  float cost() { return assemblyCost+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

The skewer and hub are BicycleCompoenents, so we can use BicycleComponent!

Another Example: Boolean Expressions

- A Boolean expression can be
  - Variable (e.g., **`x`**)
  - Boolean constant: **`true, false`**
  - Or expression (e.g., **`x or true`**)
  - And expression (e.g., **`(x or true) and y`**)
  - Not expression (e.g., **`not x, not (x or y)`**)
- And, Or, Not: collections of expressions
- Variable and Constant: atomic units
  - Maybe we should intern the Boolean constants?
- See BooleanDemo.java

Interpreter Pattern



See BooleanDemo.java

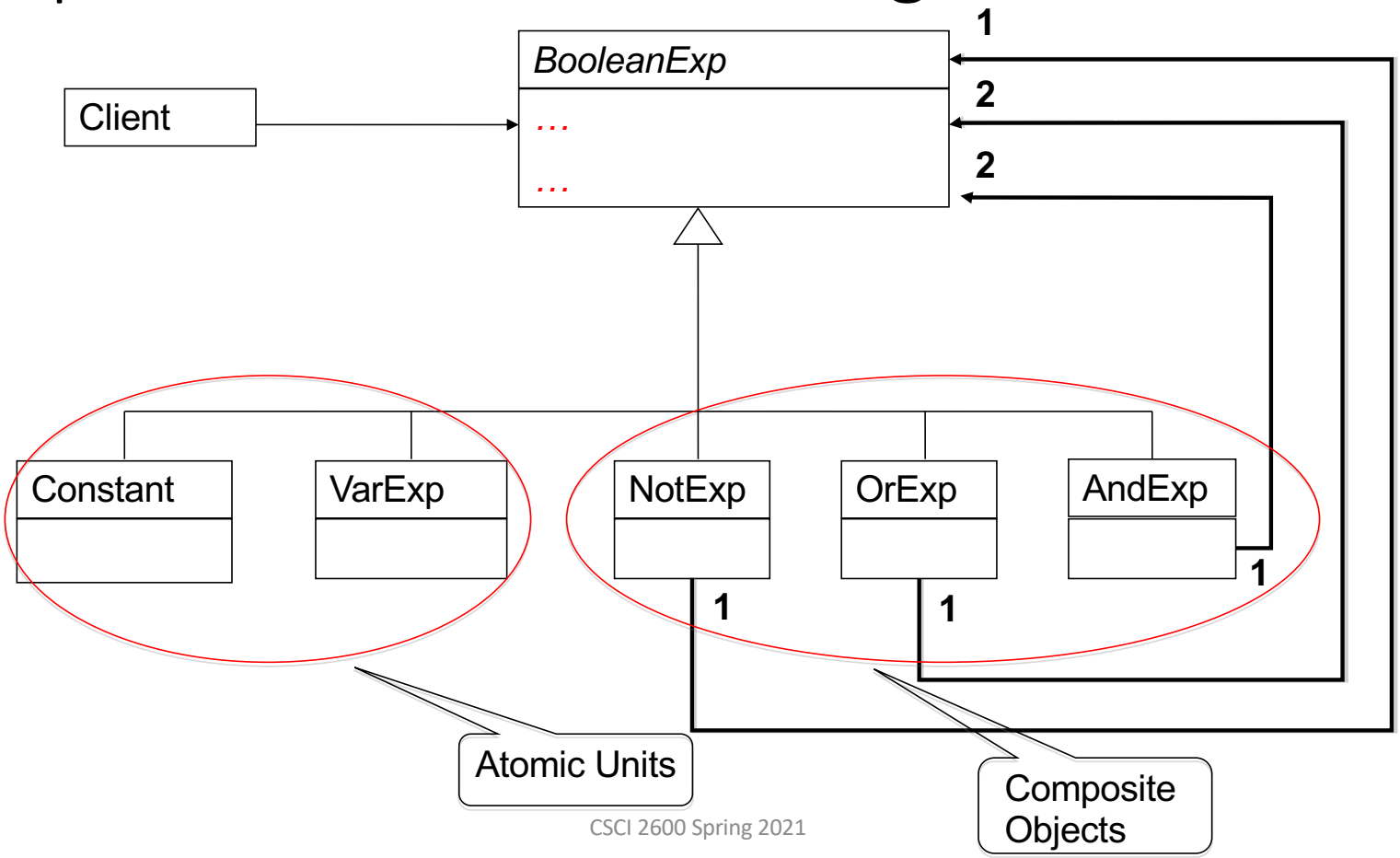
Using Composite to Represent Boolean Expressions

```
abstract class BooleanExp {
  boolean eval(Context c);
}
class Constant extends BooleanExp {
  private boolean const;
  Constant(boolean const) { this.const=const; }
  boolean eval(Context c) { return const; }
}
class VarExp extends BooleanExp {
  String varname;
  VarExp(String var) { varname = var; }
  boolean eval(Context c) {
    return c.lookup(varname);
  }
}
```

Using Composite to Represent Boolean Expressions

```
class AndExp extends BooleanExp {
  private BooleanExp leftExp;
  private BooleanExp rightExp;
  AndExp(BooleanExp left, BooleanExp right) {
   leftExp = left;
   rightExp = right;
  }
  boolean eval(Context c) {
   return leftExp.eval(c) && rightExp.eval(c);
  }
}

// analogous definitions for OrExp and NotExp
```

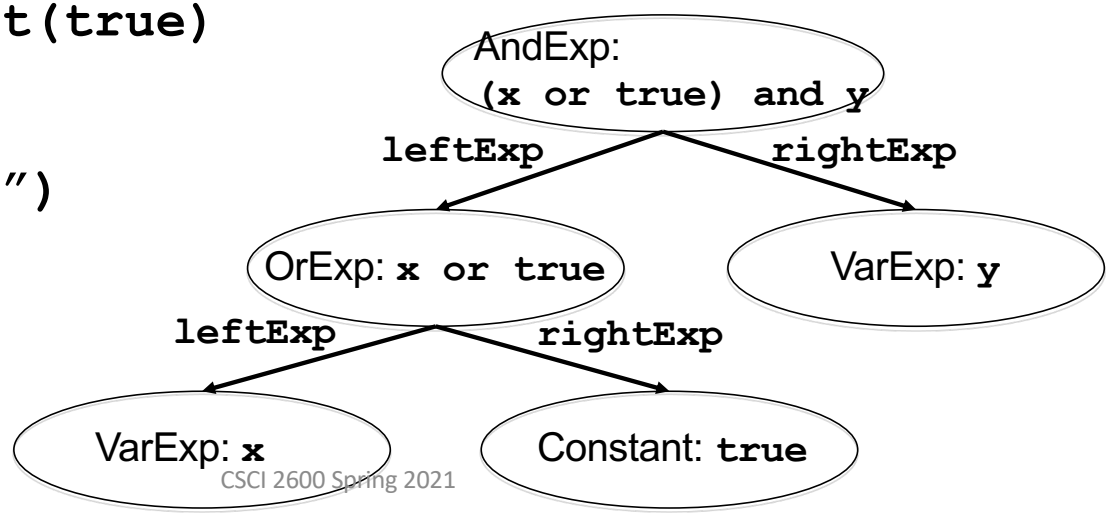
Composite Pattern: Class diagram



Object Structure

- Expression **`(x or true) and y`**

```
new AndExp(
    new OrExp(
      new VarExp("x"),
      new Constant(true)
    ),
    new VarExp("y")
)
```

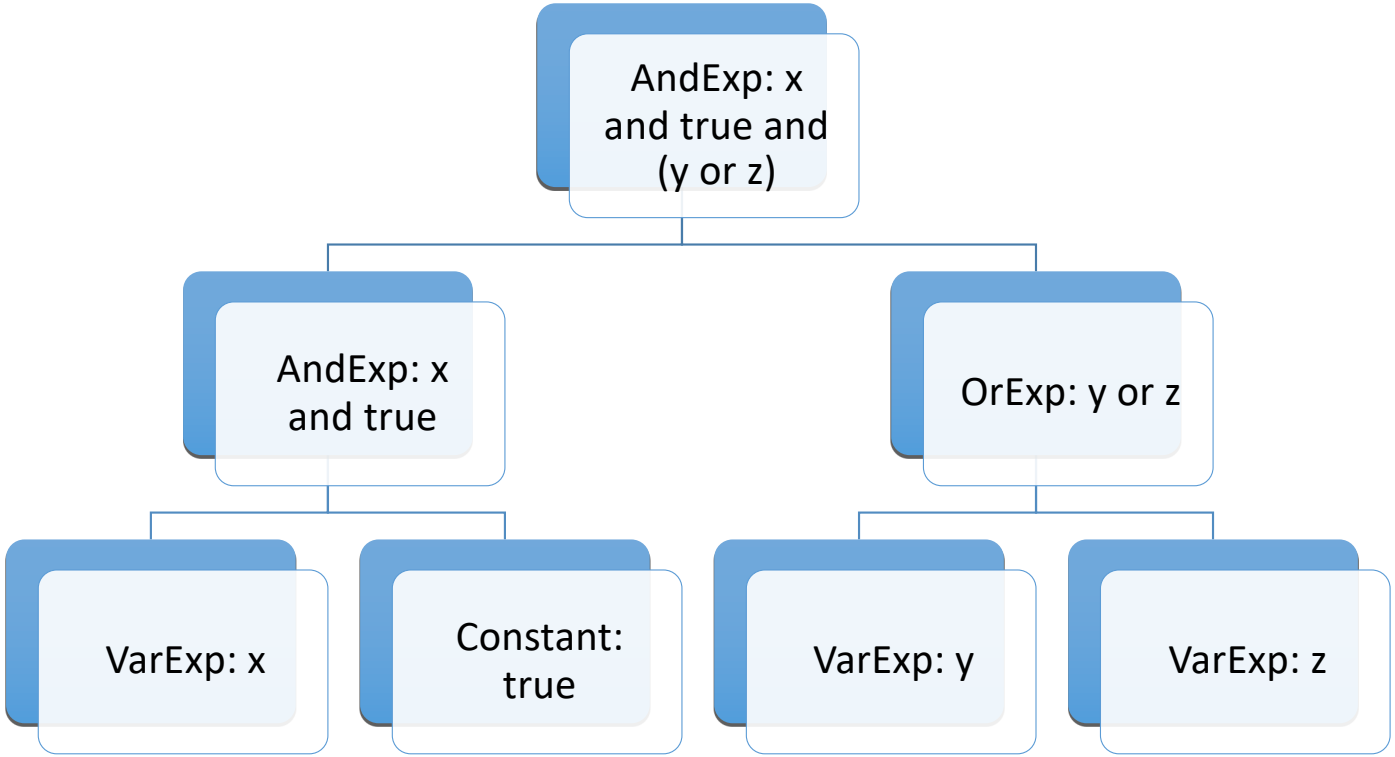


Exercise: Object Structure

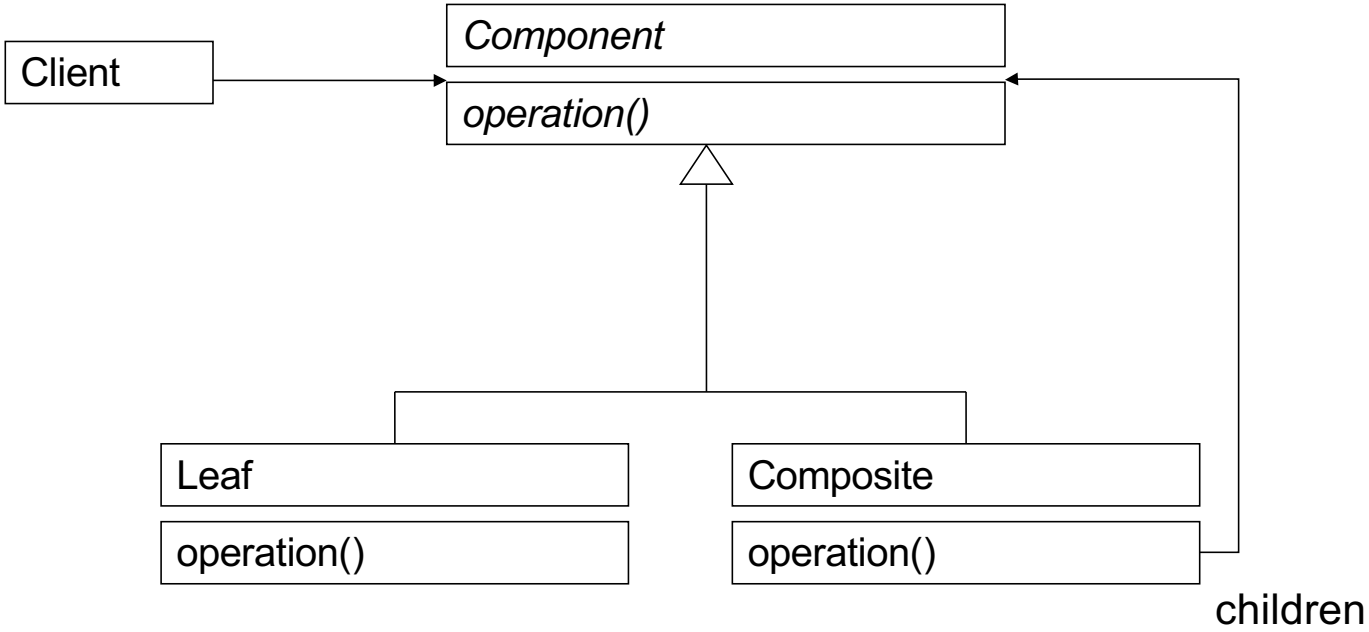
- Draw the object structure (a tree!) for expression (**`x and true) and (y or z)`**

```
(x and true) and (y or z)
```

new AndExp(new AndExp(new VarExp("x") new Constant(true)), new OrExp(new VarExp("y"), new VarExp("z")))



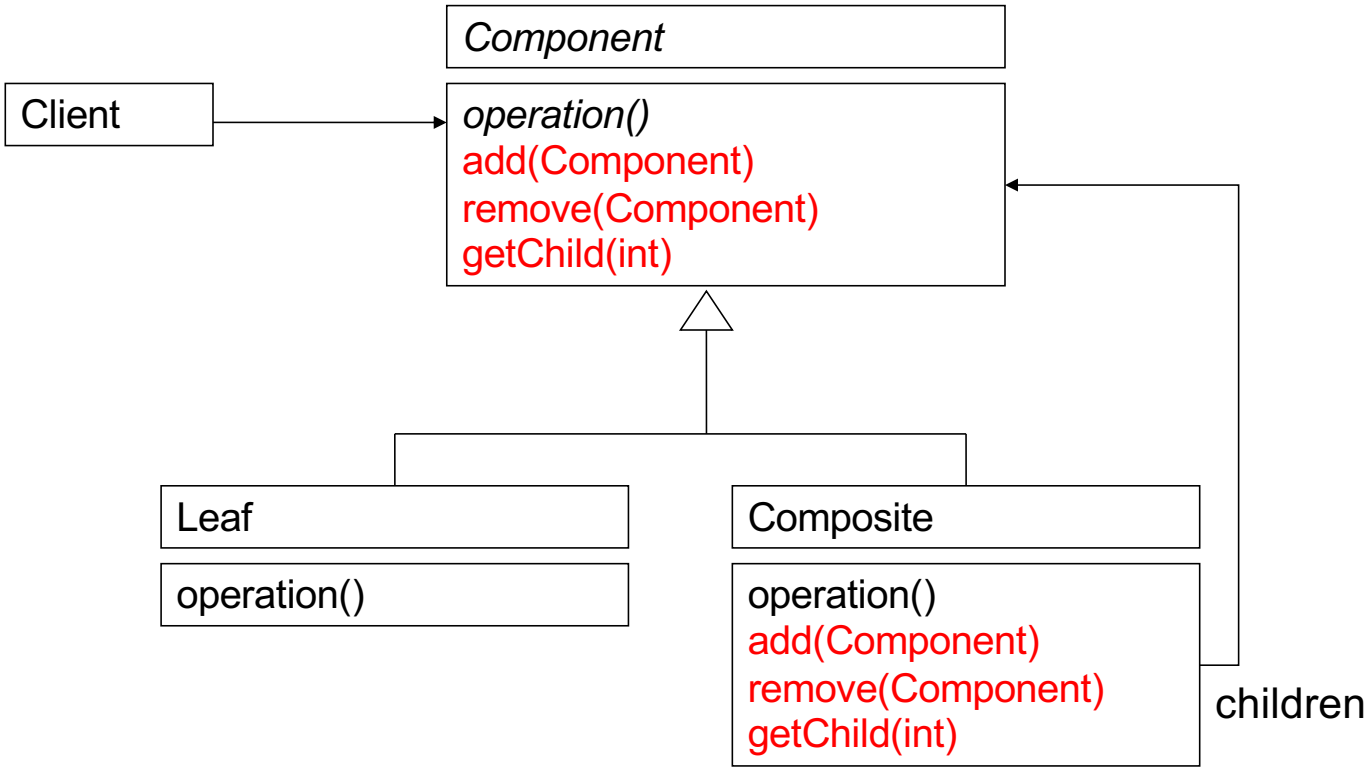
General Structure of Composite



Structure of Composite

- Component
  - Typically declared as an interface.
  - Implements default behavior for the interface common to all classes as appropriate.
  - Declares an interface for accessing and managing its child components.
- Leaf
  - Represents leaf objects. A leaf has no children.
  - Defines behavior for primitive objects in the composition.
- Composite
  - Defines behavior for components having children.
  - Stores child components.
  - Implements child related operations in the component interface.
- Client
  - Manipulates objects in the composition through the component interface.

Add Operations to Manage Composites

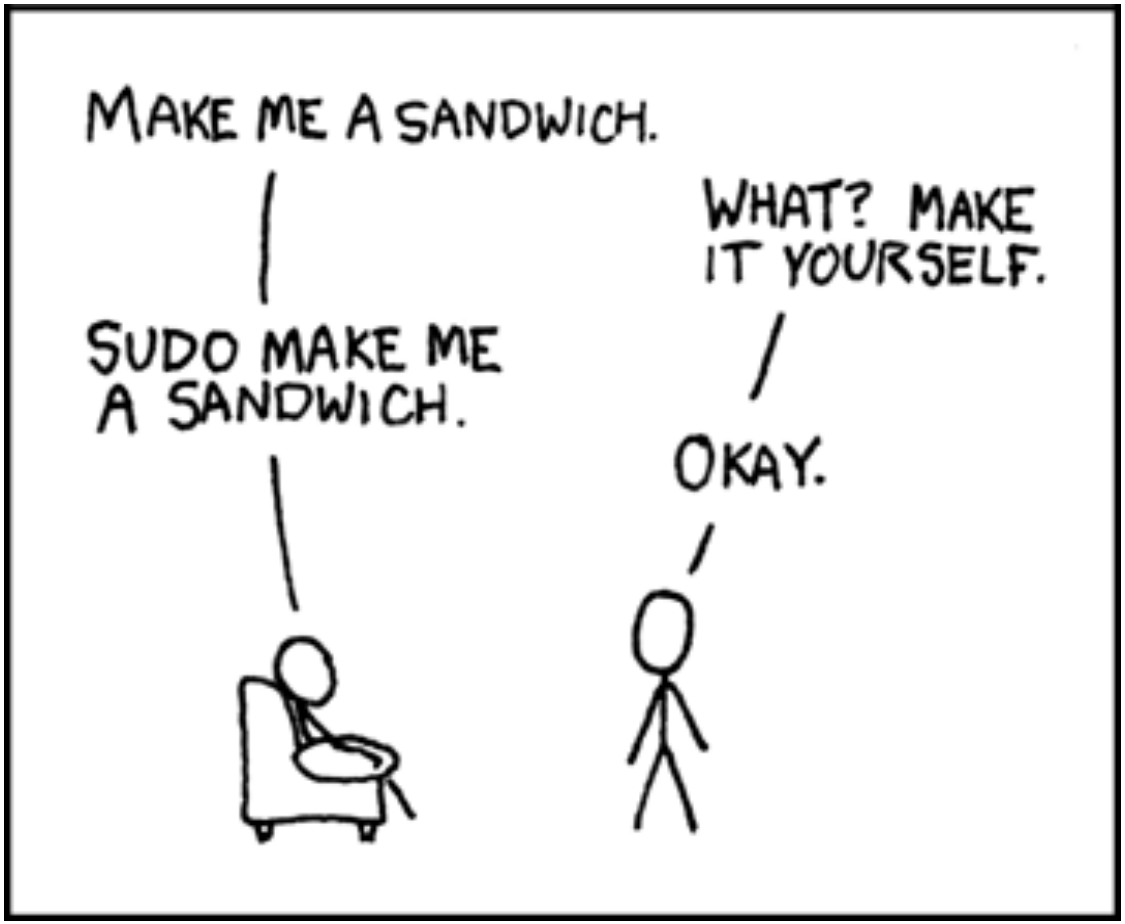


Decorators

- A GoF pattern
  - Adapter is a wrapper
  - Composite is not
- Decorators can add functionality without changing the interface
- When to use
  - Add to existing method to do something in addition while preserving the interface and spec
- Similar to subclassing
  - Not all subclassing is decoration

Decorators

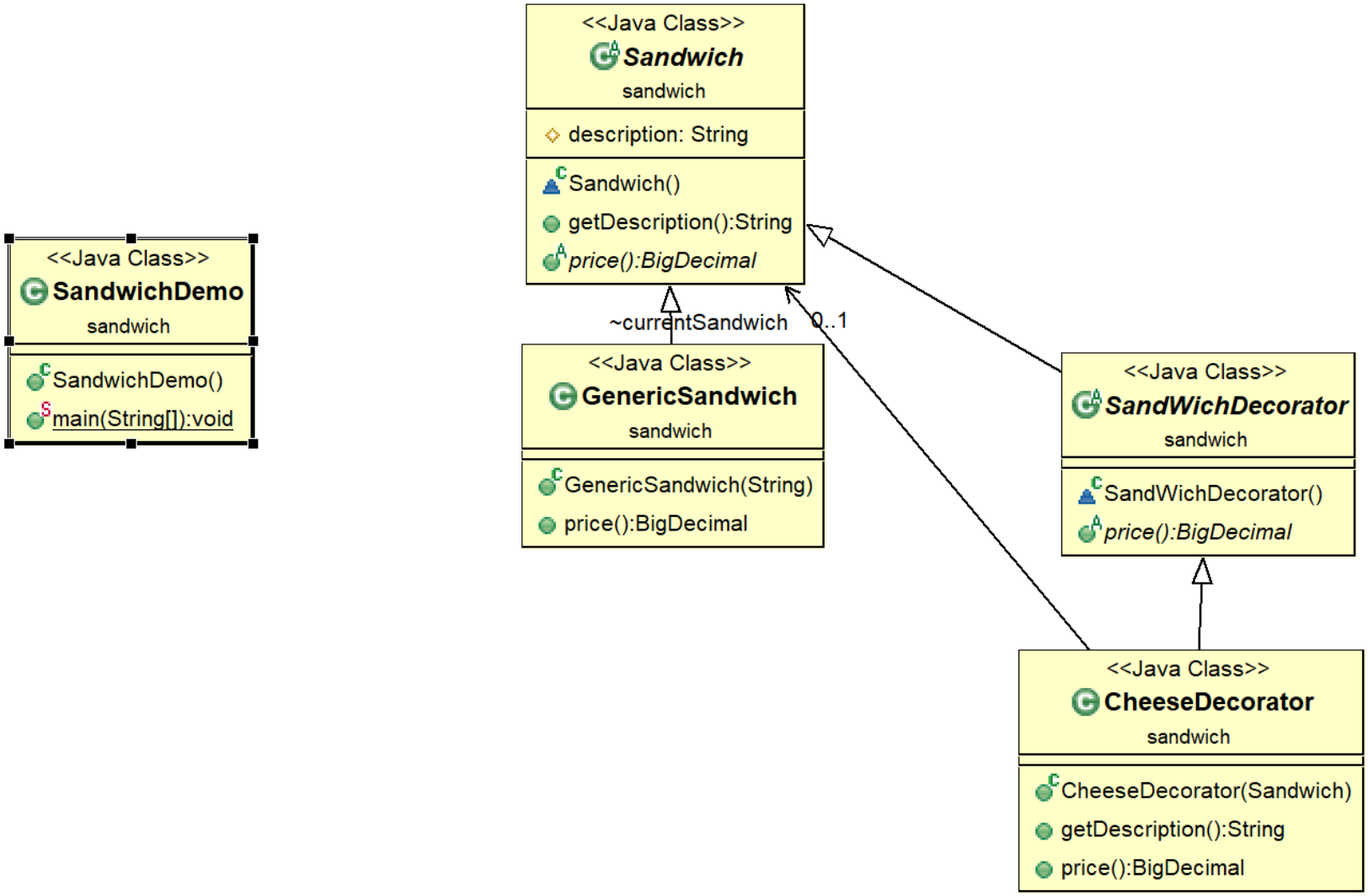
- The decorator pattern allows a user to add new functionality to an existing object without altering its structure.
- This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class method's signatures intact.



https://xkcd.com/149/

Decorator Example

- Sandwich class
  - We can have a basic sandwich – bread, filling
  - Might want to add cheese, tomato, lettuce, or other additions
    - decorations
  - We could make a subclass for each combination of additions
    - Each combo has a different price
    - Effective if we have a small number of combinations
  - Decorator pattern allows extension of the functionality of Sandwich at runtime, based upon customer's request
    - We can decorate a sandwich by adding ingredients and charging additional cost



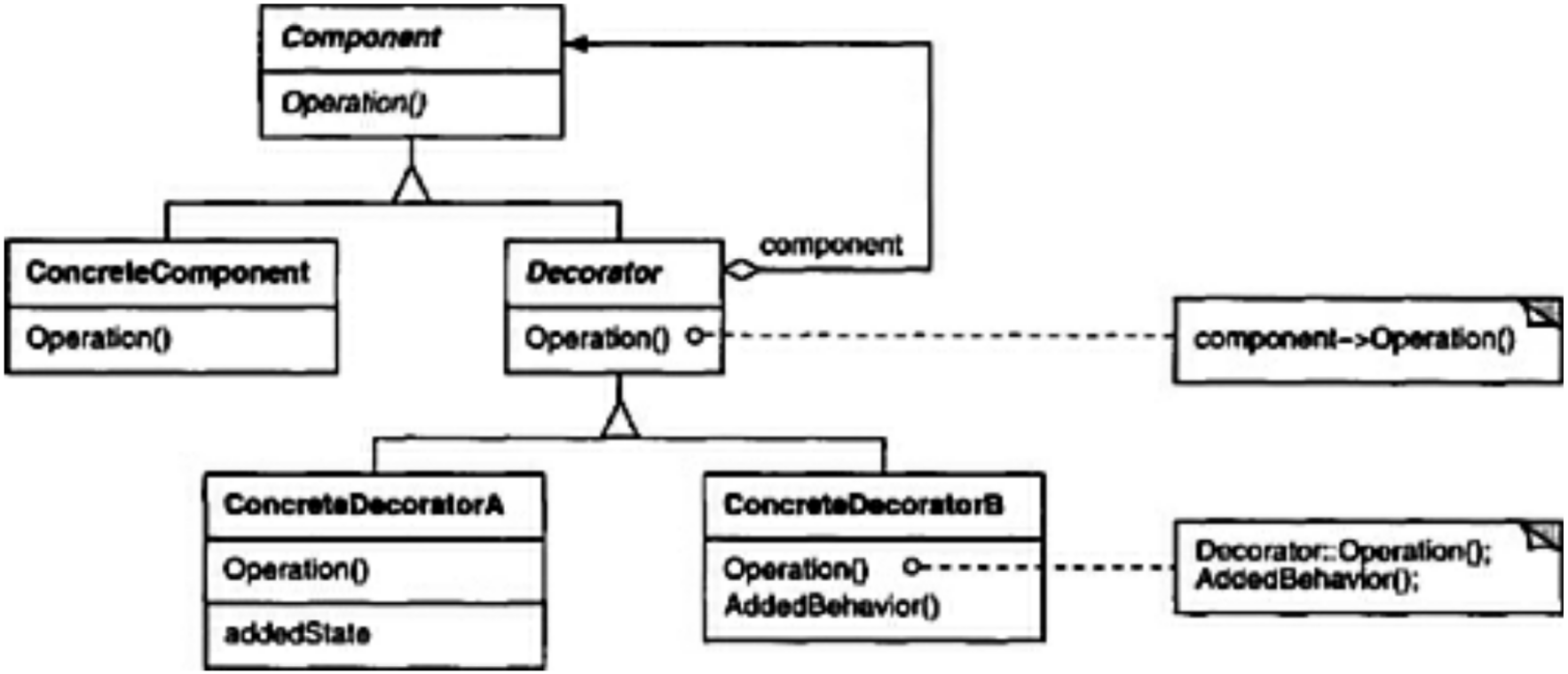
See SandwichDemo.java

Decorators

- The decorator must be of the same type of object, which they are decorating.
  - This can be achieved either by implementing the interface of the object or by extending an abstract class
- Decorator is usually based on Composition
  - This is achieved by creating a constructor of the decorator class which accepts a base type of original object.
  - In the example the constructor of CheeseDecorator accepts a Sandwich object.
- The Decorator pattern is an example of Open/Closed design principle
  - We don't have to change Sandwich to add cheese.
- The Decorator pattern affects objects at runtime

Structure of Decorator

Motivation: add small chunks of functionality without changing the interface



Example

```
abstract class Component { void draw(); }
class TextView extends Component {
  public void draw() {
    // Draw the TextView
  }
}
abstract class Decorator extends Component {
  private Component component;
  public Decorator(Component c) {
   this.component = c;
  }
  public void draw() {
   component.draw();
   … // additional functionality
  }
}
```

Example: Bordered Windows

Adds a border to the text view

```
class BorderDecorator extends Decorator {
  public BorderDecorator(Component c,
                         int borderwidth) {
     super(c);
     …
   }
  private void drawBorder() { … }
  public void draw() {
      super.draw();
     drawBorder();
   }
}
class ScrollDecorator extends Decorator {
   …
}
```

Calls **`Decorator.draw`** which redirects work to the enclosed component

Adds a scroll bar to the text view

Example

```
public class Client {
   public static void main(String[] args) {
      TextView textView = new TextView();
      Component decoratedComponent =
          new BorderDecorator(
             new ScrollDecorator(textView),1);
      }
      …
      decoratedComponent.draw();
      …
}
```

Bordered Windows: Another Version

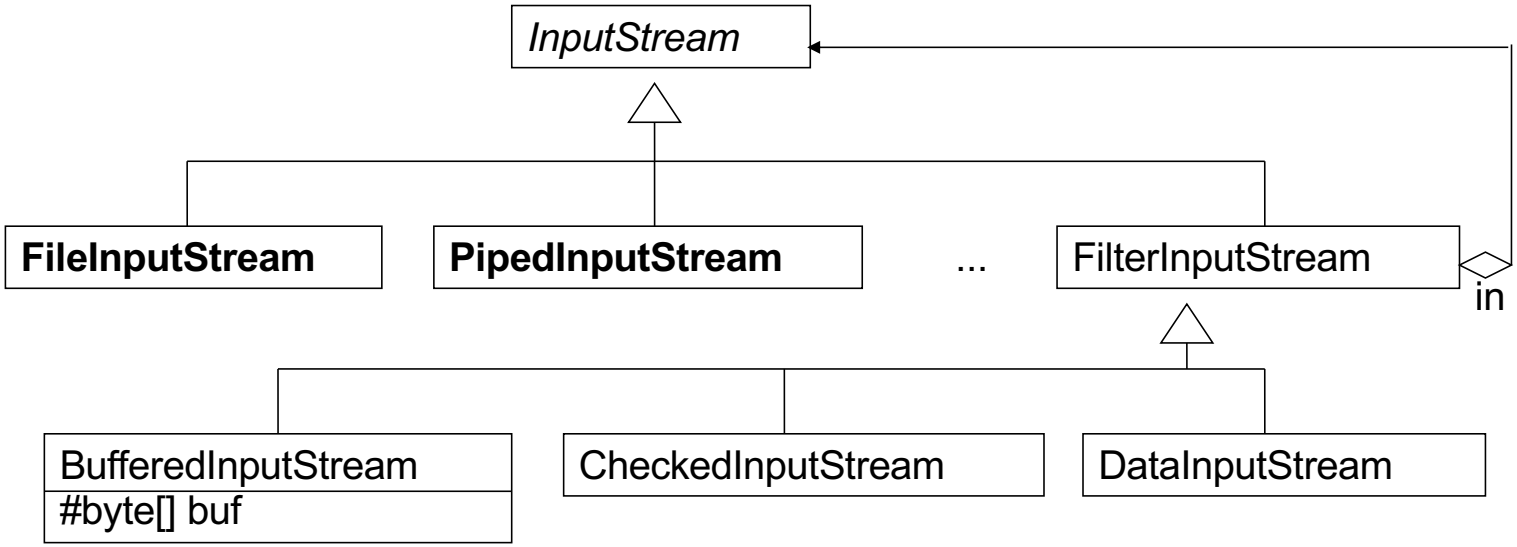
```
interface Window {
        // rectangle bounding the window
        Rectangle bounds();
        // draw this on the specified screen
        void draw(Screen s);
        ...
}
class WindowImpl implements Window {
        ...
}
```

Bordered Windows: Delegation

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

// 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);
        }
}
```

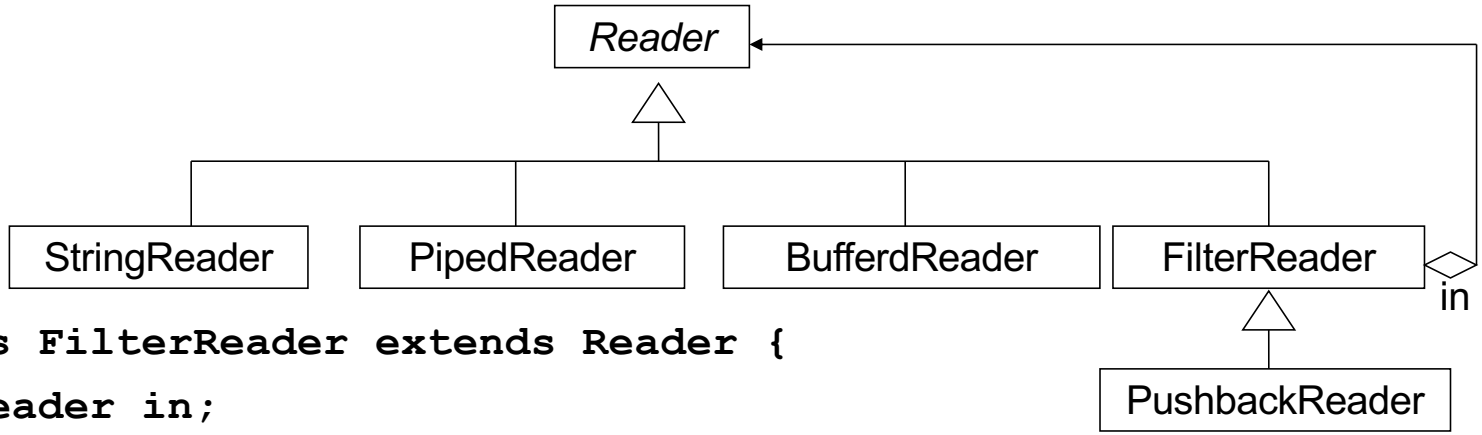
Delegation permits multiple borders on a window, or a window that is both bordered and shaded (or either one of those)

Java I/O Package
InputStream: byte input streams



- FilterInputStream is a Decorator. Enables the “chaining” of streams
- Each FilterInputStream redirects input action to the enclosed InputStream

Readers: character input streams




```
Class FilterReader extends Reader {
   Reader in;
   int read() {
     return in.read();
   }
   …
}
```

Another Decorator Example

```
public class UppercaseConvertor extends
                                FilterReader {
  public UppercaseConvertor(Reader in) {
    super(in);
  }

  public int read() throws IOException {
    int c = super.read();
    return ( c==-1 ? c :
            Character.toUpperCase((char)c));
  }

}
```

We also have **`LowercaseConverter extends FilterReader`**, which (surprise!) converts to lowercase

Another Decorator Example

```
public static void main(String[] args)  {
 Reader f =
       new UppercaseConverter(
        new LowercaseConvertor(
            new StringReader(args[0])));
  int c;
 while ((c = f.read()) != -1)
     System.out.print((char)c);
 System.out.println();
}
```

What is the object structure of **f**?

What does this code do?

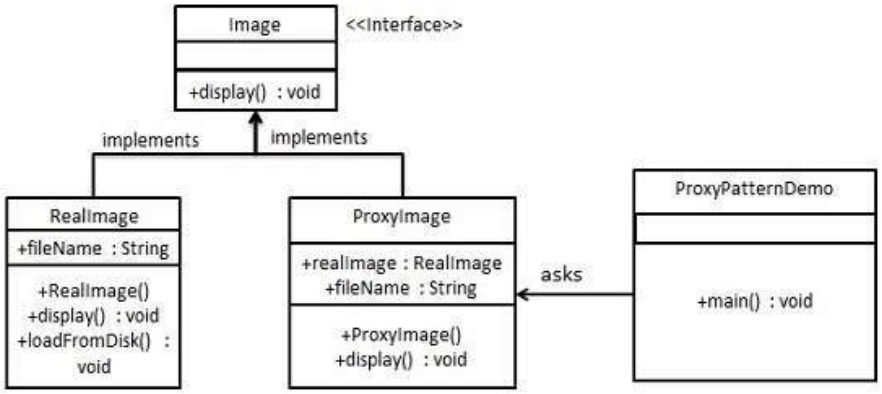
Aside: Convert Characters the functional way

```
public static List<String> convertToUpperCase8(List<String> inList) {
      return inList.stream() // Convert collection to Stream
            .map(String::toUpperCase) // Convert each element to upper case
            .collect(toList()); // Collect results to a new list
```

Proxy Pattern

- Same interface and functionality as the enclosed class
- Control access to enclosed object
  - 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 never used
    - Similar to singleton/interning

Recap: Simple Proxy Example



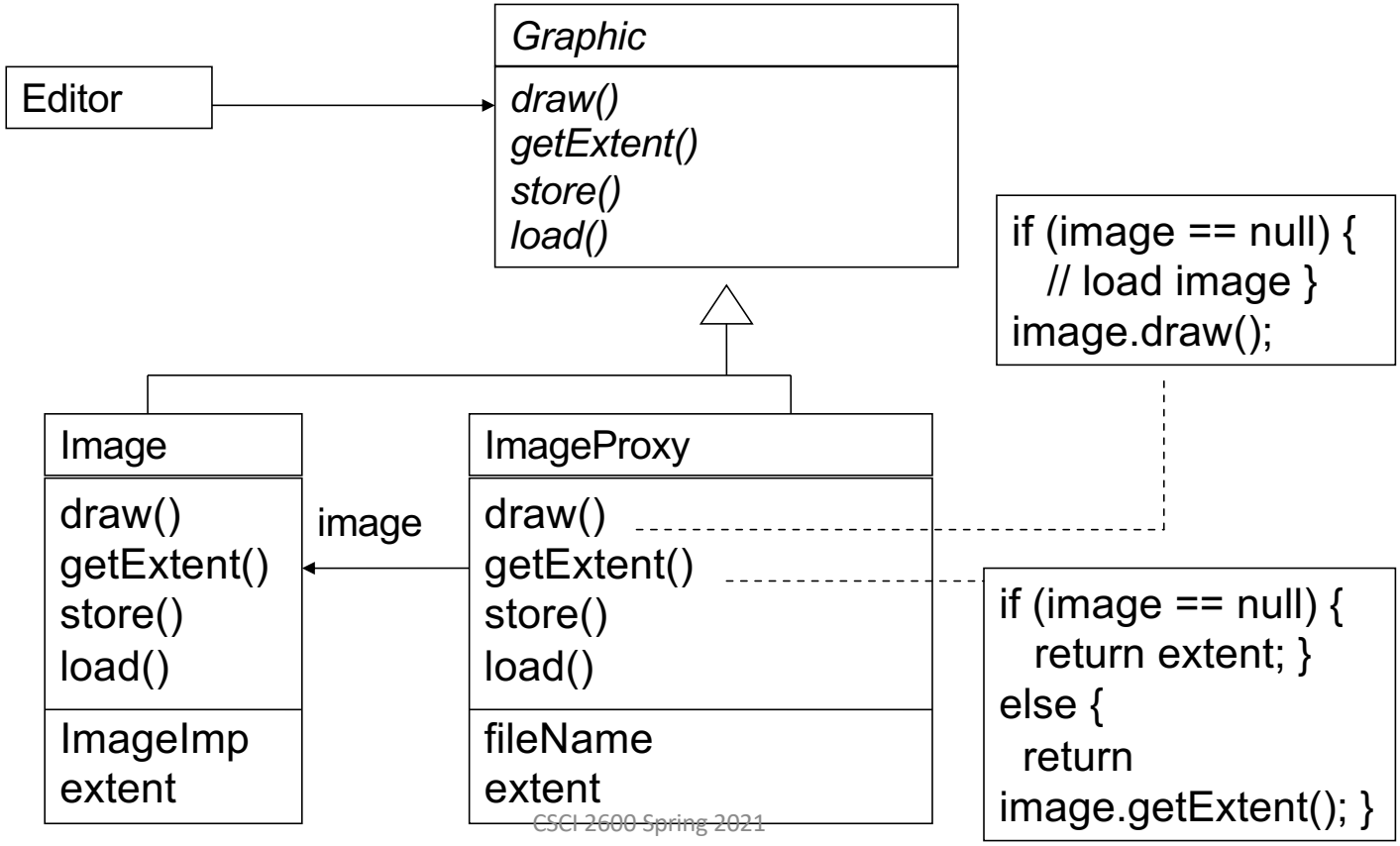
https://www.tutorialspoint.com/design_pattern/proxy_pattern.htm

```java
public interface Image {
  void display();
}

public class RealImage implements Image {

  private String fileName;

  public RealImage(String fileName){
    this.fileName = fileName;
    loadFromDisk(fileName);
  }

  @Override
  public void display() {
    System.out.println("Displaying " + fileName);
  }

  private void loadFromDisk(String fileName){
    System.out.println("Loading " + fileName);
  }
}
```

```java
public class ProxyImage implements Image{
  // delegation
  private RealImage realImage;
  private String fileName;

  public ProxyImage(String fileName){
    this.fileName = fileName;
  }

  @Override
  public void display() {
    if(realImage == null){
      realImage = new RealImage(fileName);
    }
    realImage.display();
  }
}
```

```
public class ProxyPatternDemo {

  public static void main(String[] args) {
    Image image = new ProxyImage("test_100mb.jpg");

    //image will be loaded from disk
    image.display();
    System.out.println("");

    //image will not be loaded from disk
    image.display();
  }
}
```

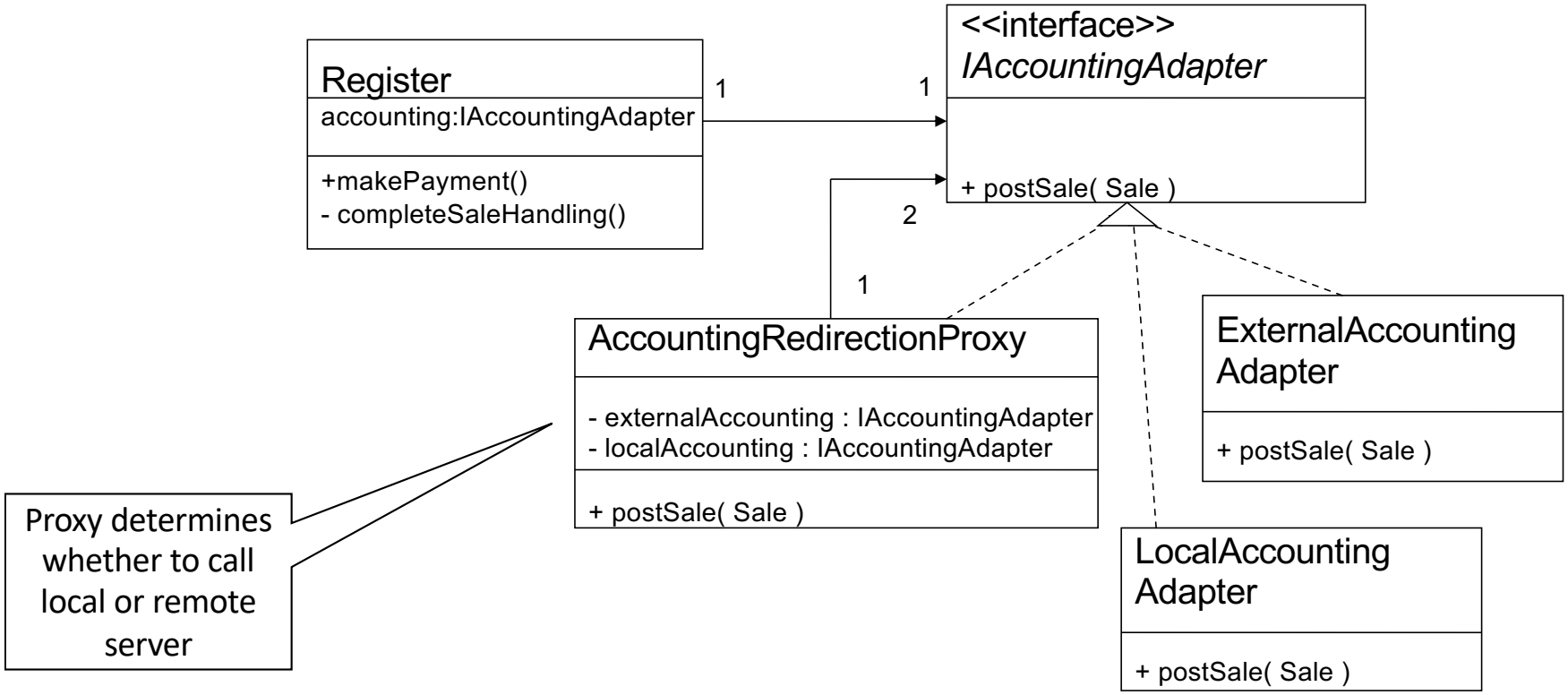
Proxy Example: Manage Creation of Expensive Object



Proxy Example: Manage Details When Dealing with Remote Object

- Recovery from remote service failure in the Point-Of-Sale system
  - When **`postSale`** is sent to an accounting service (remember, an **`AccountingAdapter`**), if connection cannot be established, failover to a local service
  - Failover should be transparent to **`Register`**
    - I.e., it should not know whether **`postSale`** was sent to the accounting service or to some special object that will redirect to a local service in case of failure

Proxy Example: Manage Details When Dealing with Remote Object



Traversing Composites

- Question: How to perform operations on all parts of a composite?
  - E.g., evaluate a boolean expression, print a boolean expression

Perform Operations on boolean expressions

- Need to write code for each Operation/Object pair
- Question: do we group together (in a class) the code for a particular operation or the code for a particular object?

Objects

Operations

	VarExp	Constant	AndExp	OrExp	NotExp
evaluate					
pretty-print					

Interpreter and Procedural Patterns

- Interpreter: groups code per object, spreads apart code for similar operations

- Procedural: groups code per operation, spreads apart code for similar objects

	VarExp	AndExp
evaluate		
pretty-print		

	VarExp	AndExp
evaluate		
pretty-print		

Interpreter Pattern

	VarExp	AndExp
evaluate		
pretty-print		

```
abstract class BooleanExp {
  abstract boolean eval(Context c);
  abstract String prettyPrint();
}

class VarExp extends BooleanExp {
  …
  boolean eval(Context c);
  String prettyPrint();
}

class AndExp extends BooleanExp {
  boolean eval(Context c);
  String prettyPrint();
}
```

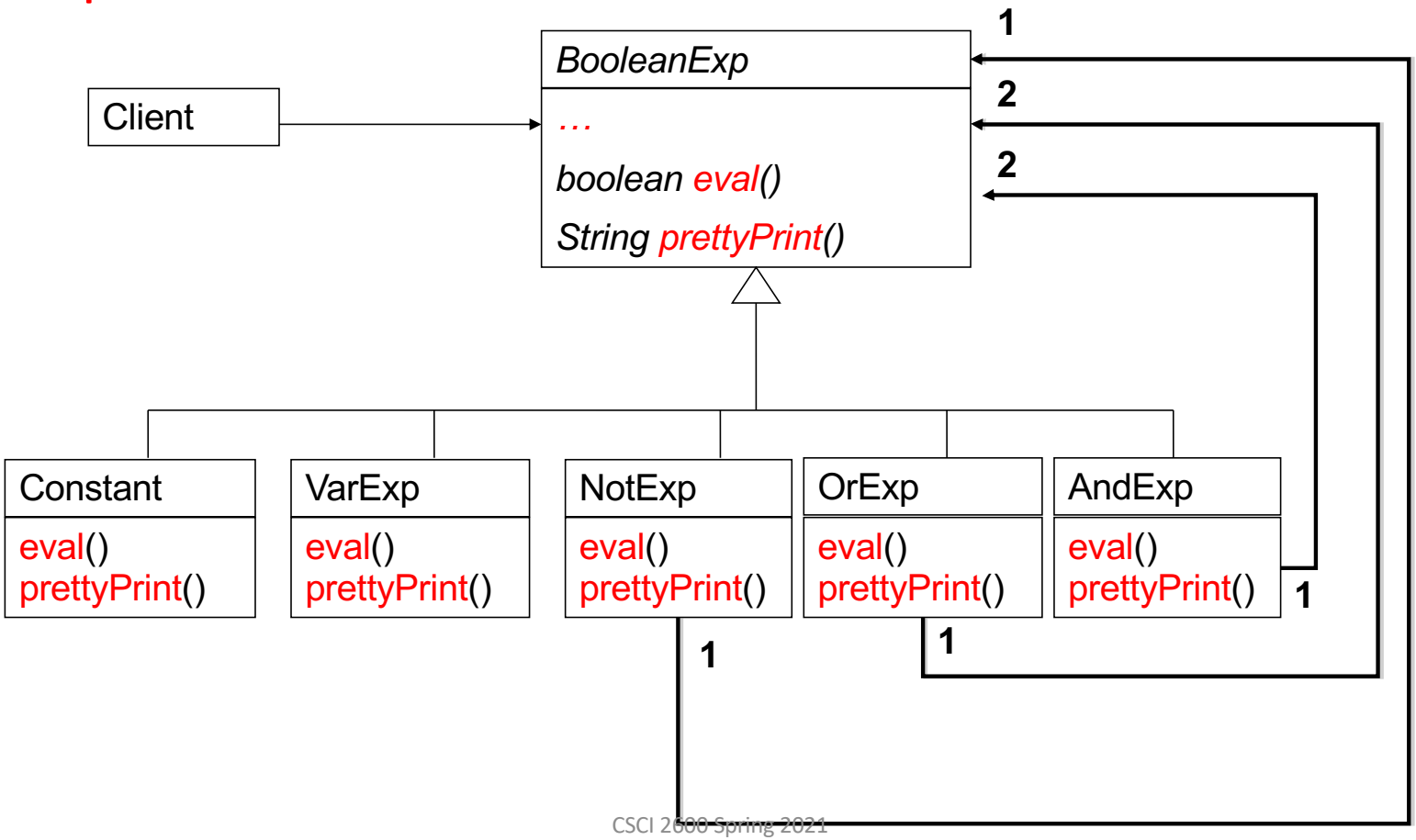
Add a method to each class for each supported operation

Dynamic dispatch chooses right implementation at call `myExpr.eval(c);`

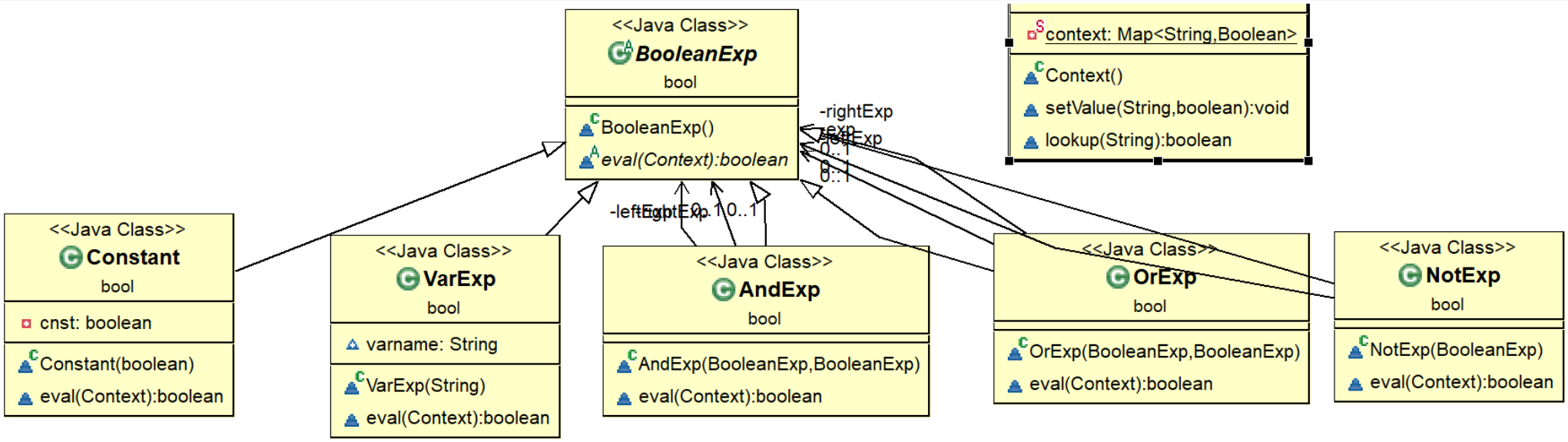
Interpreter Pattern

```
class AndExp extends BooleanExp {
  private BooleanExp leftExp;
  private BooleanExp rightExp;
  AndExp(BooleanExp left, BooleanExp right) {
   leftExp = left;
   rightExp = right;
  }
  boolean eval(Context c) {
   return leftExp.eval(c) && rightExp.eval(c);
  }
  String prettyPrint() {
       leftExp.prettyPrint(); rightExp.prettyPrint();
  }
}

// analogous definitions for OrExp and NotExp
```

Interpreter Pattern



Interpreter Pattern



See BooleanDemo.java

Procedural pattern

	VarExp	AndExp
evaluate		
pretty print		

```
// Classes for expressions don't have eval

class Evaluate {
  boolean evalConstExp(Constant c) {
    c.value(); // returns value of constant
  }

  boolean evalAndExp(AndExp e) {
    BooleanExp leftExp = e.leftExp;
    BooleanExp rightExp = e.rightExp;

    //Problem: How to invoke the right
    //implementation for leftExp and rightExp?
  }
}
```

Procedural pattern

	VarExp	AndExp
evaluate		
pretty print		

```
// Classes for expressions don't have eval

class Evaluate {
  Context c;
  …
  boolean evalExp(BooleanExp e) {
    if (e instanceof VarExp)
      return evalVarExp((VarExp) e);
    else if (e instanceof Constant)
      return evalConstExp((VarExp) e);
    else if (e instanceof OrExp)
      return evalOrExp((OrExp)e);
    else …
  }
}
```

What is the problem with this code?

Visitor Pattern, a variant of the Procedural pattern

- Visitor helps traverse a hierarchical structure
- Nodes (objects in the hierarchy) accept visitors
- Visitors visit nodes (objects)

```
class SomeBooleanExp extends BooleanExp {
  void accept(Visitor v) {
    for each child of this node {
      child.accept(v);
    }
    v.visit(this);
  }
}
class Visitor {
  void visit(SomeBooleanExp e) { do work on e }
}
```

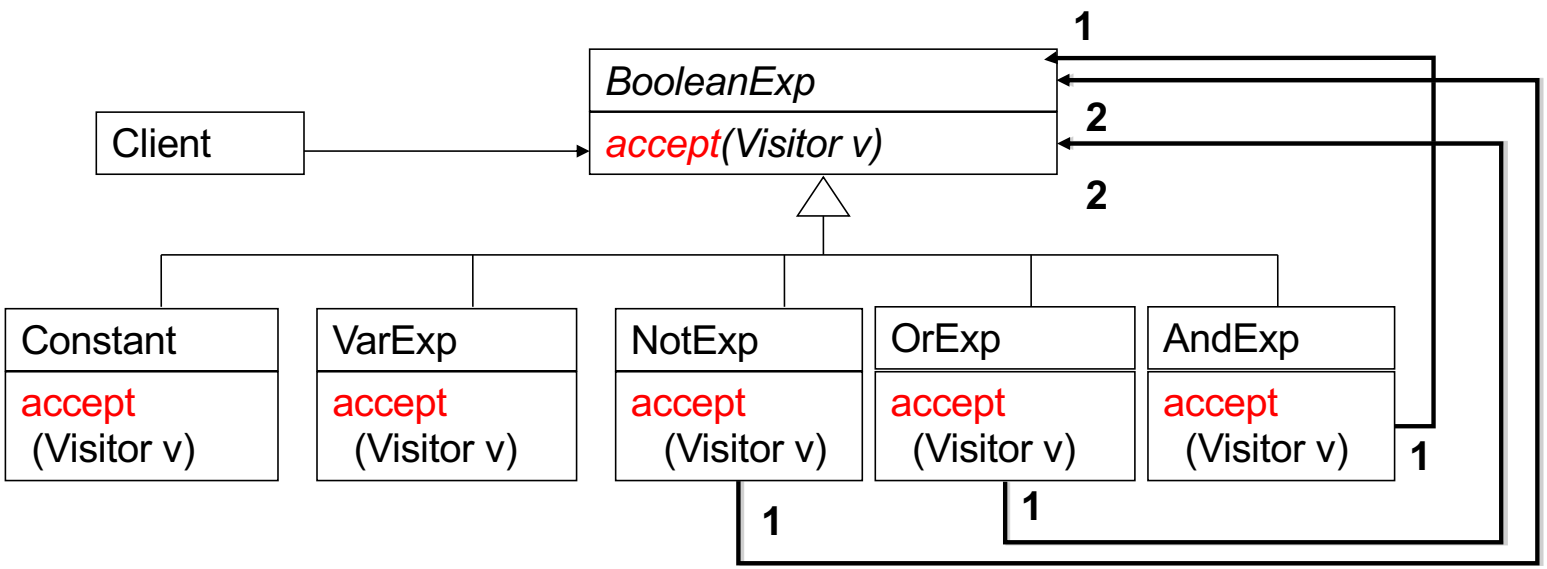
`n.accept(v)` traverses the structure root at **`n`**, performing **`v`**'s operation on every element

Visitor Pattern

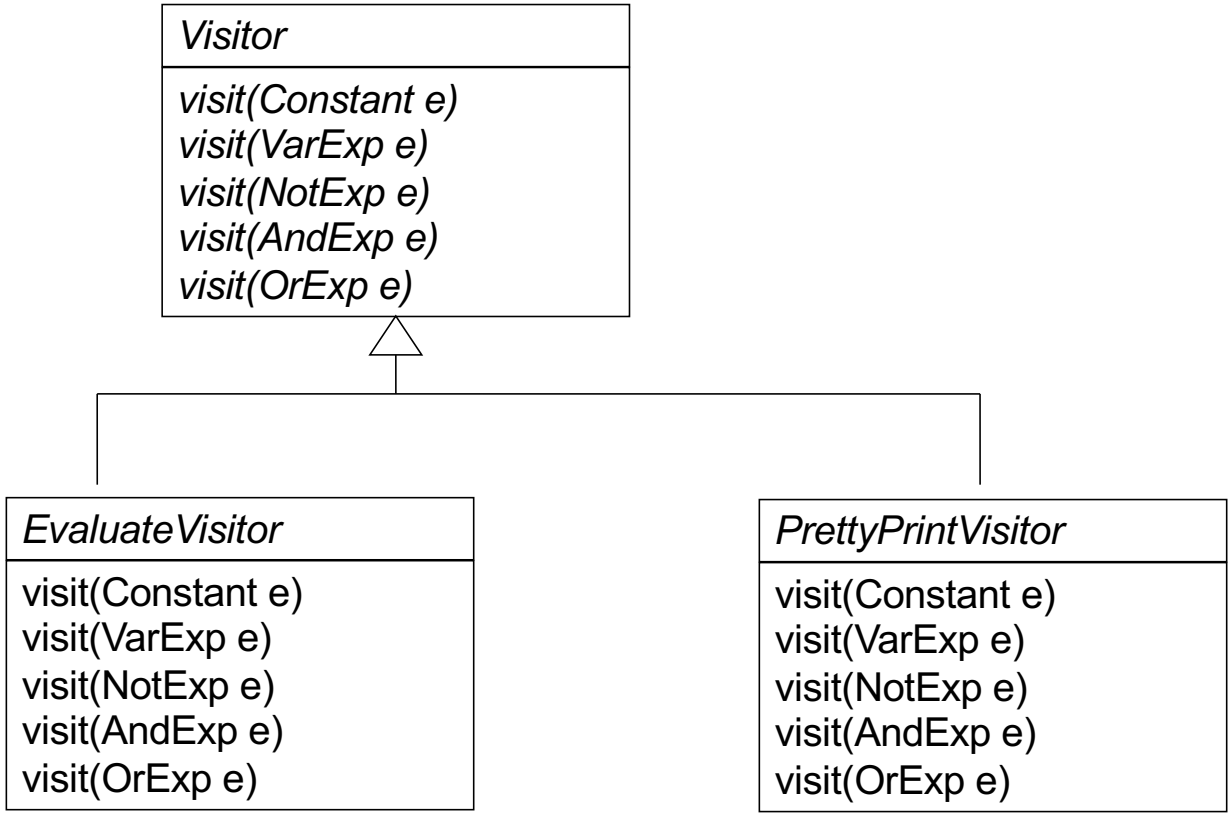
```
class VarExp extends
          BooleanExp {
 void accept(Visitor v) {
  v.visit(this);
 }
}
class AndExp extends
           BooleanExp {
 BooleanExp leftExp;
 BooleanExp rightExp;
 void accept(Visitor v) {
  leftExp.accept(v);
  rightExp.accept(v);
  v.visit(this);
 }
}
```

```
class Evaluate
  implements Visitor {
 void visit(VarExp e)
 {
  //evaluate var exp
 }
 void visit(AndExp e)
 {
  //evaluate And exp
 }
}
class PrettyPrint
  implements Visitor {
…
}
```

The Visitor Pattern



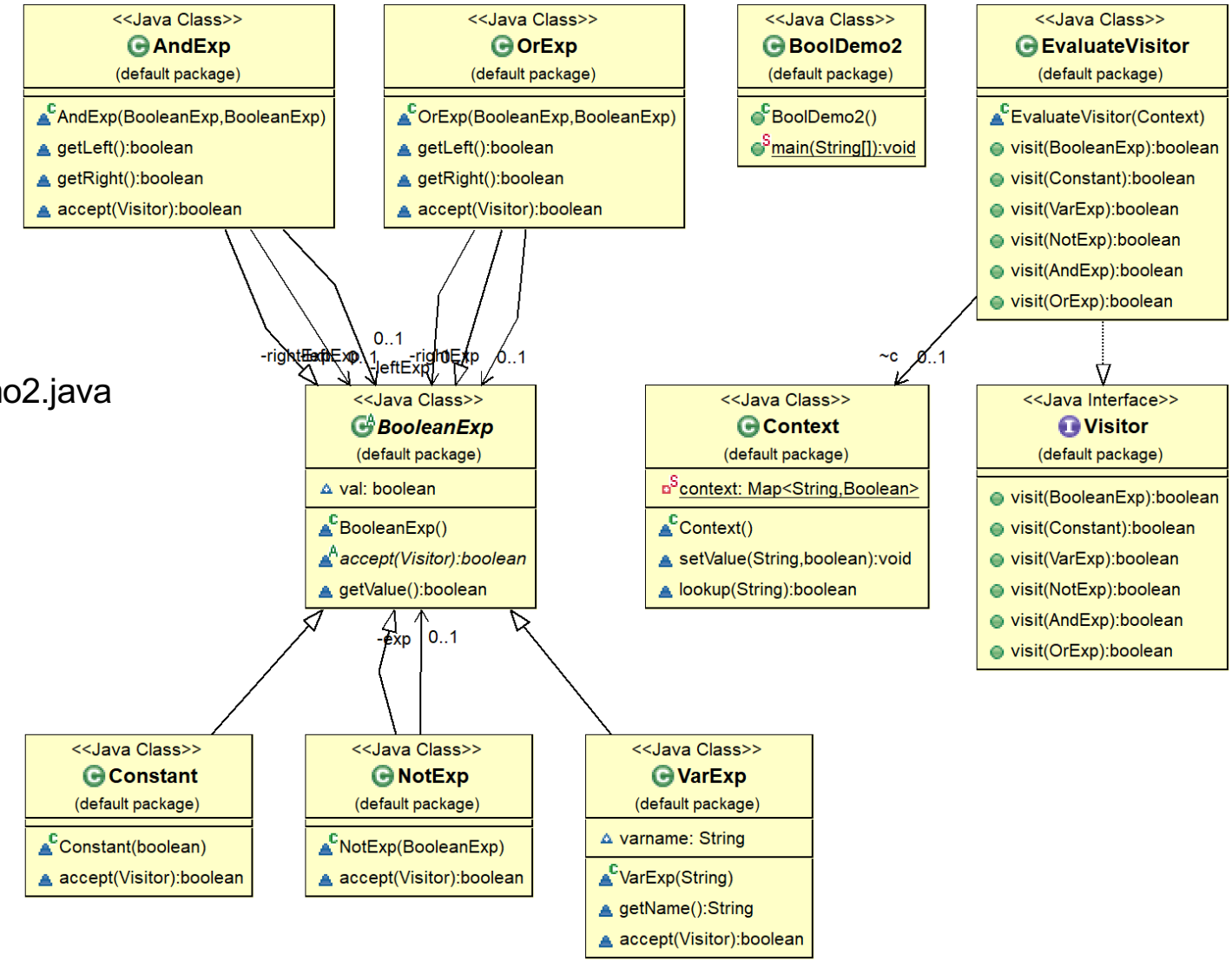
The Visitor Pattern



Visitor Pattern

- Must add definitions of visit (in Visitor hierarchy) and accept (in Object hierarchy)
- visit may do many different things: evaluate, count nodes, pretty print, etc.
- It is easy to add operations (a new Visitor class), hard to add nodes (must modify entire hierarchy of Visitors)
- Visitors are similar to iterators but different because they have knowledge of structure not just sequence

See BoolDemo2.java



Visitor Pattern Dispatch

- In BoolDemo2
- exp.accept(new EvaluateVisitor(c))
  - We pass exp's accept() method an instance of an evaluator
  - OrExp passes the evaluator to its children, when done, it calls the evaluator's visit method which calculates the result.
- To add PrettyPrint, we need to add a PrettyPrint Visitor which does the printing. PrettyPrint would implement Visitor.
  - Open/closed principle

Visitor Pattern's
Double Dispatch

	VarExp	AndExp
evaluate		
pretty-print		

```
myExp.accept(v): we want to choose the right operation
myExp.accept(v) // dynamically dispatch the right
                // implementation of accept, e.g., AndExp.accept
class AndExp {
    void accept(Visitor v) {
      …
          v.visit(this); // at compile-time, chooses
    }                    // visit(AndExp). At
}        // runtime, dispatches the right implementation of
    // visit(AndExp), e.g.,
    // EvaluateVisitor.visit(AndExp)
```

Design Patterns Summary so Far

- Factory method, Factory class, Prototype
  - Creational patterns: address problem that constructors can't return subtypes
- Singleton, Interning
  - Creational patterns: address problem that constructors always return a new instance of class
- Wrappers: Adapter, Decorator, Proxy
  - Structural patterns: when we want to change interface or functionality of an existing class, or restrict access to an object

Design Patterns Summary so Far

- Composite
  - A structural pattern: expresses whole-part structures, gives uniform interface to client
- Interpreter, Procedural, Visitor
  - Behavioral patterns: address the problem of how to traverse composite structures