Structural Patterns

Composite

Decorator

Composite

- Intent
  - Compose objects into **tree** structures to represent **part-whole** hierarchy.
  - Lets clients **treat** individual objects and composition objects ***uniformly***
- Applicability
  - Want to represent part-whole hierarchy
  - Want clients to be able **to ignore the difference** between composites and individuals

Example 1: Company Structure

- Organizational hierarchy btw Employees
  - Each worker has a manager
  - Each manager has a manager
    - Unless they are the Head of the company
- Tree structure, let's build a tree
- Not enough…

Example 1: Company Structure

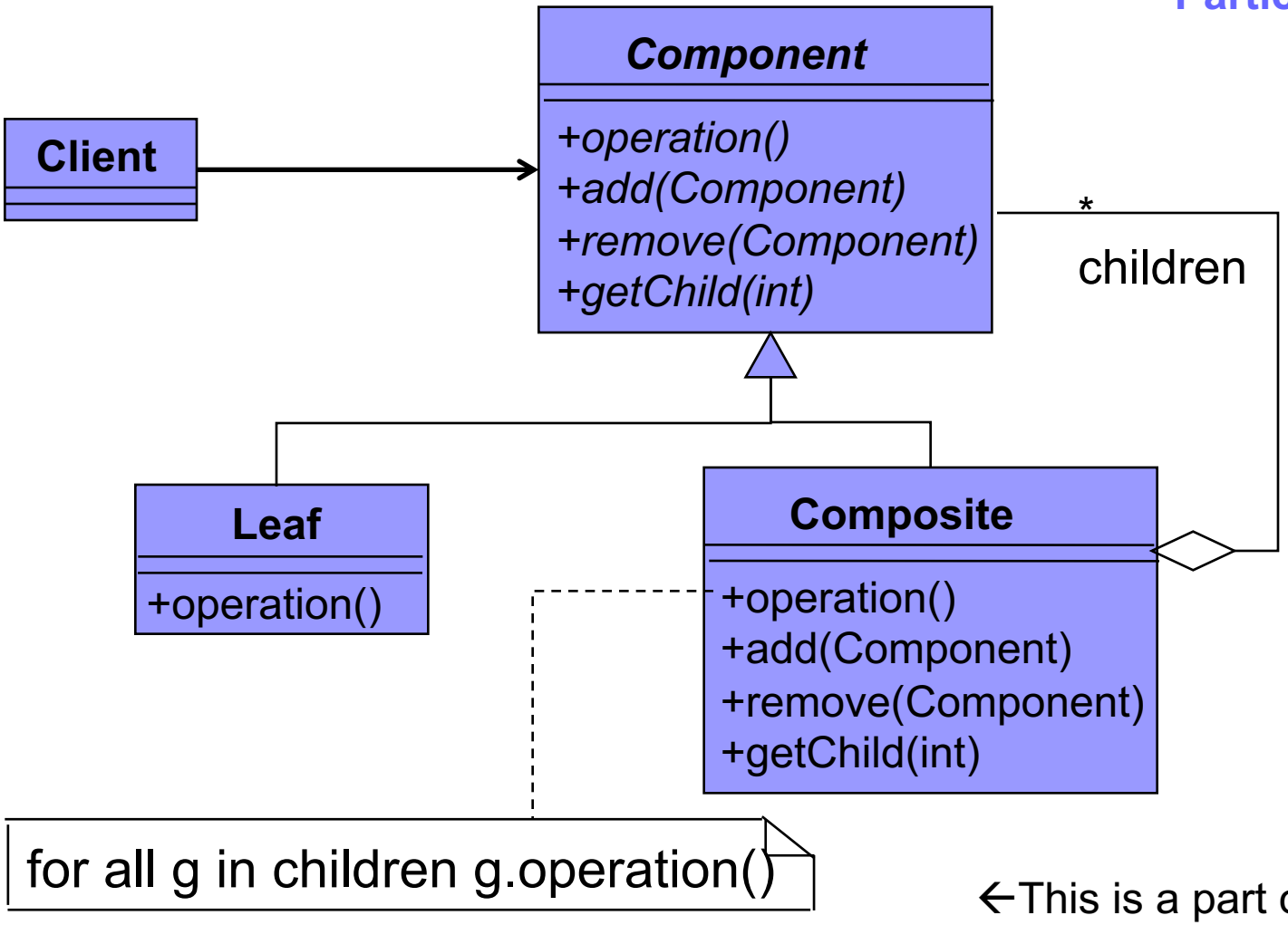
- Organizational hierarchy btw Employees
  - Each worker has a manager
  - Each manager has a manager
- Tree structure, let's build a tree
- Not enough…
- I want to apply the same operations over both composite and individual objects
  - Say "move to 2nd floor" to a worker
  - Say "move to 2nd floor" to a manager to move their people to 2nd floor

Example 1: Company Structure

- Tree structure
  - Node class which as children as other nodes
- Client code should be able to request a leaf or a node uniformly
  - employee.print()
    - prints the subtree if the employee is a manager
    - prints the worker info if the employee is not a manager
  - employee.move()
  - employee.printVacationDays()
- the primitive and composite objects are treated alike

Composite - Structure

Participants?

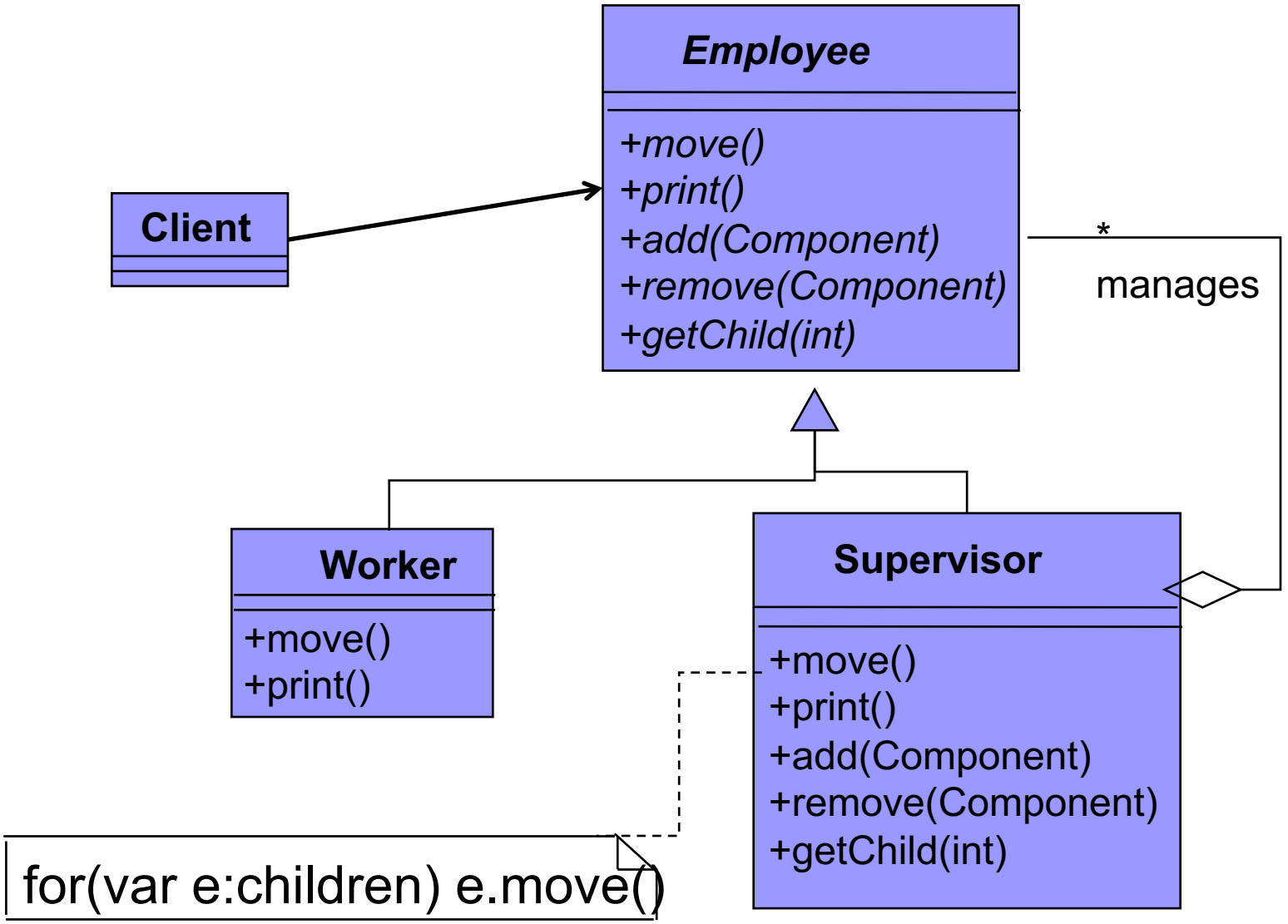


←This is a part of the pattern

Collaboration & Participants

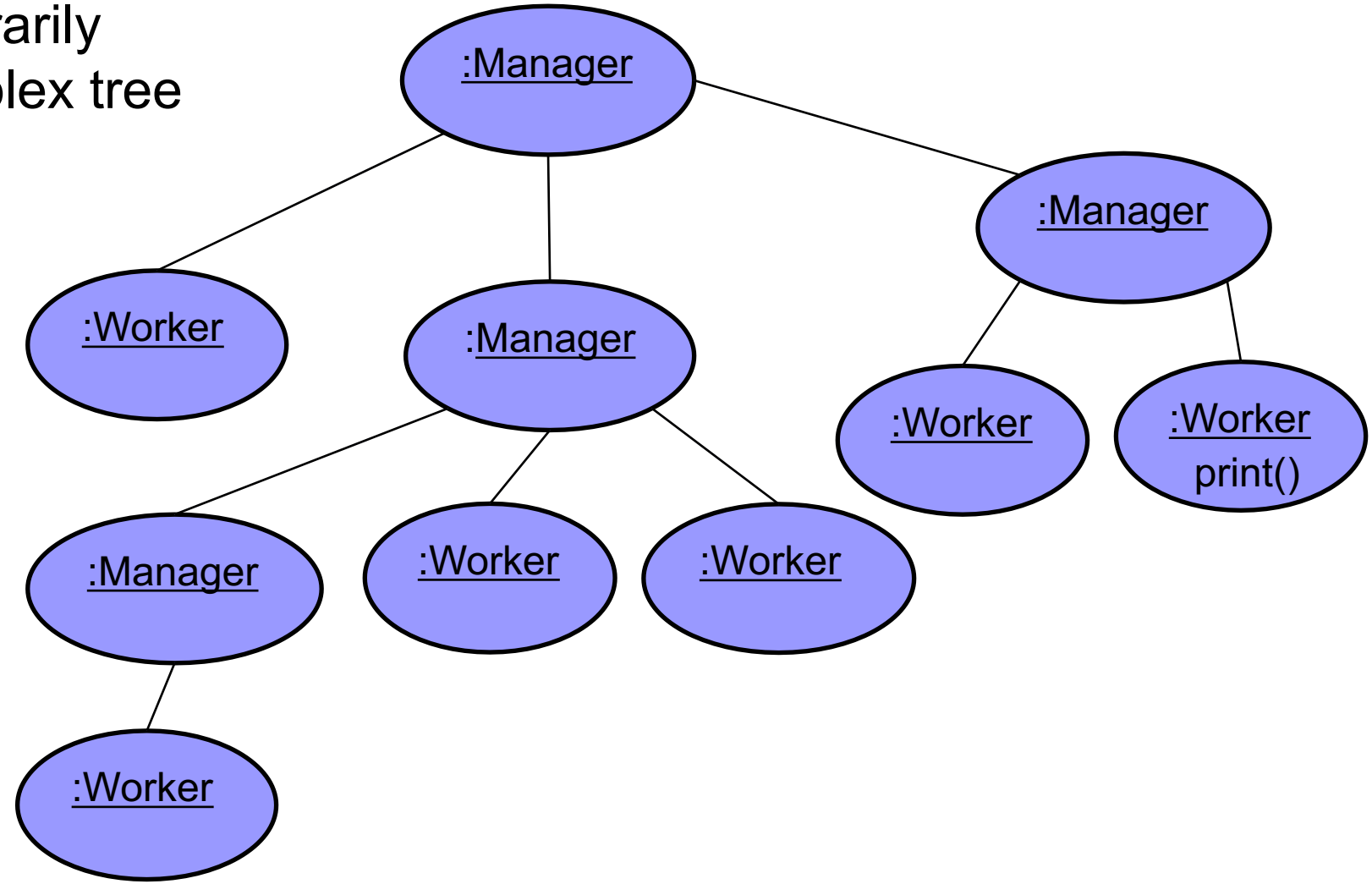
- Clients use the Component class interface to interact with objects in the composite structure.
- If the recipient is a Leaf, then the request is handled directly.
- If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Composite Company



Composite company

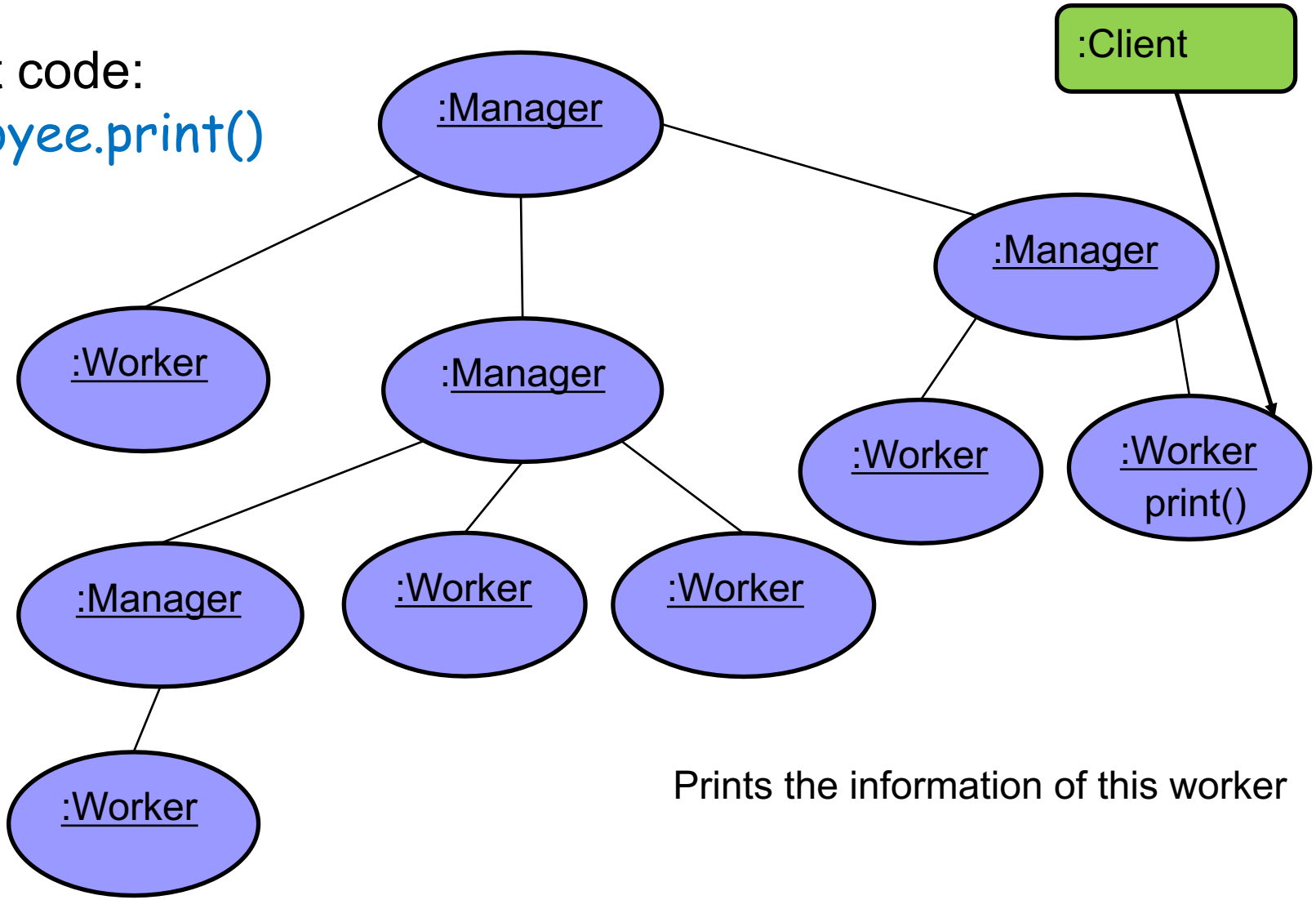
Arbitrarily complex tree



Composite company

Client code:

employee.print()

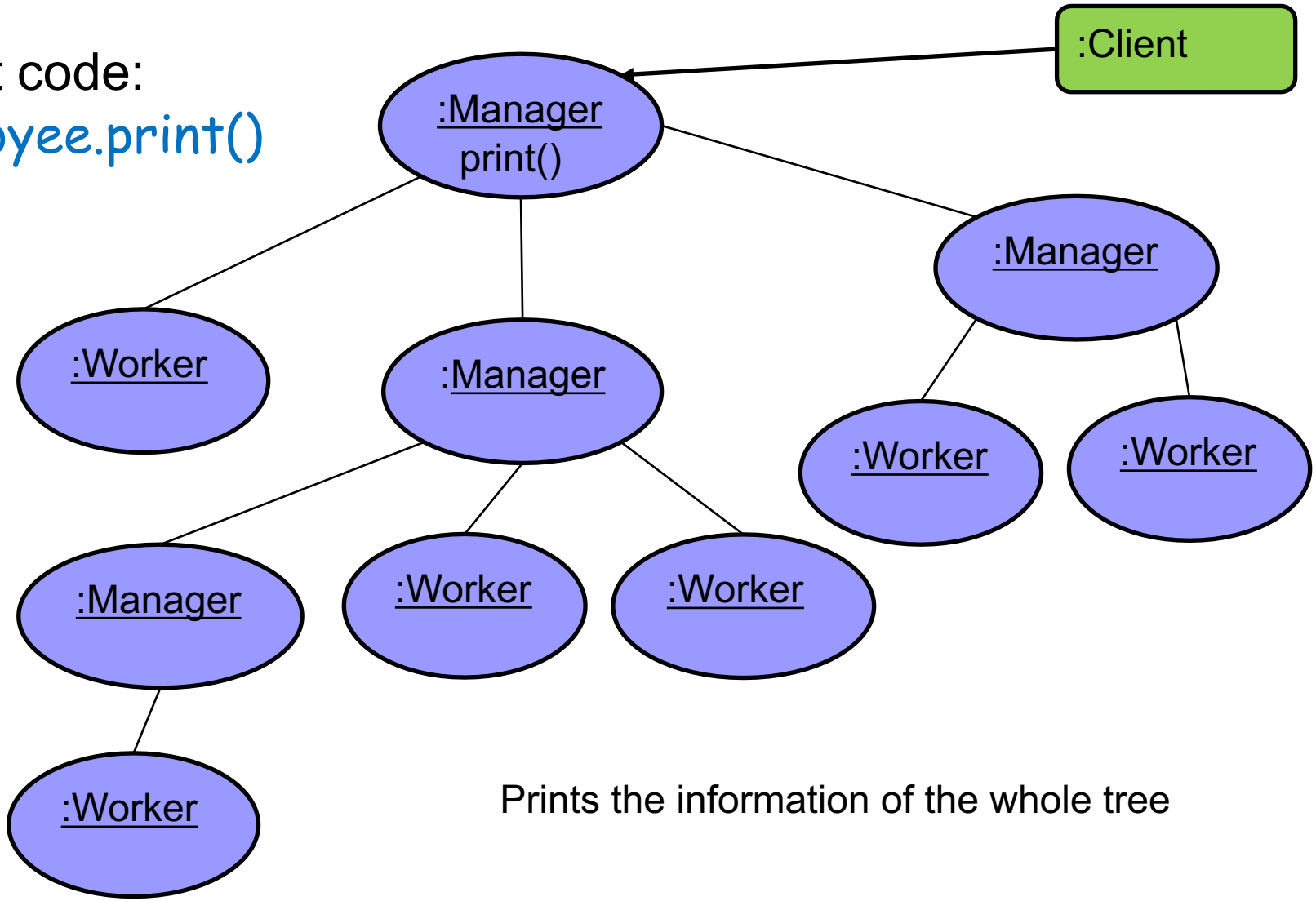


Prints the information of this worker

Composite company

Client code:

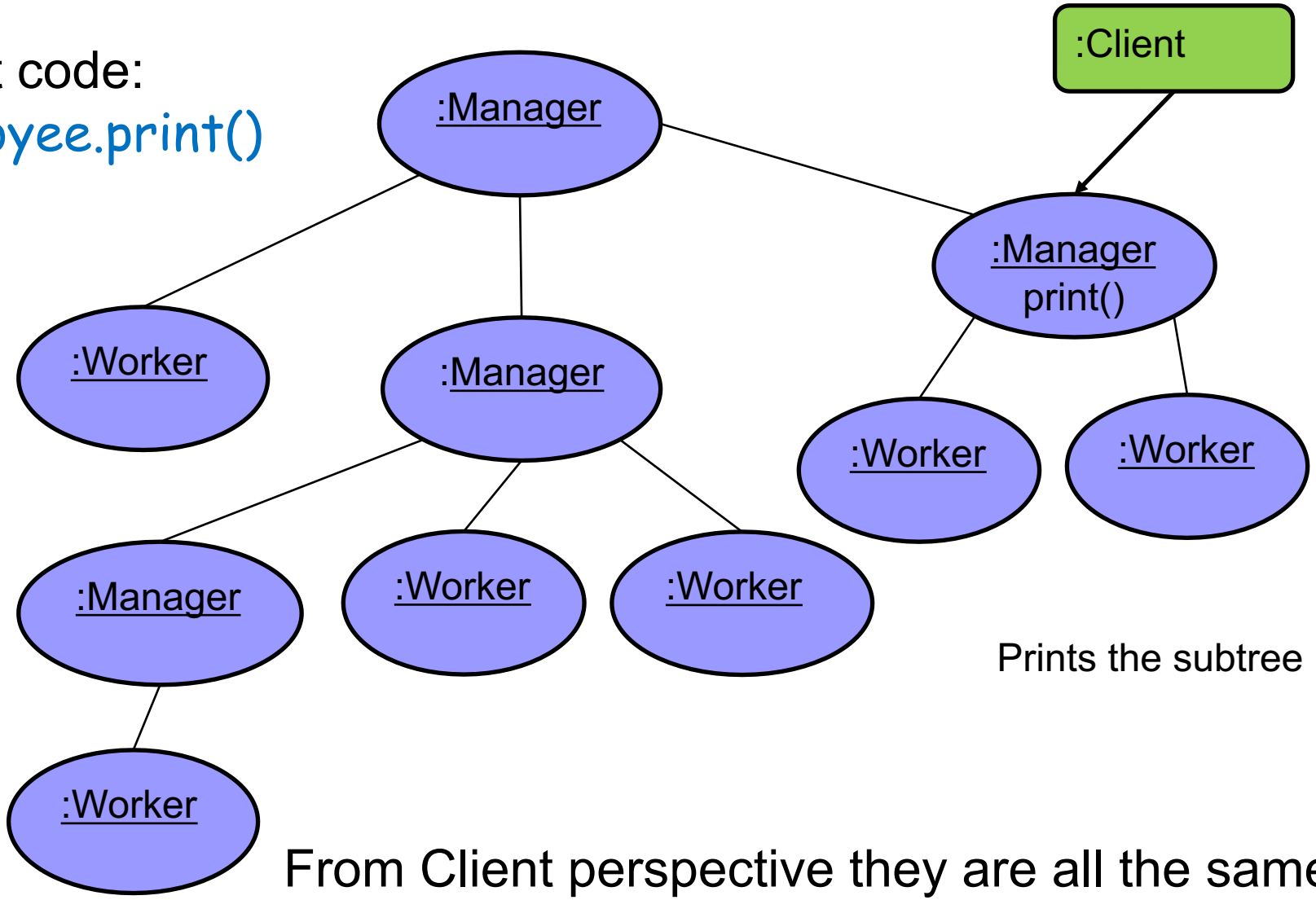
employee.print()



Prints the information of the whole tree

Composite company

Client code:
employee.print()



Prints the subtree

From Client perspective they are all the same

Sample code

```
abstract class Employee{
   abstract void print();
   void add(Component) {
        //throw exception};
   void remove(Component) {..};
   Component get(int){….};
}
class Worker extends Employee{
   private int data;
   public void print(){
        System.out.println(value);}
   public Worker(int v){ data=v;}
}
```

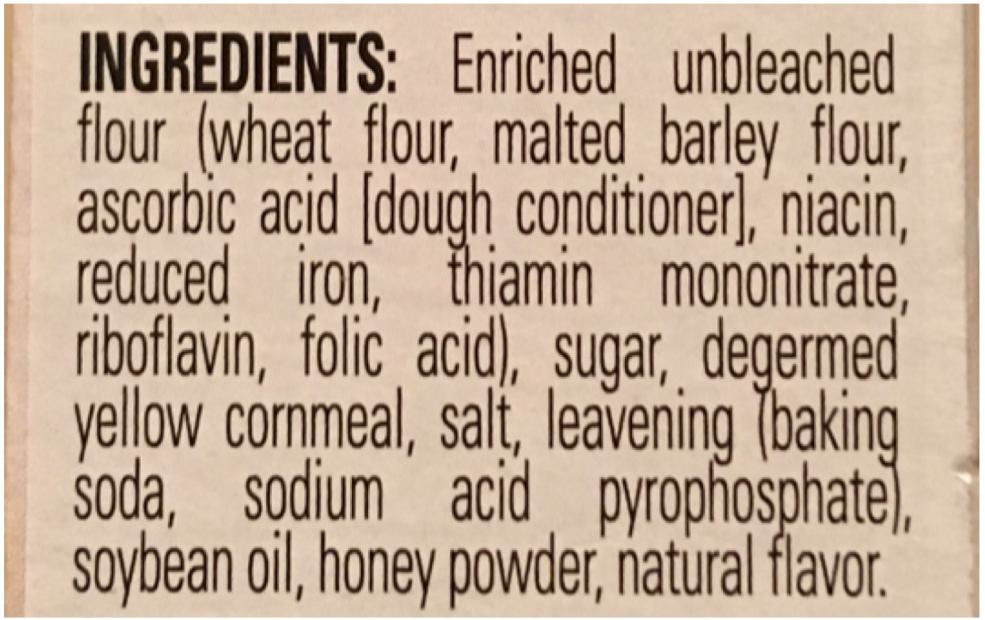
```
class Manager extends Employee{
   List<Employee> children;
   void add (Employee  c){
        children.add(c);}
   void remove(Employee c){
         children.remove(c);}
   Employee get(int i){
        return children.get(i);}
   void print(){
        //…print own data first, then
        for(Component c: children)
              c.print();
   }
  public Manager(int v) {data=v;/*..*/};
  private int data;
}
```

Sample client code

```
Employee example=new Manager(10);
example.add(new Worker(3));
example.add( new Manager(5).add(new Worker(2));
Employee another=new Manager();
…..
example.add(another);
….
example.print(); //print the whole structure

example.get(1).print(); //prints the whole subtree
```

A recurring problem…

- Any organizational hierarchy
- Organizing files and folders
  - Is it different to move a file from move a folder?
- Even printing ingredients☺
- Graphical User Interface frameworks



Example 2: Scenario

- A GUI system has window objects which can contain various GUI components (widgets) such as, buttons and text areas.
- A window can also contain widget container objects which can hold other widgets.
- We want to update the window and all its content for some user interactions
- Problem:
  - How to represent these widgets?
  - Update one widget or whole window
    - E.g. change the theme

Example2: Solution 1

```
public class Window {
   Button[] buttons;
   Menu[] menus;
   TextArea[] textAreas;
   WidgetContainer[] containers;
   public void update() {
      if (buttons != null)
        for (int k = 0; k < buttons.length;
              k++) buttons[k].draw();
      if (menus != null)
         for (int k = 0; k < menus.length;
              k++) menus[k].refresh();
      // Other widgets handled similarly.
     if (containers != null)
       for (int k = 0; k<containers.length;
      k++)containers[k].updateWidgets();
}
...
```

- Solution 1: Design all the widgets with different interfaces for "updating" the screen.
  - a Window update() method
- A bad solution
- violates the Open-Closed Principle.
  - If we want to add a new kind of widget, we have tomodify the update() method of Window to handle it.

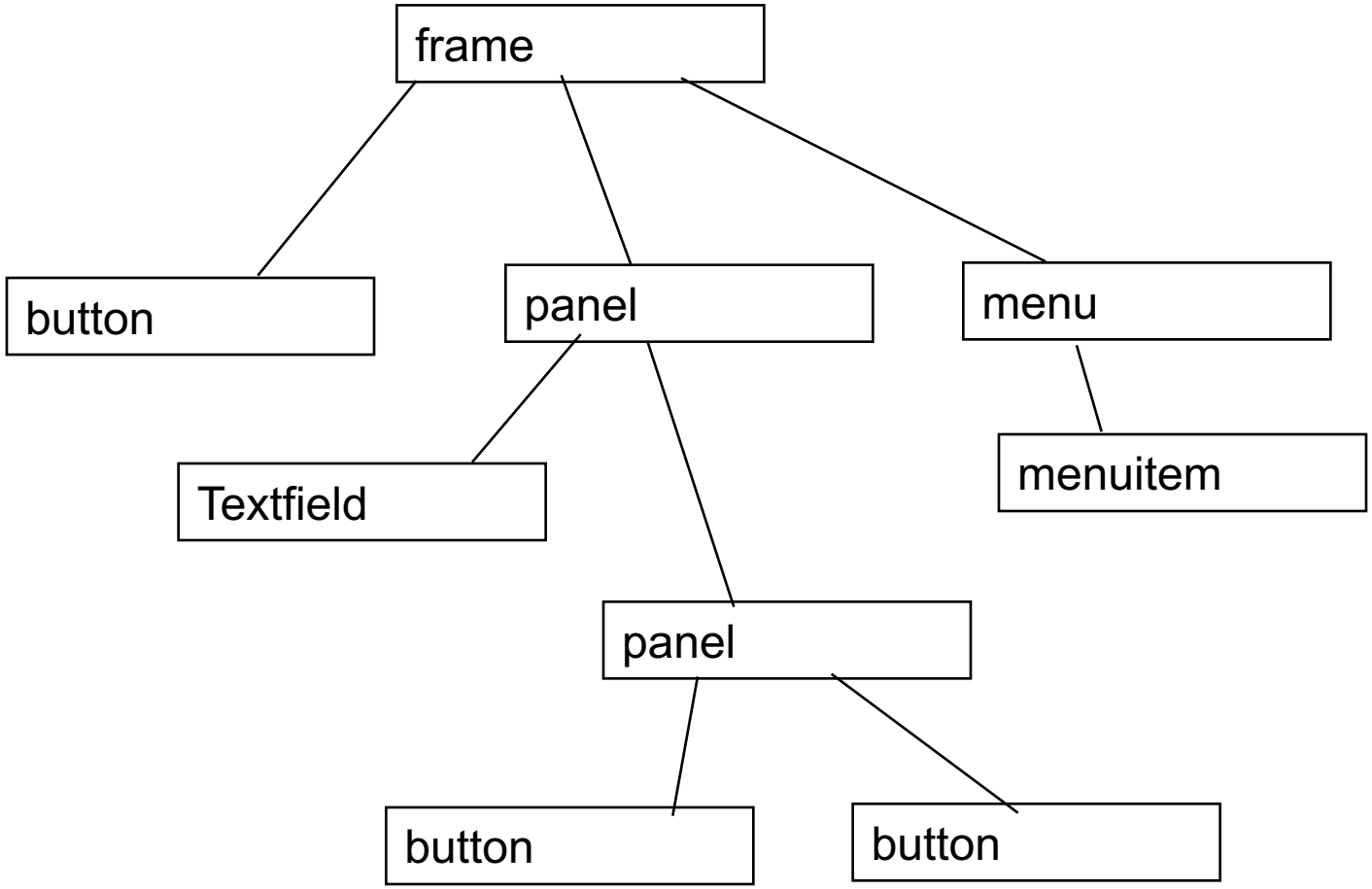
Example2: Solution 2

- Try to program to an interface. All widgets support the Widget interface,
  - subclasses of a Widget class

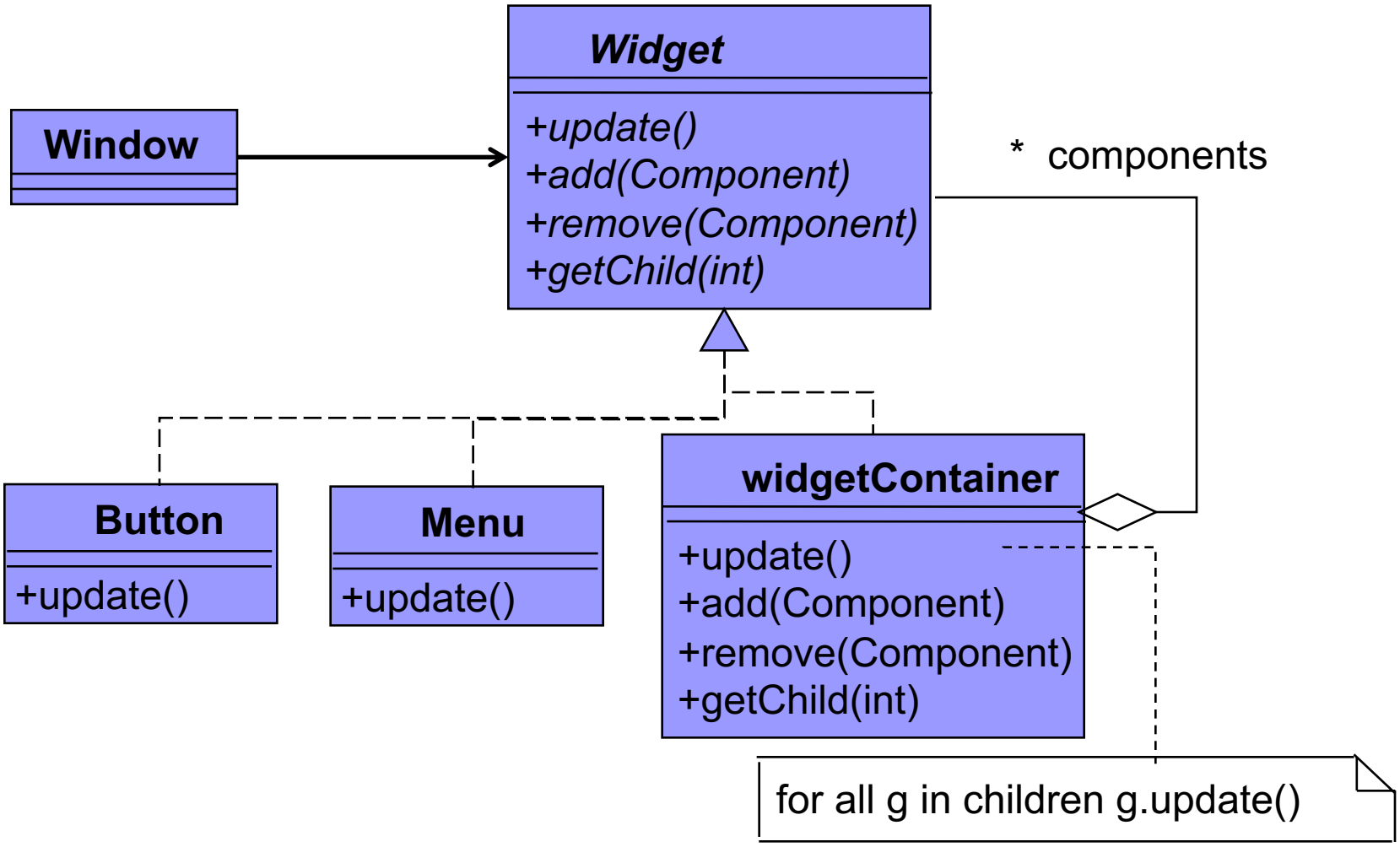
```
public class Window {
   Widget[] widgets;
   WidgetContainer[] containers;
   public void update() {
     if (widgets != null)
       for (int k = 0; k < widgets.length; k++) widgets[k].update();
     if (containers != null)
       for (int k = 0; k < containers.length; k++ )
            containers[k].updateWidgets();
}}
```

still distinguishing between widgets and widget containers

The tree of widgets and containers



Applying the Composite



Example 2 –Composite pattern soln

```
abstract class Widget{
   abstract void update();
   void add(Component) {
         //throw exception};
   void remove(Component) {..};
   Component get(int){….};
}
class Button extends Widget{
   public void update(){
         //updating this button
   }
   /*…other methods*/
}
```

```
class WidgetContainer extends Widget{
   List<Widget> children;
   void add (Widget  c){
         children.add(c);}
   void remove(Widget c){
          children.remove(c);}
   Widget get(int i){
         return children.get(i);}
   void update(){
         for(Widget c: children)
               c.update();
   }
  /*…other operations and fields*/
}
```

Sample client code

```
public class Window{
   private Widget root;
   void initialize(){….}
   /*….*/
  public void update() {
    root.update(); // updates itself and all its children
  }
}
```

Example 3: Navigation menu

- How a navigation menu looks (a sitemap)

Root
——Folder1
————————-File11
————————-File12
——Folder2
————————-File21
————————-Folder21
————————————File211
————————-Folder22
————————-Folder23
————————————File231
——Folder3

- We only want to show those links to user which he is authorized to view.
- if all the links in a category are not visible then the category itself will not be visible.
- Operations: isAuthorized() and display()

Draw a class diagram…

http://www.zapbuild.com/bitsntricks/composite-design-pattern/

Component responsibilities

1. Its own responsibility
   - i.e. leaf operations
   - e.g. employee methods
2. Child management
   - add(), remove(), getChild()

- Trading *single responsibility* for uniform access

Implementation issues-1

- Where should the child management methods (add(), remove(), getChild()) be declared?
  - In the Component class: Gives transparency, since all components can be treated the same. But it's **not safe**, since clients can try to do meaningless things to (add, remove) leaf components at run-time.
    - Add/remove fail by default, the composite overrides them
    - Add/remove does nothing, the composite overrides them
  - In the Composite class: Gives safety, since any attempt to perform a child operation on a leaf component will be caught at compile-time. But we **lose transparency**, since now leaf and composite components have different interfaces.

Implementation issues-1

- In the Component class
  - Interpret the leaves as components with 0 child.
  - If we view a Leaf as a Component that *never* has children, then we can define a default operation for child access in the Component class that never returns any children.
  - Leaf classes can use the default implementation,
    - Throw exception by default
    - Do nothing by default
    - Return null or false
  - Composite classes will reimplement it to return their children, add or remove children

Implementation issues-1

- In the Composite class
  - Leaf and composite components have different interfaces
  - Requires the client to check the type of every object before making a call so the object can be cast correctly.
    - Uniformity is traded for safety

Implementation issues-2

- Should components maintain a reference to their parent component?
  - Depends on application
  - If needed, put the parent reference in the *Component* class
    - Add and Remove operations handle the parent reference
  - Suppose we have a reference to a child and need to delete it.
    - We must get the parent to remove the child.
    - Having the parent reference makes that easier.
  - Having parent references supports the Chain of Responsibility pattern (later)

Composite Pattern with Parent: C++

```
class Component{
 public:
  virtual ~Component() {}
  virtual void add(Component *component) {}
  virtual void remove(Component *component) {}
  virtual Component *get(int i) const {return 0;}
  void setParent(Component *parent) {
      this->parent_ = parent;  }
  Component *getParent() const {
      return this->parent_;
  }
 virtual std::string operation() const = 0;
 protected:  Component *parent_;
};
```

```
class Leaf : public Component {
 public:
  string operation() const override {
     return "Leaf";  }
};
```

Composite Pattern with Parent: C++

```
class Composite : public Component {
protected:   std::vector<Component *> children_;
public:
  void add(Component *component) override {
      this->children_.push_back(component);
      component->setParent(this);}
void remove(Component *component) override {
    children_.remove(component);
    component->setParent(0);   } //assuming no sharing
std::string operation() const override {
    std::string result;
    for (const Component *c : children_) {
       result += c->operation() + " ";      }
     return result;}
  };
```

Implementation issues-3

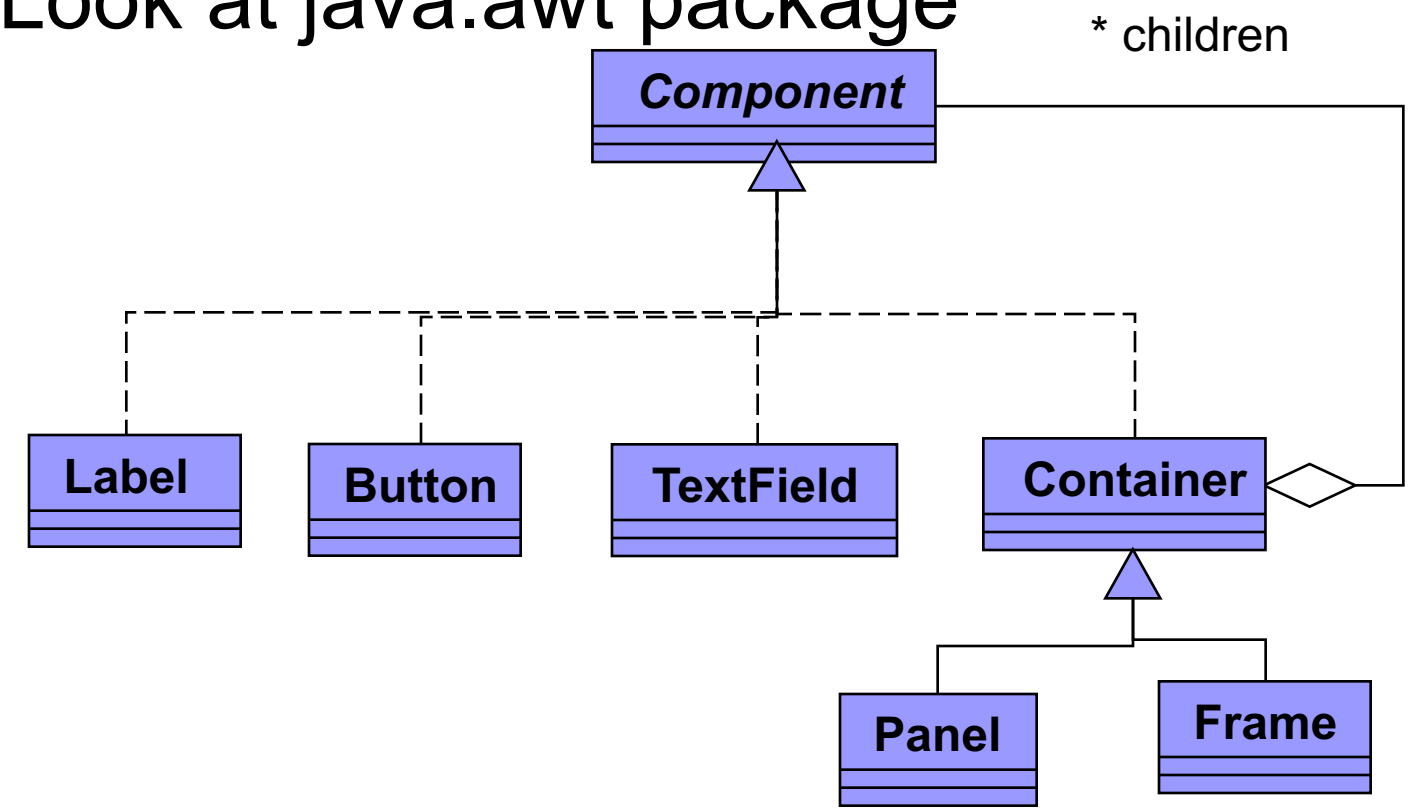
- Should Component maintain the list of components that will be used by a composite object?
  - Should children list be an instance variable of Component rather than Composite class?
  - Better to keep this part of Composite and avoid wasting the space in every Leaf object
- Is child ordering important?
  - Depends on application:
    - parse tree, graphics front-to-back order

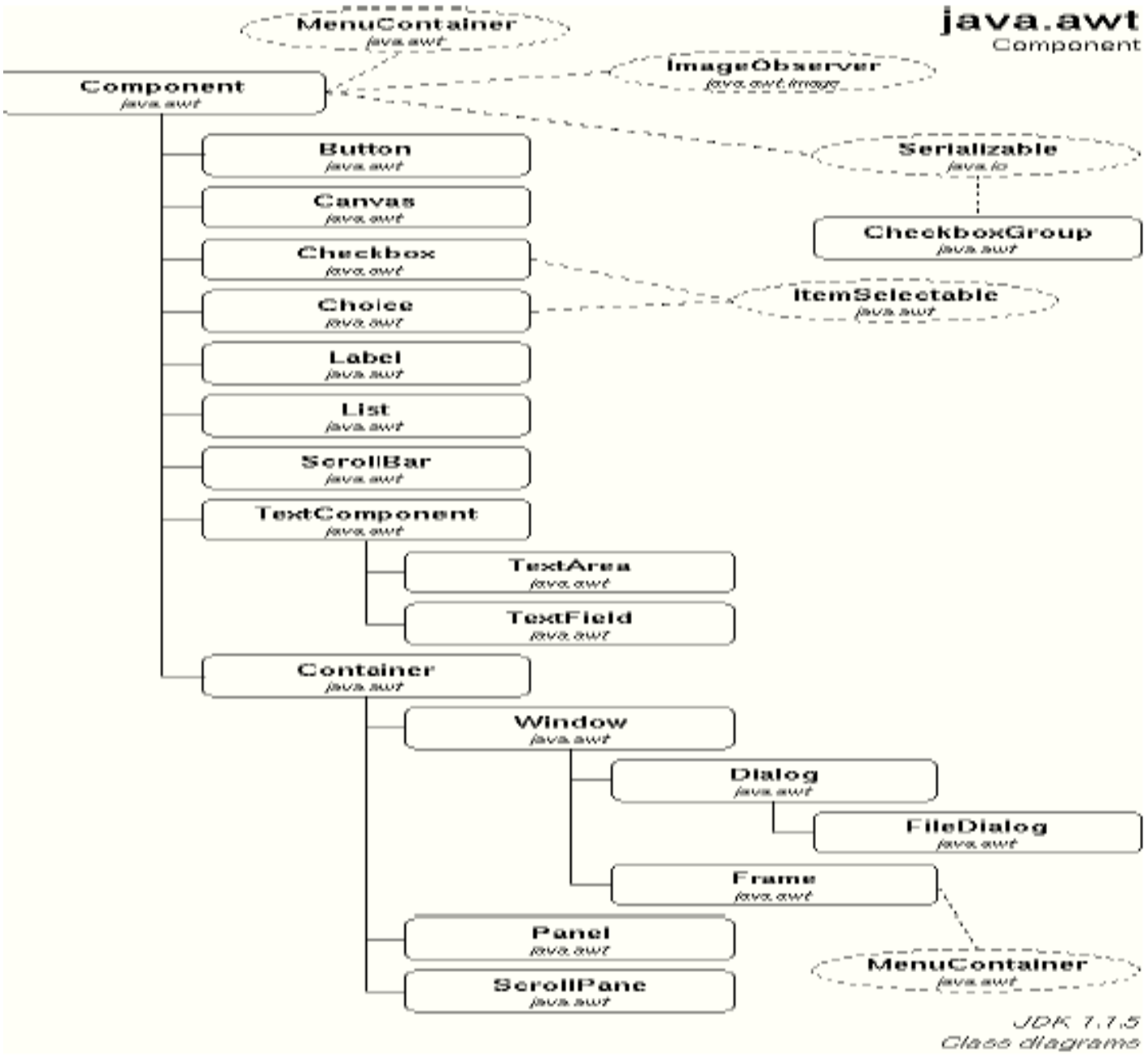
Implementation issues-4

- Sharing components to remove redundancy
  - When components have reference to their parents, it gets tricky
- Who should delete a component?
  - If there is garbage collection, like Java, this is not an issue
  - Composite deletes its children when destroyed
    - Destructor of Composite calls destruct for all children
  - Careful when components are shared by multiple composites

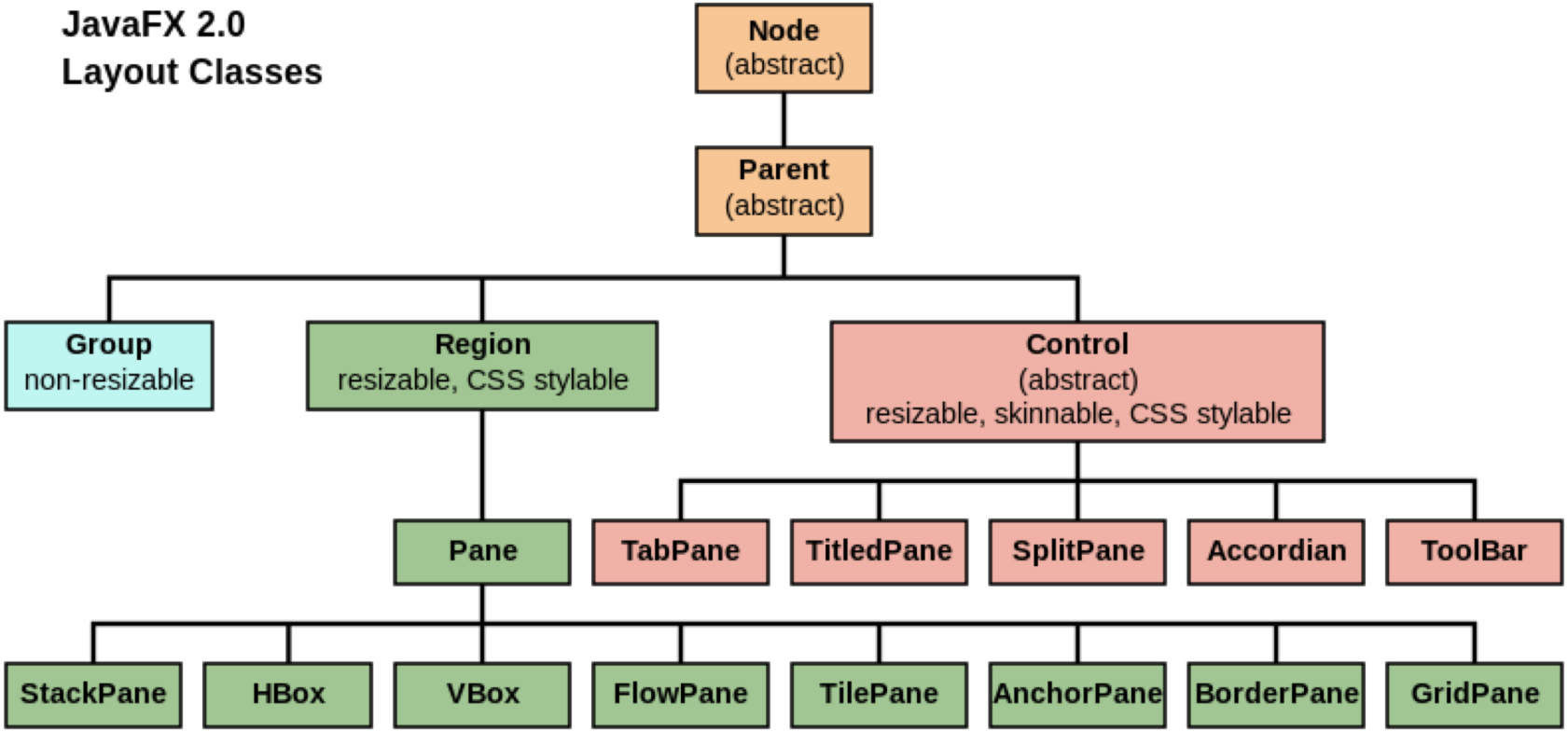
Composite –Known uses

- GUI frameworks
- Web page layouts
- Look at java.awt package




java.awt
Component
MenuContainer
java.awt
Component
java.awt
ImageObserver
java.awt.image
Serializable
java.io
Button
java.awt
Canvas
java.awt
Checkbox
java.awt
CheckboxGroup
java.awt
Choice
java.awt
ItemSelectable
java.awt
Label
java.awt
List
java.awt
ScrollBar
java.awt
TextComponent
java.awt
TextArea
java.awt
TextField
java.awt
Container
java.awt
Window
java.awt
Dialog
java.awt
FileDialog
java.awt
Frame
java.awt
Panel
java.awt
ScrollPane
java.awt
MenuContainer
java.awt
JDK 1.1.5
Class diagrams

JavaFX 2.0
Layout Classes



Composite - Consequences

- Defines class hierarchies consisting of primitive and composite objects
- Clients can treat composite structures and individual objects uniformly
  - Simpler clients, no case-based handling
  - Client can make one method call and execute an operation over an entire structure.
- Easier to **add** new kinds of components
- Makes it **harder** to restrict the components of a composite.You have to check the constraints on runtime

DECORATOR

Motivation

- Sometimes we want to add responsibilities to individual objects, not to an entire class
  - E.g. I want to add scrollbar to a TextArea
- We can do this with inheritance, but it limits our flexibility.
  - How to add and withdraw scrollbar whenever necessary?
  - Change the type of the object at runtime?
  - Later, I want to add border to some TextArea objects
  - Too many subclasses!
- A better way is to use object composition!

Question

- What if I put flags for each add-on
  - isScroolbar, isBorder,…..
  - Loosing advantage of object orientation
  - Violates which principle?
- No more class explosion
- I can withdraw scrollbar by setting the flag to false
- Not a good idea

Object composition

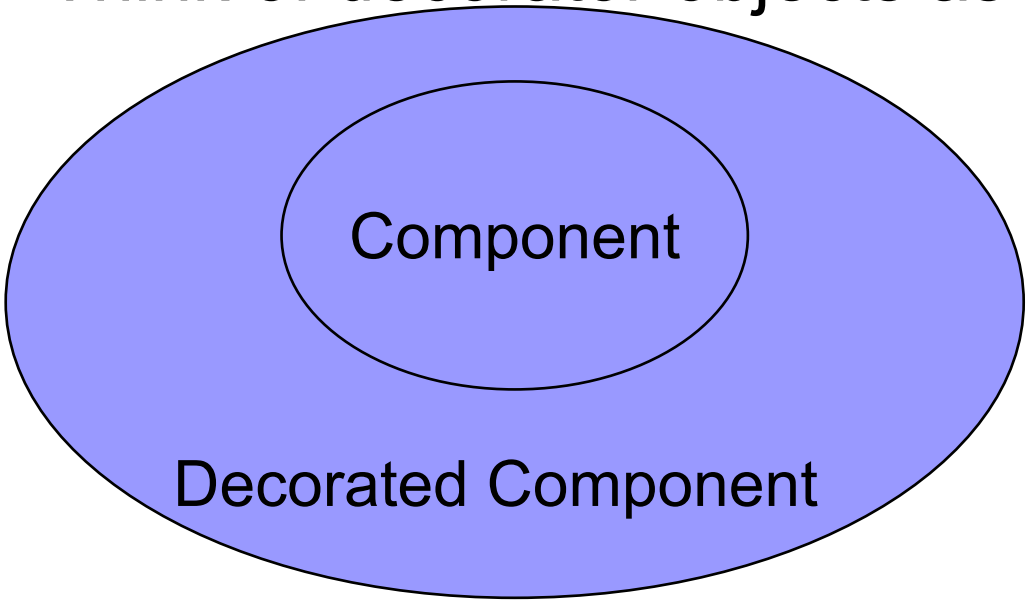
- TextArea and Border. Who composes whom?
  - TextArea contains Border
    - Must modify subclasses, if any, to make it aware of the border.
  - Border contains TextArea
    - keeps the border-drawing code entirely in the Border class, leaving other classes alone.
- What does the Border class look like?
  - Clients should not care whether TextArea objects have borders or not.
  - Border and TextArea have the same interface
- Border is decorating the TextArea

Decorator

- Intent
  - Attach additional responsibilities to an object dynamically
  - a.k.a. Wrapper
- You'll be able to give objects new responsibilities without changing the client code

Decorator is a Wrapper

Think of decorator objects as “wrappers.”

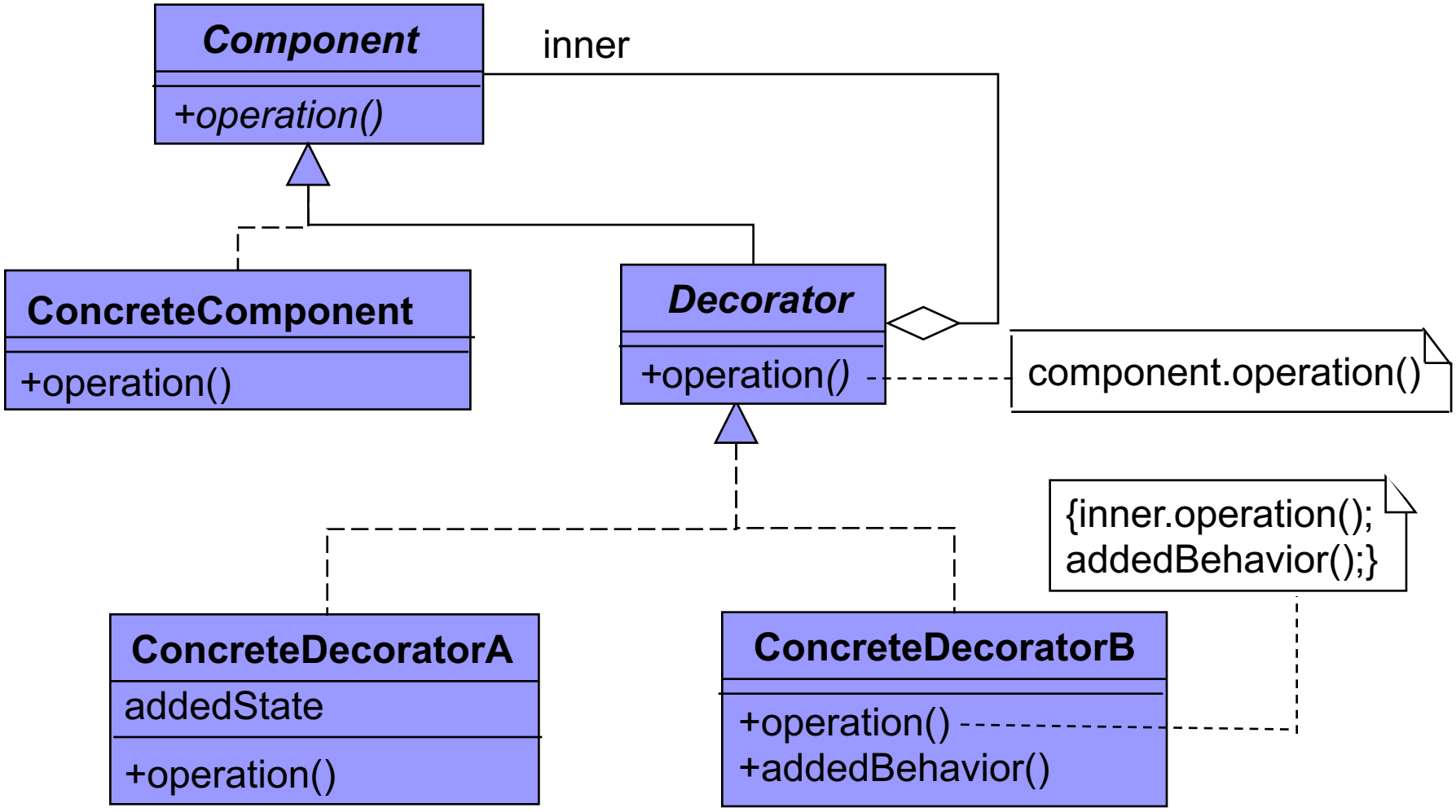


Wrapping additional responsibility

Wrapping with composition

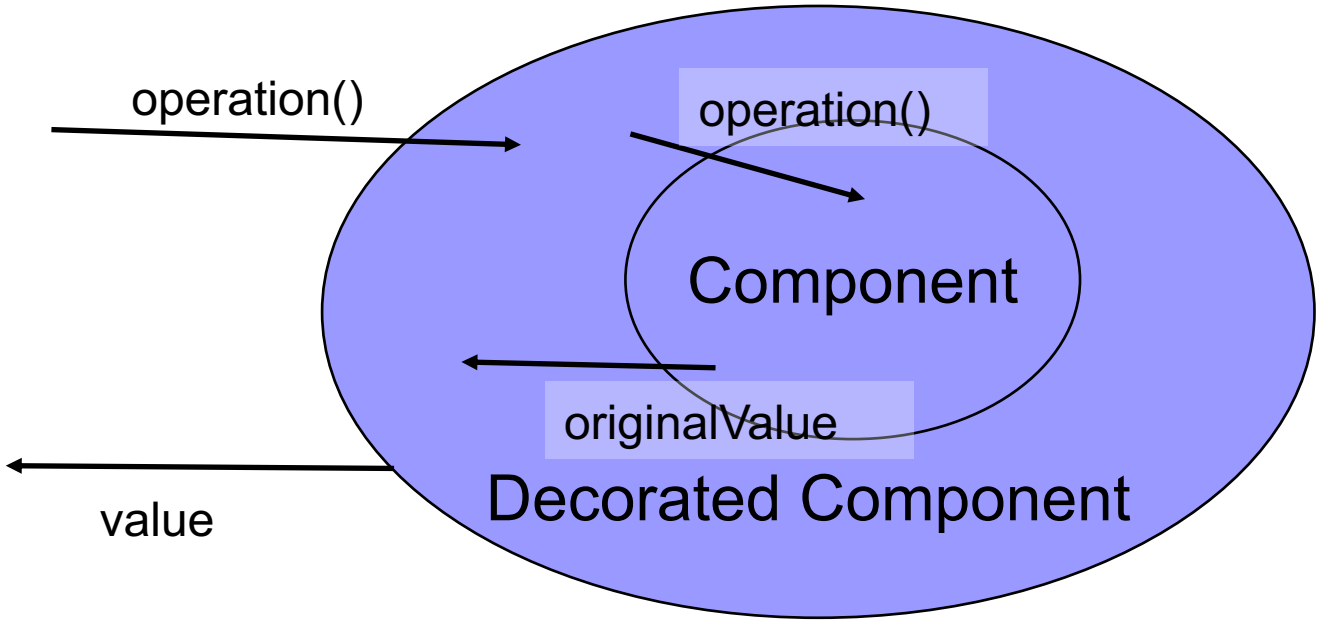
Decorator -Structure

Participants?



Key points

- Decorator forwards requests to its Component object.
- A concrete decorator executes its behavior **before or after** the call to the parent method
  - which always delegates to the wrapped object



Decorator is a **Wrapper**

Key points

- Decorator forwards requests to its Component object.
- A concrete decorator executes its behavior **before or after** the call to the parent method
  - which always delegates to the wrapped object

- We can pass around a decorated object in place of the original (wrapped) object.
  - Decorators have the same supertype as the objects they decorate.
- Objects can be decorated at any time
  - We can decorate objects dynamically at runtime with as **many** decorators as we like

Key points

- We can use one or more decorators to wrap an object.

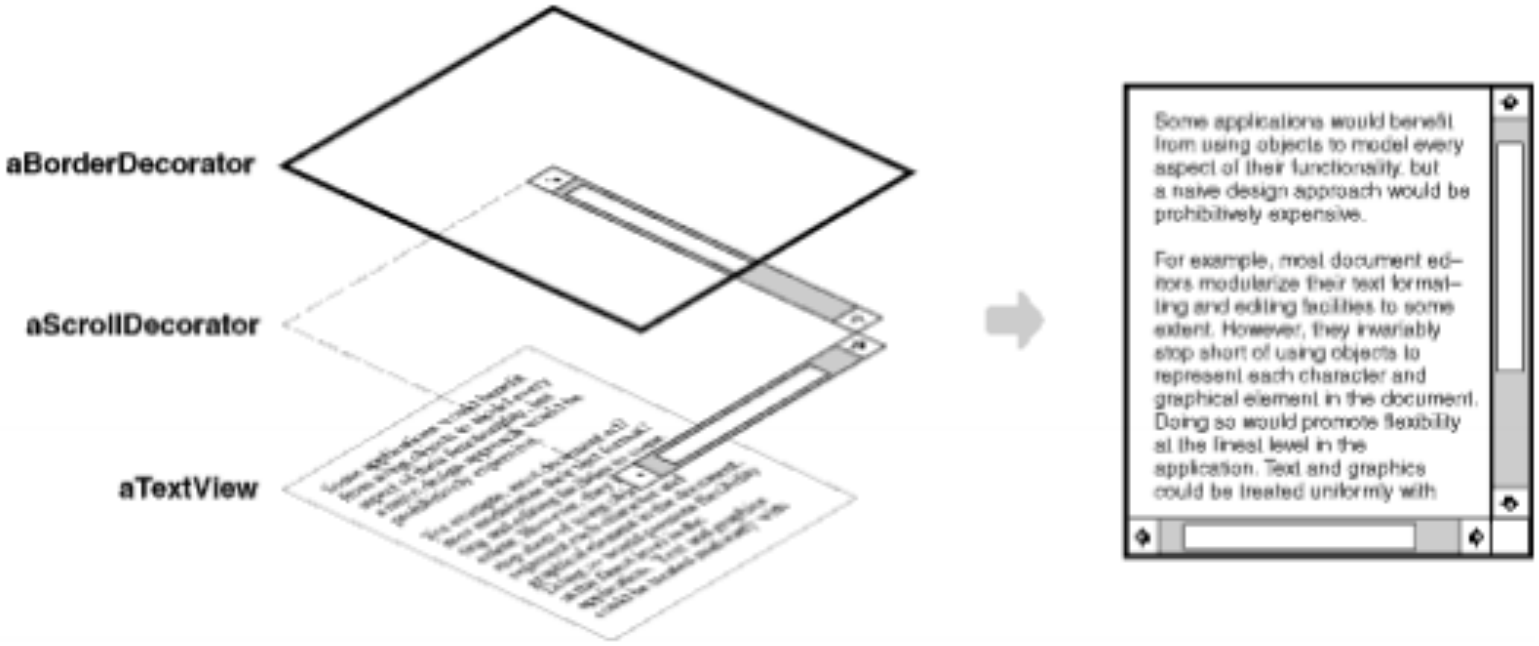
- Chaining decorators is like layering logic
  - We can structure the business logic into layers,
  - create a decorator for each layer and compose objects with various combinations of this logic at runtime.
  - The client code can treat all these objects in the same way, since they all follow a common interface.

Decorator

- Intent
  - Attach additional responsibilities to an object dynamically.
- Applicability
  - Need to add responsibility to individual objects ***dynamically*** and without affecting others
  - Need to add responsibilities ***that can be withdrawn***
  - Allow behavior to extend **without modifying or** breaking the code that uses these objects.
  - When extension by subclassing is impractical
    - Class definition may be hidden
    - Large number of independent extension will result subclass explosion

Example 2: Decorating with borders and scrollbars

- Scrollbar will be added whenever necessary



Draw a class diagram

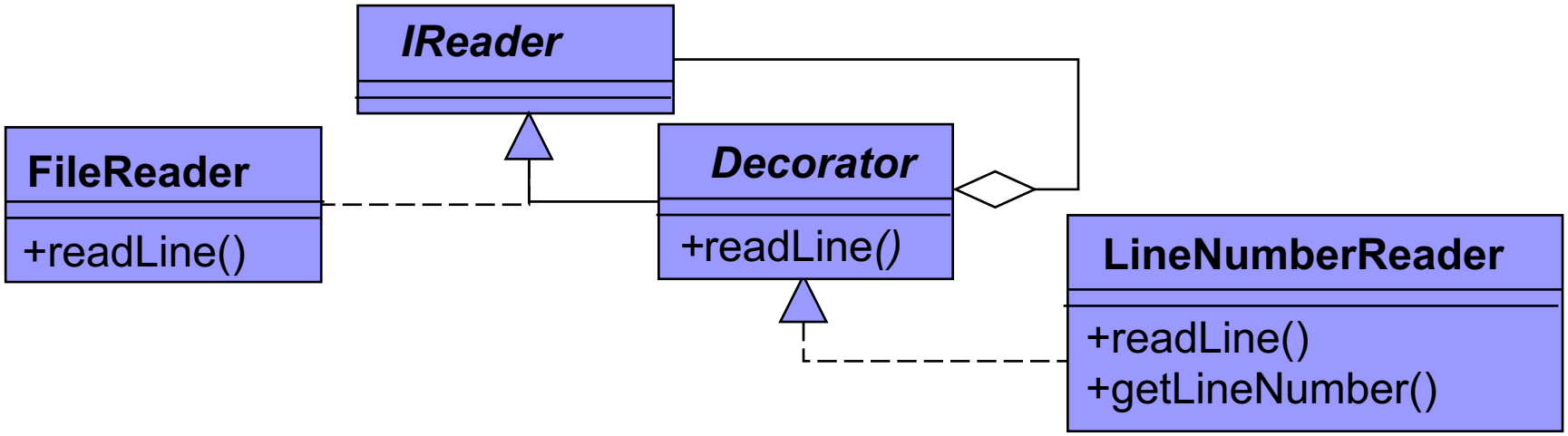
```
class Widget{
  public: virtual void draw() = 0;
};
class TextView: public Widget{
    int width, height;
  public:
    TextView(int w, int h)
            :width(w),height(h){}
    virtual void draw(){/*actual draw*/   }
};
class Decorator: public Widget {
    Widget *wid;
  public:
    Decorator(Widget *w): wid(w){}
    virtual void draw(){
       wid->draw(); //  Delegation
    }
};
```

```
class BorderDecorator: public Decorator{
  public:
    BorderDecorator(Widget *w): Decorator(w){}
    void draw(){
       Decorator::draw();
       cout << “after--BorderDecorator" << '\n';
    }
};

class ScrollDecorator: public Decorator{
  public:
   ScrollDecorator(Widget *w): Decorator(w){}
   void draw(){
      cout << " before-- ScrollDecorator" << '\n';
      Decorator::draw();
      }
};
```

Write a client that builds a double bordered textview with a scrollbar.

Example 3:

- We have a FileReader implementing a Reader interface
- but sometimes we want to read the number of lines in a file
- Decorate FileReader with LineNumberReader



Example 3: LineNumberReader

```
abstract class Decorator implements Reader{
   private Reader innerObject;
   public Decorator(Reader r){innerObject=r;}
   public String  readline(){ innerObject.readLine();}
}
class LineNumberReader extends Decorator{
   int counter;
   public LineNumberReader(Reader inner){ super(inner);}
   public String  readline(){ super.readLine(); counter++;}
   public int getLineNumber(){return ccounter;}
}
```

- Client code

```
FileReader fr=new FileReader(filename);
LineNumberReader lrdr = new LineNumberReader(fr);
```

Implementation issues

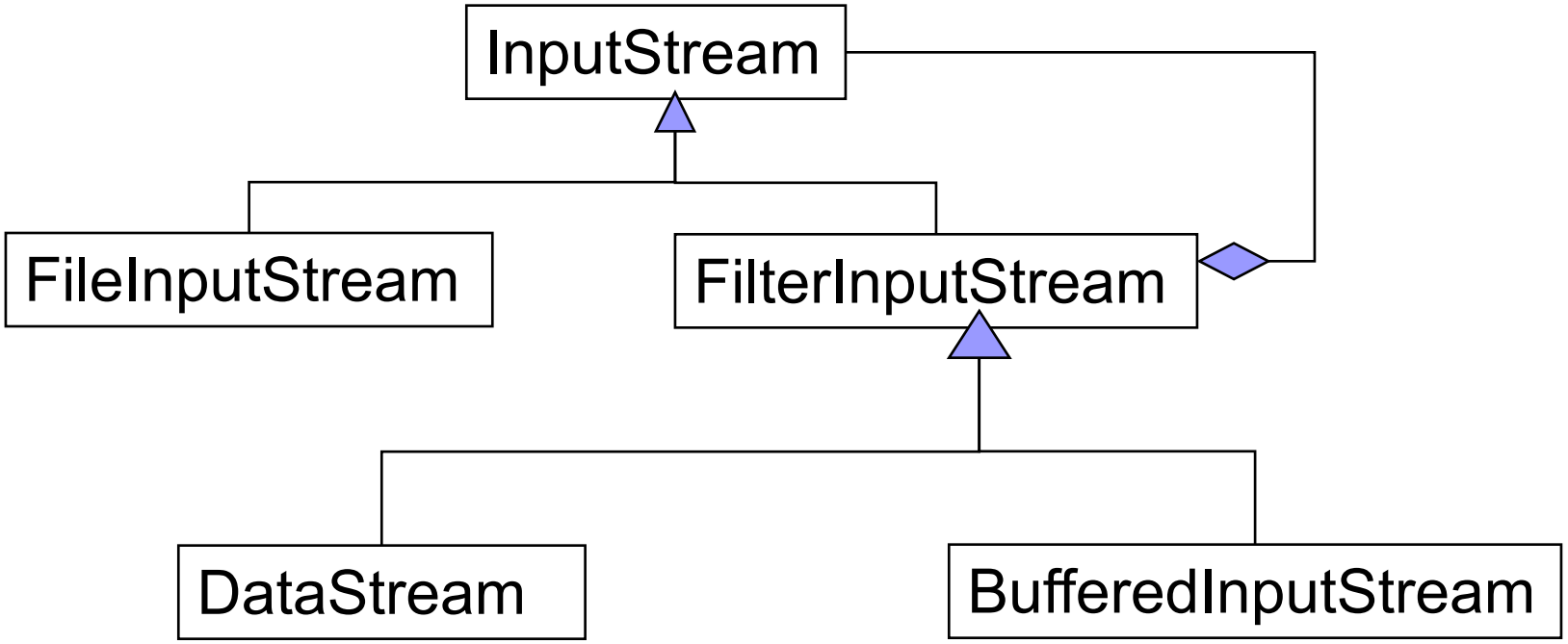
- Keep Component lightweight
  - Java interface, C++ pure virtual
  - concrete class should not pay for features they do not need
  - Intrinsically heavyweight Component makes the Decorator pattern too costly to apply.
- Adding only one responsibility, abstract decorator may be overkill
  - Forwarding the request is in the concrete one
- Decoration order do matter

Known Uses

- X Window use Decorators to add a title bar, border, and scroll bars to a window
- The java.io package
  - FileInputStream is wrapped with BufferedInputStream
  - Further wrapped with LineNumberInputStream
  - Both wrappers extend FilterInputStream (abstract decorator)

Know use

- java.io.InputStream structure



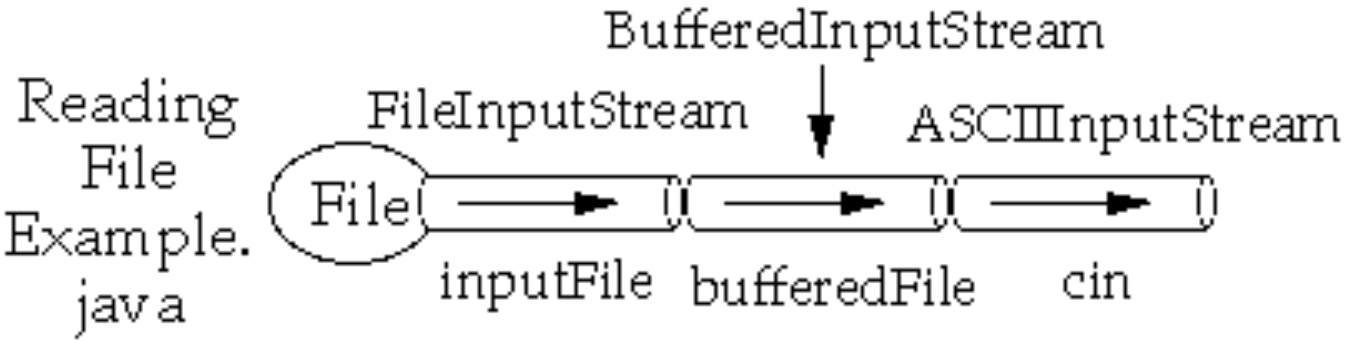
Known use: java.io package

- The basic I/O classes are InputStream, OutputStream, Reader and Writer
- Decorations:
  - Buffered Stream - adds buffering for the stream
  - Data Stream - allows I/O of primitive Java data types
  - Pushback Stream - allows undo operation

Example decoration chaining

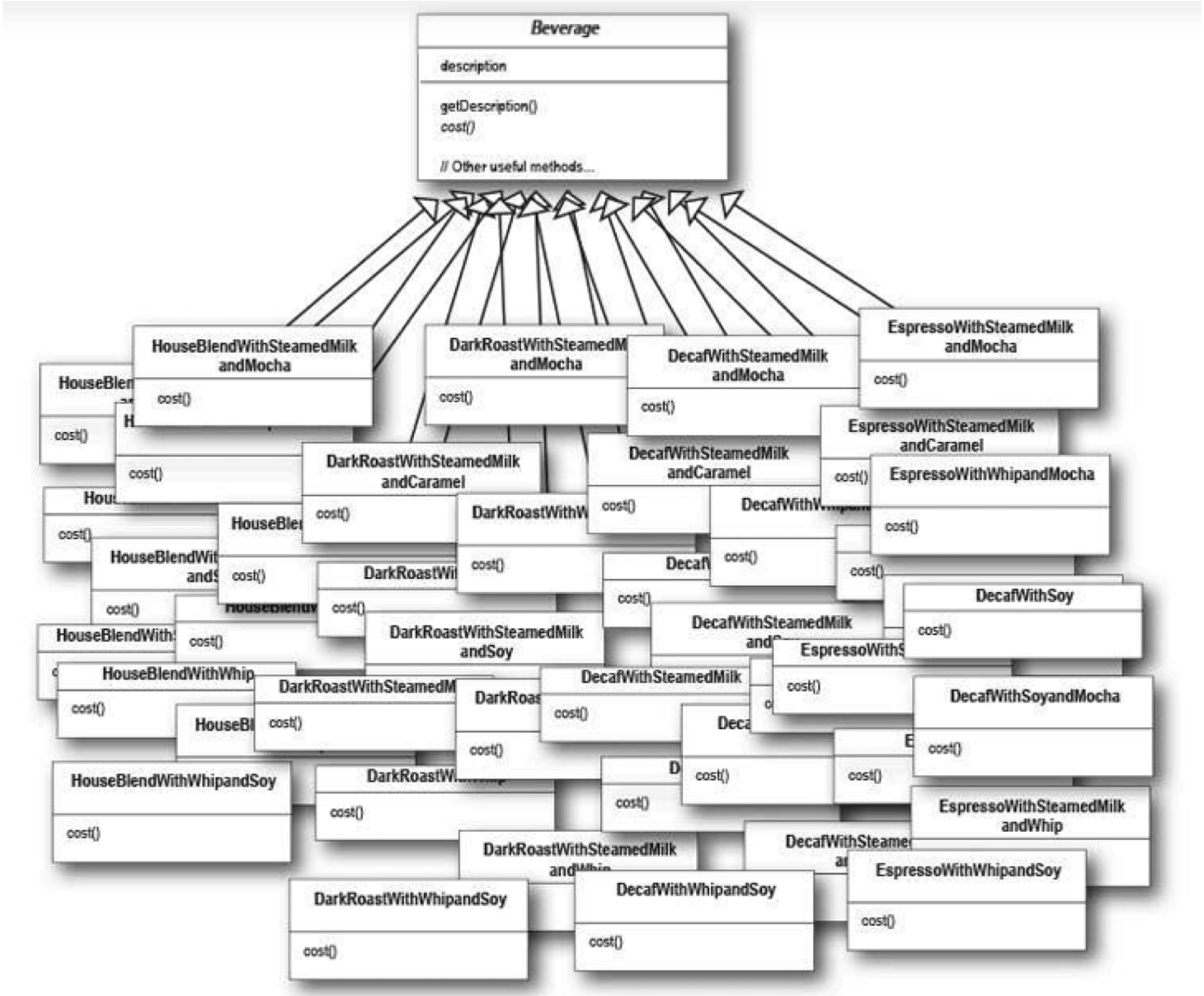
```
public class JavaIO {
 public static void main(String[] args) {
    // Open an InputStream.
    FileInputStream in = new FileInputStream("test.dat");
   // Create a buffered InputStream.
     BufferedInputStream bin = new BufferedInputStream(in);
    // Create a buffered, data InputStream.
    DataInputStream dbin = new DataInputStream(bin);
    // Create an unbuffered, data InputStream.
    DataInputStream din = new DataInputStream(in);
    // Create a buffered, pushback, data InputStream.
     PushbackInputStream pbdbin = new PushbackInputStream(dbin);


   }
 }
```

```
class  ReadingFileExample     {
  public  static  void  main( String  args[]  ) throws Exception
{

      ASCIIInputStream  cin =new ASCIIInputStream(
            new BufferedInputStream (
              new FileInputStream( "ReadingFileExample.java" );

       System.out.println(  cin.readWord()  );
       for  ( int  k = 1 ; k< 4;k++ ) System.out.println(
cin.readLine()  );
    }
 }
```



Decorator – Consequences: Pro

- Flexible **alternative** to **inheritance** for extending functionality
  - Adding/removing responsibility at runtime
  - Mix and match responsibilities
  - Decorators are simple and stackable
  - Avoid class explosion due to inheritance



Source: Head First Design Patterns. Reading this chapter is advised

Decorator – Consequences: Pro

- Flexible alternative to inheritance for extending functionality
  - Adding/removing responsibility at runtime
  - Mix and match responsibilities
  - Decorators are simple and stackable
- Avoids complex classes high up in the hierarchy
  - Keep base simple, Add functionality incrementally
    - a client need not pay for features it does not use.
  - Easy to extend, no need to crowd the base
    - Extending a complex class tends to expose details unrelated to the responsibilities being adding.
- Enables Single Responsibility
  - divide a monolithic implementing many possible variants of behavior into several smaller ones.

Decorator – Consequences: Cons

- Do not rely on identity check
  - They have the same interface, but decorated object is not identical to the object itself.
- Lots of little objects that all look alike
  - Differ only the way they are connected
  - Hard to understand the design and hard to debug
  - Be cautious!
- Initial configuration is ugly ☺
  - See Java I/O

Decorator-Related Patterns

- **Adapter** *modifies* an objects interface.
  - Decorator only alters an objects behaviors while maintaining the interface.
- **Composite:** Decorator looks like a degenerate composite, but the intention is different
  - Object aggregation vs adding responsibilit
- **Proxy**: wrapping objects for different Intent
- Strategy : Change the skin vs change its guts

(later)