Object Orientation with Design Patterns

Adapter Pattern

Composite Pattern

Facade Pattern

Adapter

- **Intent:**
  Convert the **interface of a class** into **another interface that client expects**.
  Adapter lets classes work together that couldn't otherwise because of **incompatible interfaces.**

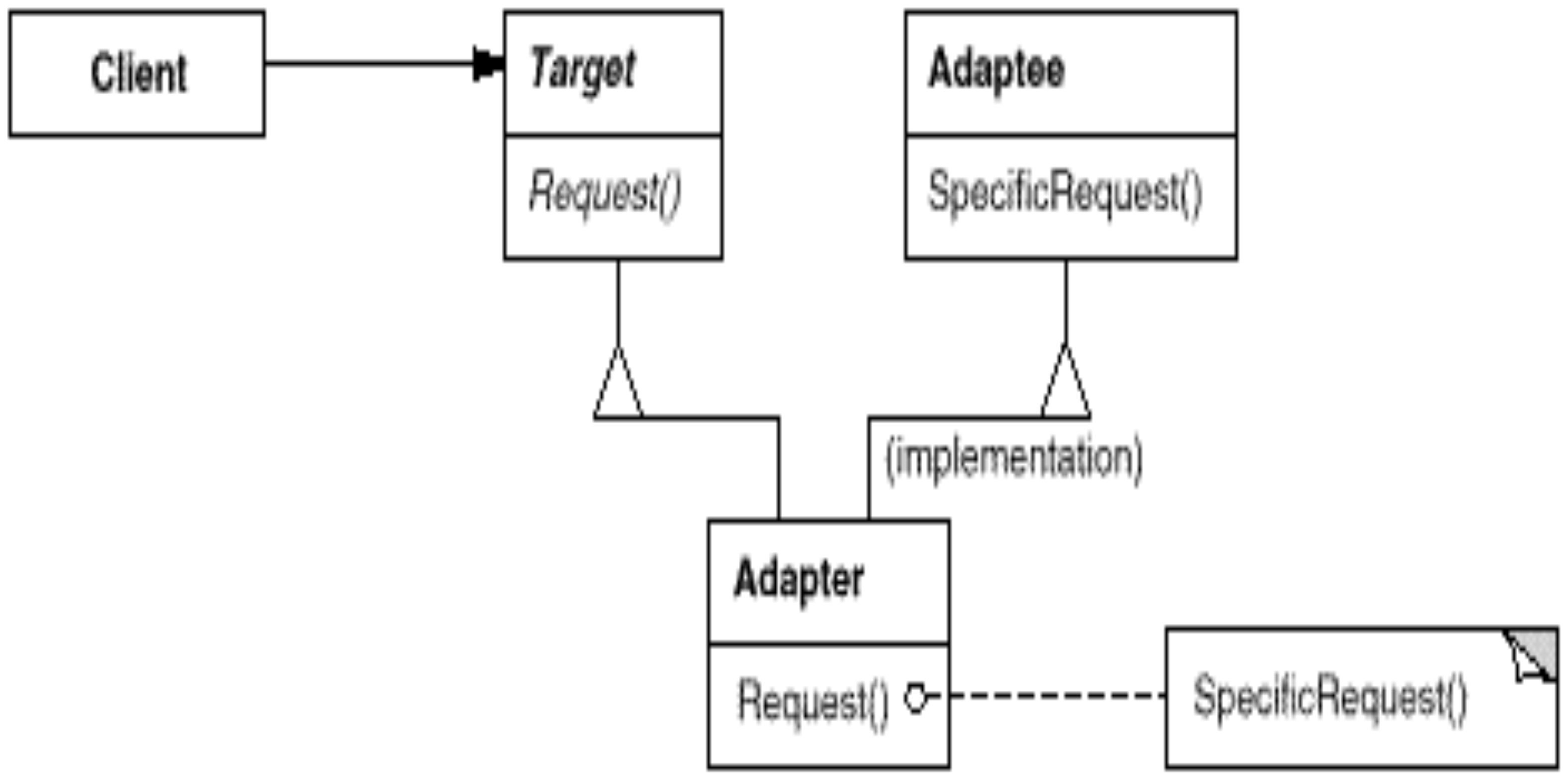
- Basically, this is saying that we need a way to create a **new interface** for **an object that does the right stuff** but has the **wrong interface.**

Adapter

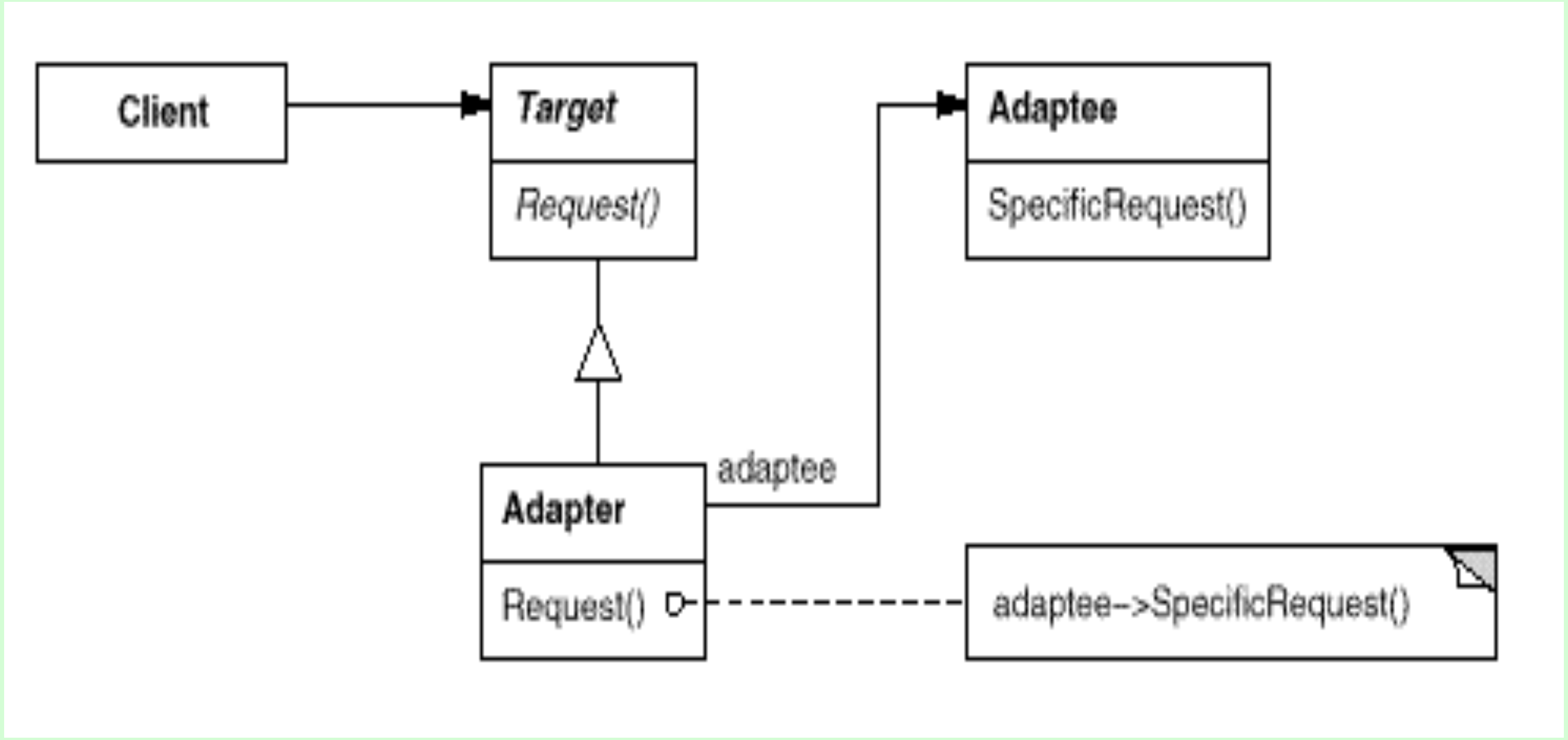
- There are actually **two types** of adapter pattern:
  - **Object Adapter Pattern**
    The circle example is an example of this.
    It relies on one object (the adapting object) containing an other (the adapted object)

  - **Class Adapter Pattern**
    Another way to implement the Adapter pattern is with **multiple inheritance**. In this case, it is called a Class Adapter pattern.

Adapter Pattern - Class



Adapter Pattern - Object



Adapter Pattern Participants

- **Target** (Shape)

  Defines the domain-specific interface that Client uses.

- **Client** (DrawingEditor)

  Collaborates with objects conforming to the Target interface.

- **Adaptee** (XXCircle)

  Defines an existing interface that needs adapting.

- **Adapter** (Circle)

  Adapts the interface of Adaptee to the Target interface.

Adapter Pattern Applicability

Use Adaptor when:

- you want to **use an existing class**, and **its interface** does **not match the one you need.**
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, **classes that don't necessarily have compatible interfaces.**
- *(object adapter only)* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Learning the Adapter Pattern

- Let’s look at an example of where it is useful.
- Let’s say I’ve been given the following **requirements:**
  - Create **classes for points, lines and squares** that have the behaviour ‘display’.
  - The client objects should not have to **know whether they actually have a point, line or square**. The just want to know that they have one of these shapes.
- In other words, I want to include these shapes in a **higher-level concept** that I will call a ‘displayable shape’.

Learning the Adapter Pattern

- To accomplish this, I will create a Shape class and then derive from it the classes that represent points, lines and squares (see fig below)

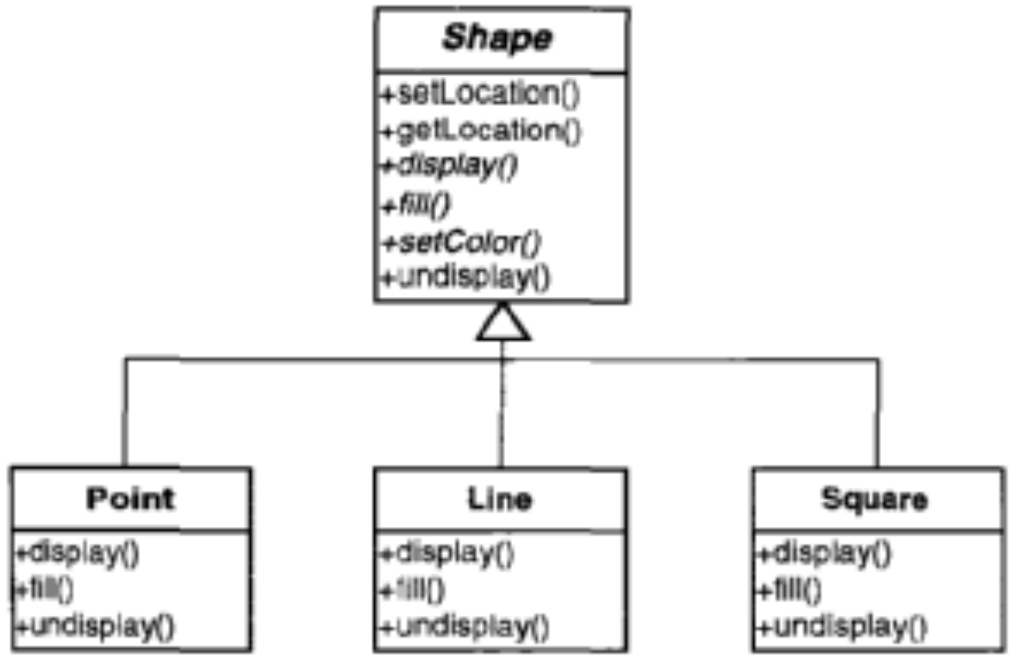


Figure 7-3 Points, Lines, and Squares showing methods.

Learning the Adapter Pattern

- Suppose I am know asked to **implement a circle,** a new kind of Shape (remember requirements always change!).
- To do this, I will want to **create a new class Circle** – that implements the shape ‘circle’ and **derive it from the shape class** so that I can still get polymorphic behaviour.

Learning the Adapter Pattern

- Now, I am faced with the task of having to write the **display, fill and undisplay methods for Circle.** *That could be a pain!*
- Fortunately as I scout around, I discover that Jill down the hall has already written a class called **XXCircle** that deals with circles already.
- Unfortunately, she has named the methods as follows:
  - displayIt
  - fillIt
  - undisplayIt

Learning the Adapter Pattern

- I cannot use XXCircle directly because I want to perserve polymorphic behaviour with Shape. There are two reasons for this:
  - I have **different names and parameter lists**
  - **I cannot derive it** – Not only must the names be the same, but the class must be derived from Shape as well.

Learning the Adapter Pattern

- Rather than change it I **adapt it.**
- I can make a **new class** that does **derive from Shape** and therefore implement's Shape's interface but **avoids rewriting** the circle implementation in XXCircle.
  - Class Circle derives from Shape
  - Circle contains XXCircle
  - Circle passes requests made to the Circle object on through to the XXCircle object

Learning the Adapter Pattern

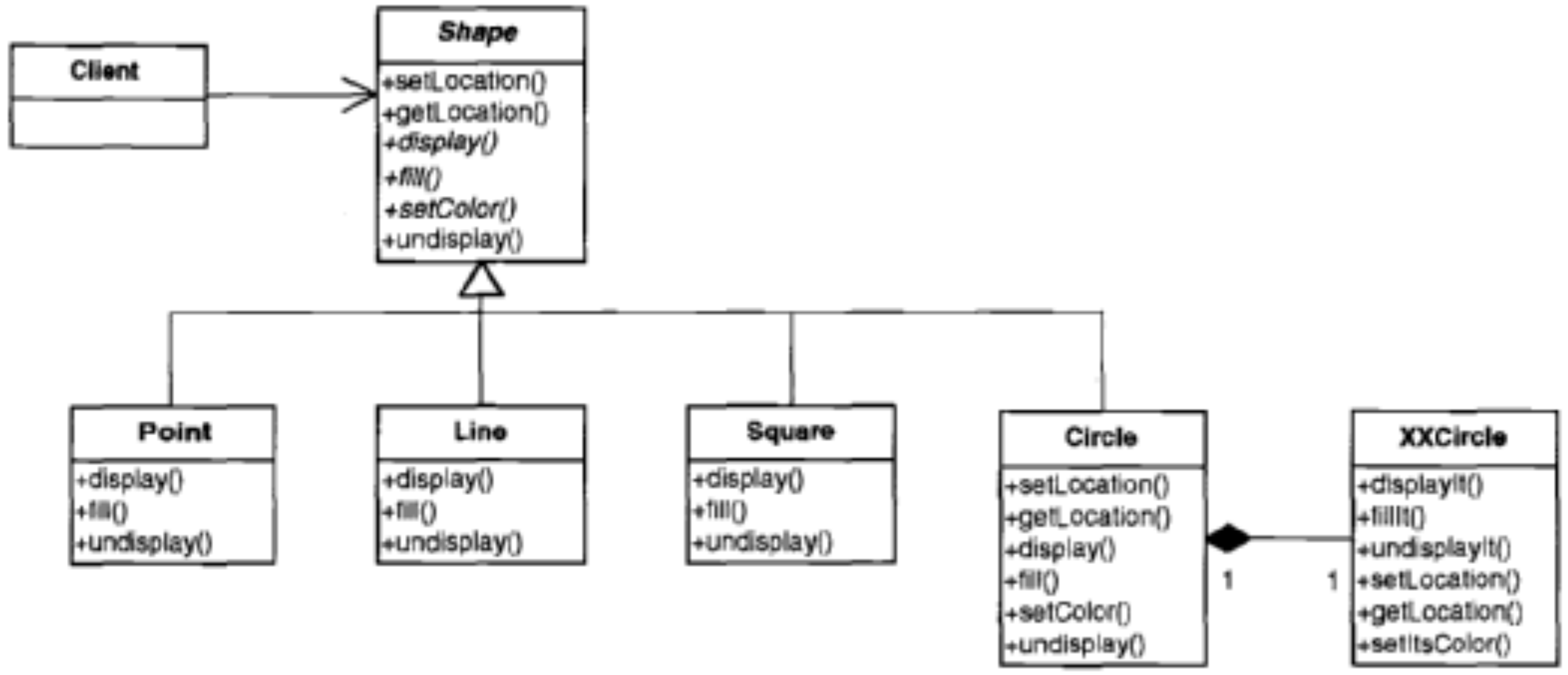
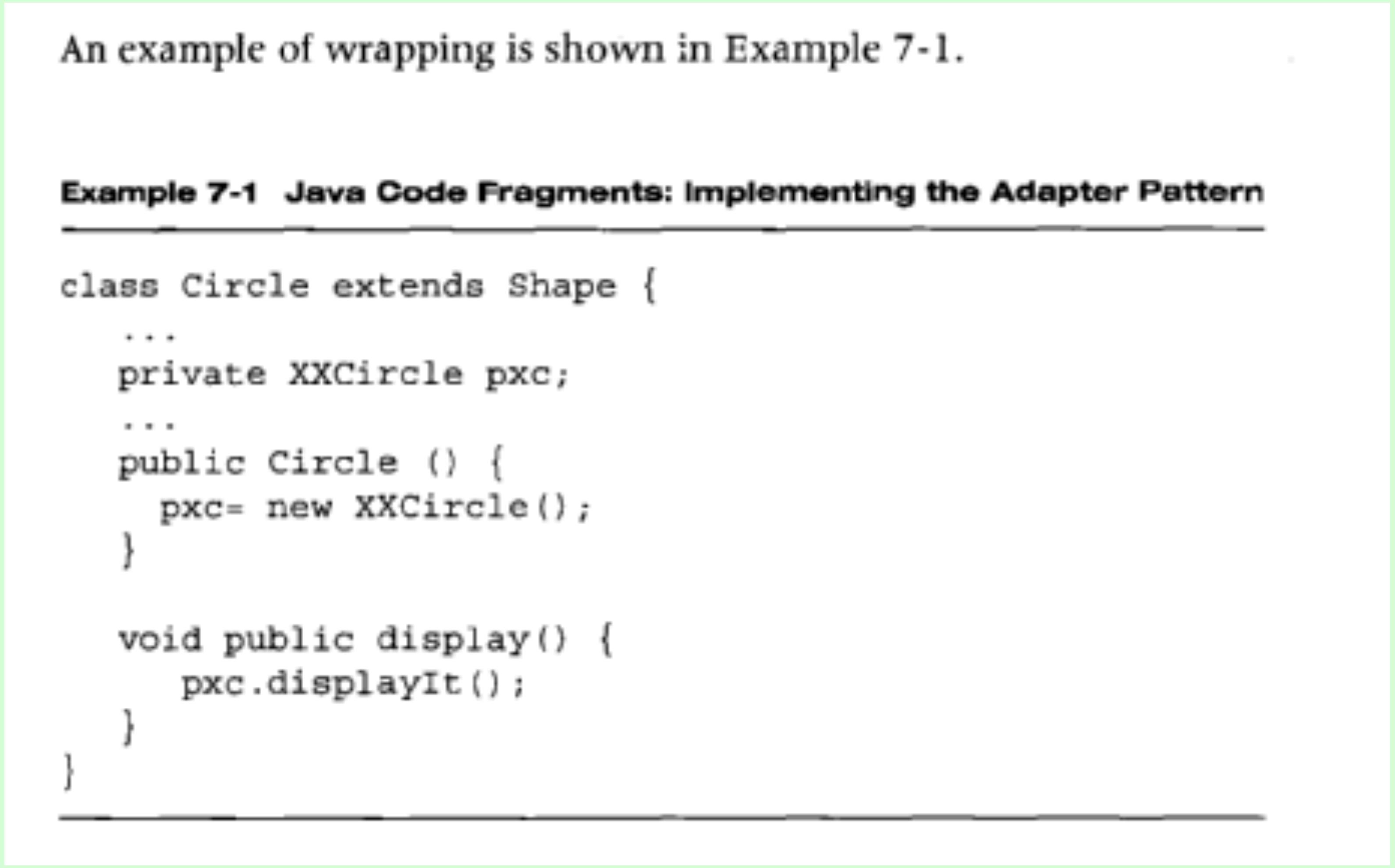


Figure 7-5 The Adapter pattern: `Circle` **"wraps"** `XXCircle`**.**

Learning the Adapter Pattern



An example of wrapping is shown in Example 7-1.

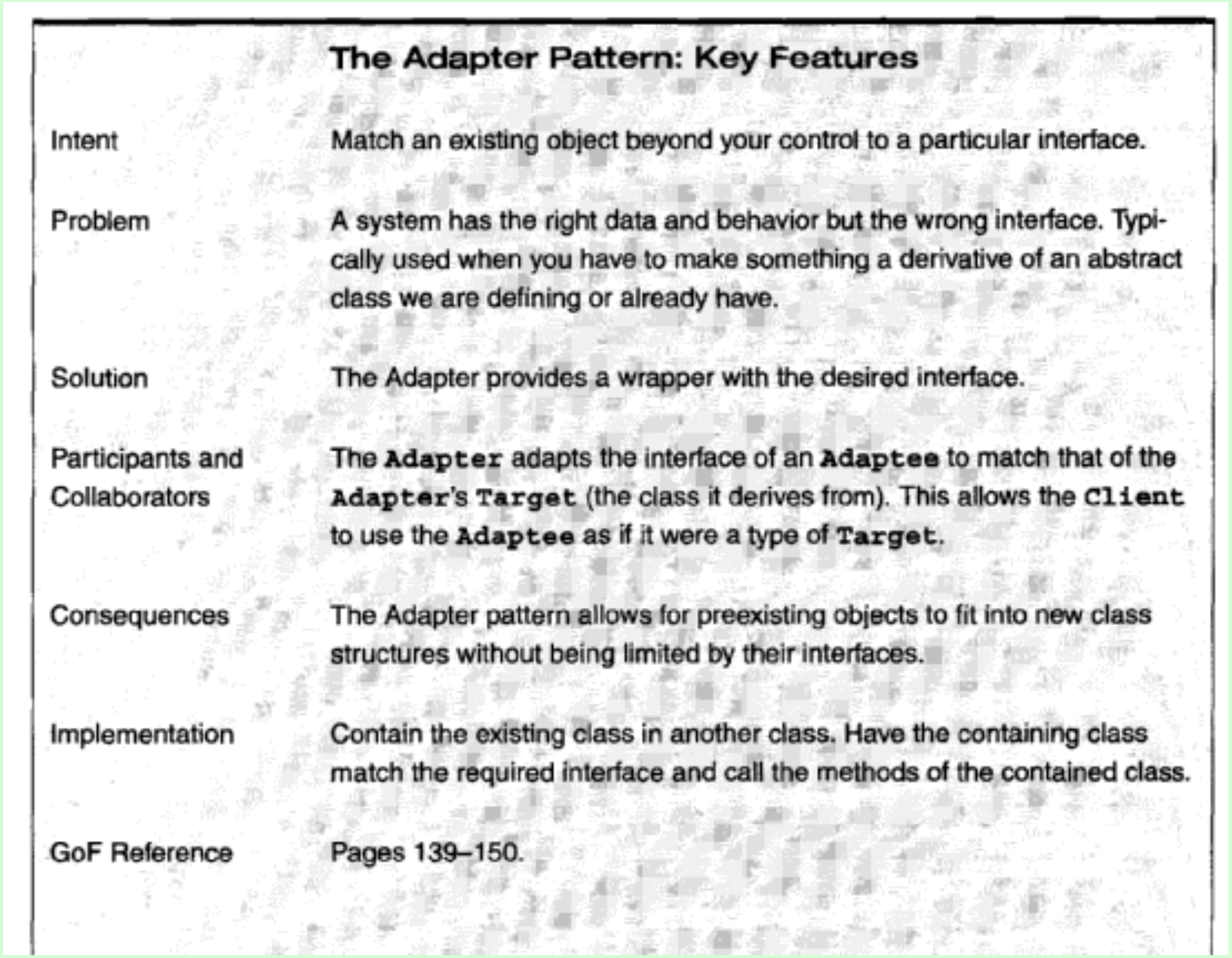
Example 7-1 Java Code Fragments: Implementing the Adapter Pattern

```
class Circle extends Shape {
   ...
   private XXCircle pxc;
   ...
   public Circle () {
     pxc= new XXCircle();
   }

   void public display() {
      pxc.displayIt();
   }
}
```

Learning the Adapter Pattern

- Using the adapter pattern allowed me to **continue using polymorphism** with Shape.
- ie., the client objects of shape **do not know** what types of shape are actually present.
- The **adapter pattern** is most commonly used to **allow for polymorphism**.

The Adapter Pattern



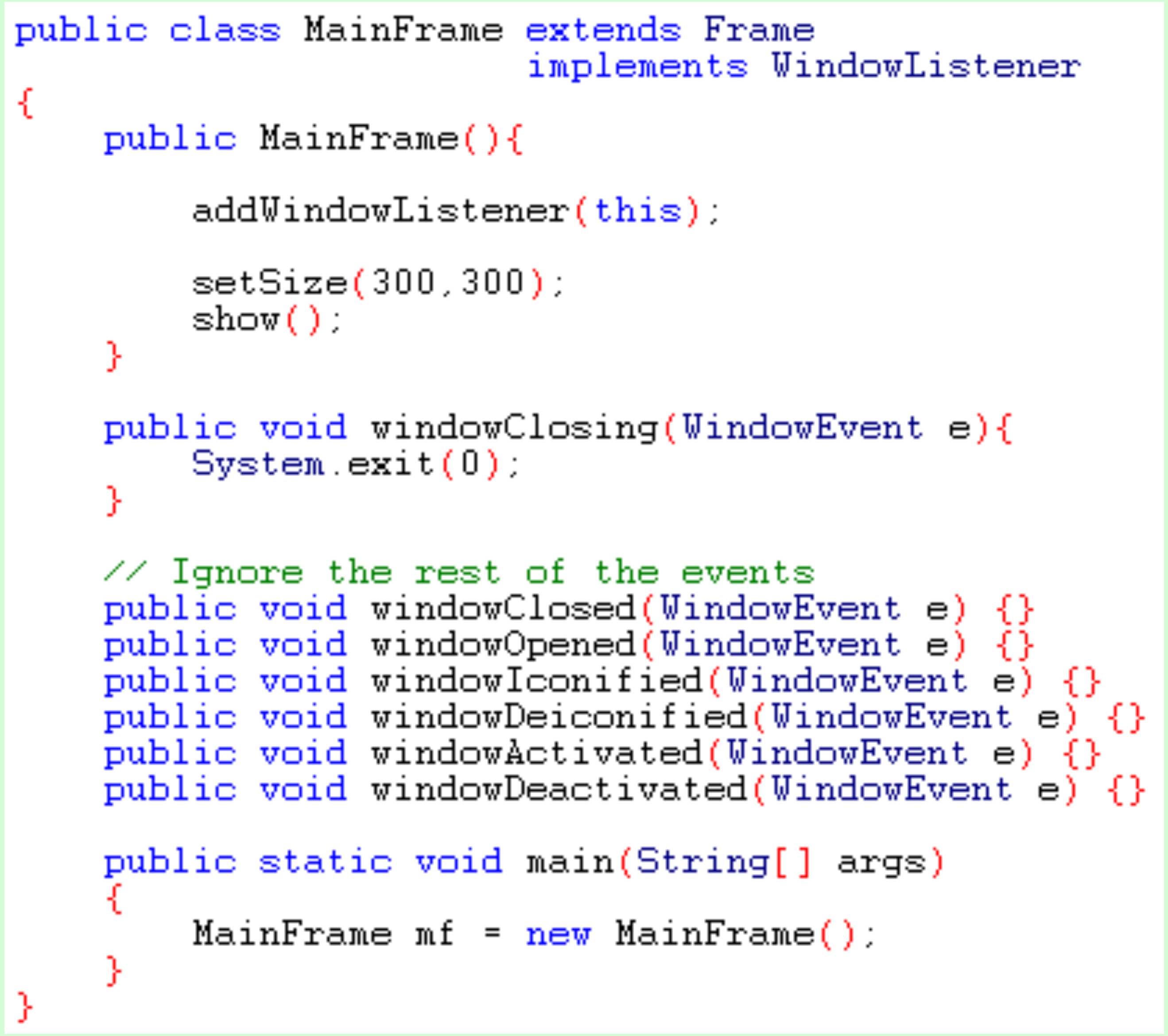
The Adapter Pattern: Key Features

Intent	Match an existing object beyond your control to a particular interface.
Problem	A system has the right data and behavior but the wrong interface. Typically used when you have to make something a derivative of an abstract class we are defining or already have.
Solution	The Adapter provides a wrapper with the desired interface.
Participants and Collaborators	The `Adapter` adapts the interface of an `Adaptee` to match that of the `Adapter`'s `Target` (the class it derives from). This allows the `Client` to use the `Adaptee` as if it were a type of `Target`.
Consequences	The Adapter pattern allows for preexisting objects to fit into new class structures without being limited by their interfaces.
Implementation	Contain the existing class in another class. Have the containing class match the required interface and call the methods of the contained class.
GoF Reference	Pages 139–150.

Adapters in Java

- In a broad sense, **a number of adapters are already built into the Java language.** In this case, the **Java adapters** serve to **simplifly** an unnecessarily complicated event interface.
- One of the most commonly used of these Java adapters is the **WindowAdapter class**.
- One inconvenience of Java is that windows do not close automatically when you click on the Close button or window Exit menu item. The general solution to this problem is to have your main Frame Window implement the **WindowListener interface** and leave all of the Window events empty except for *windowClosing*.

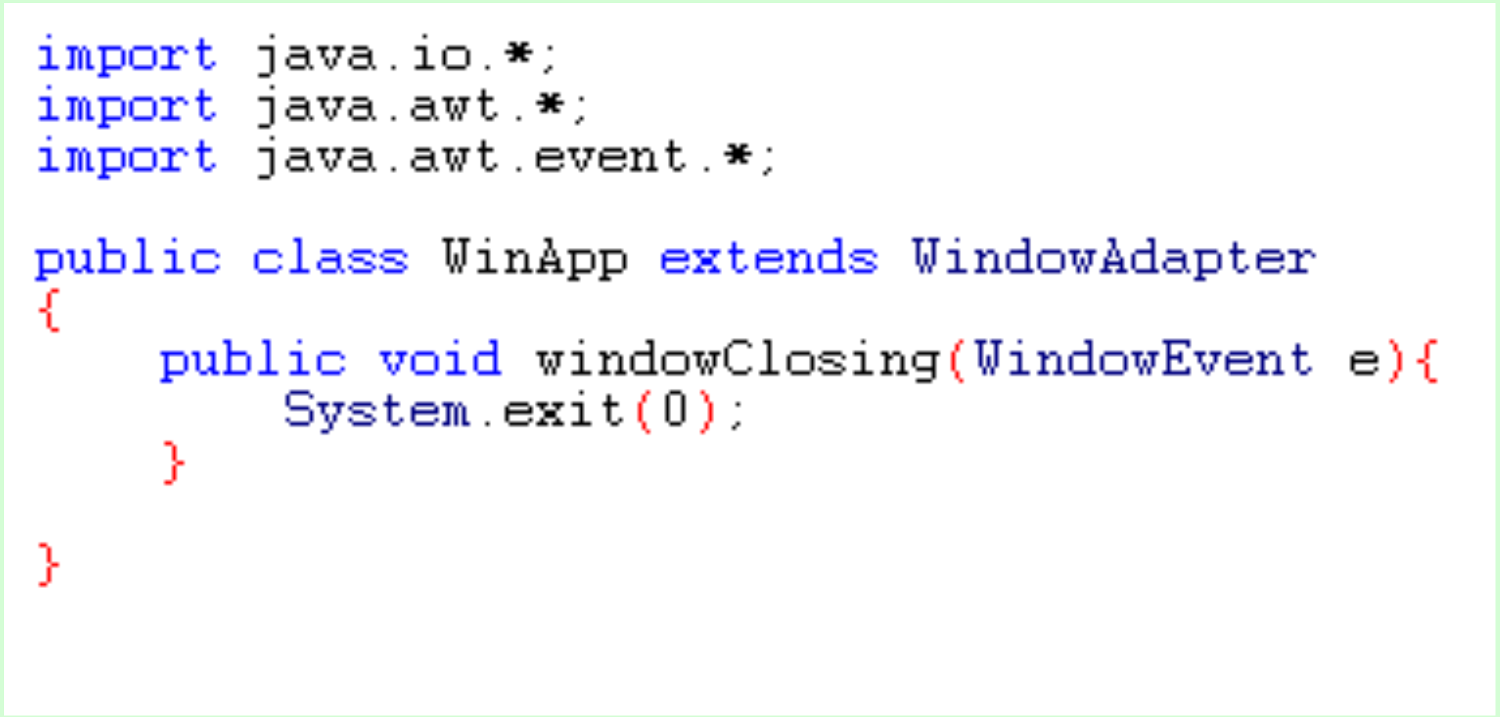
Adapters in Java



```
public class MainFrame extends Frame
                       implements WindowListener
{
    public MainFrame(){

        addWindowListener(this);

        setSize(300,300);
        show();
    }

    public void windowClosing(WindowEvent e){
        System.exit(0);
    }

    // Ignore the rest of the events
    public void windowClosed(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}

    public static void main(String[] args)
    {
        MainFrame mf = new MainFrame();
    }
}
```

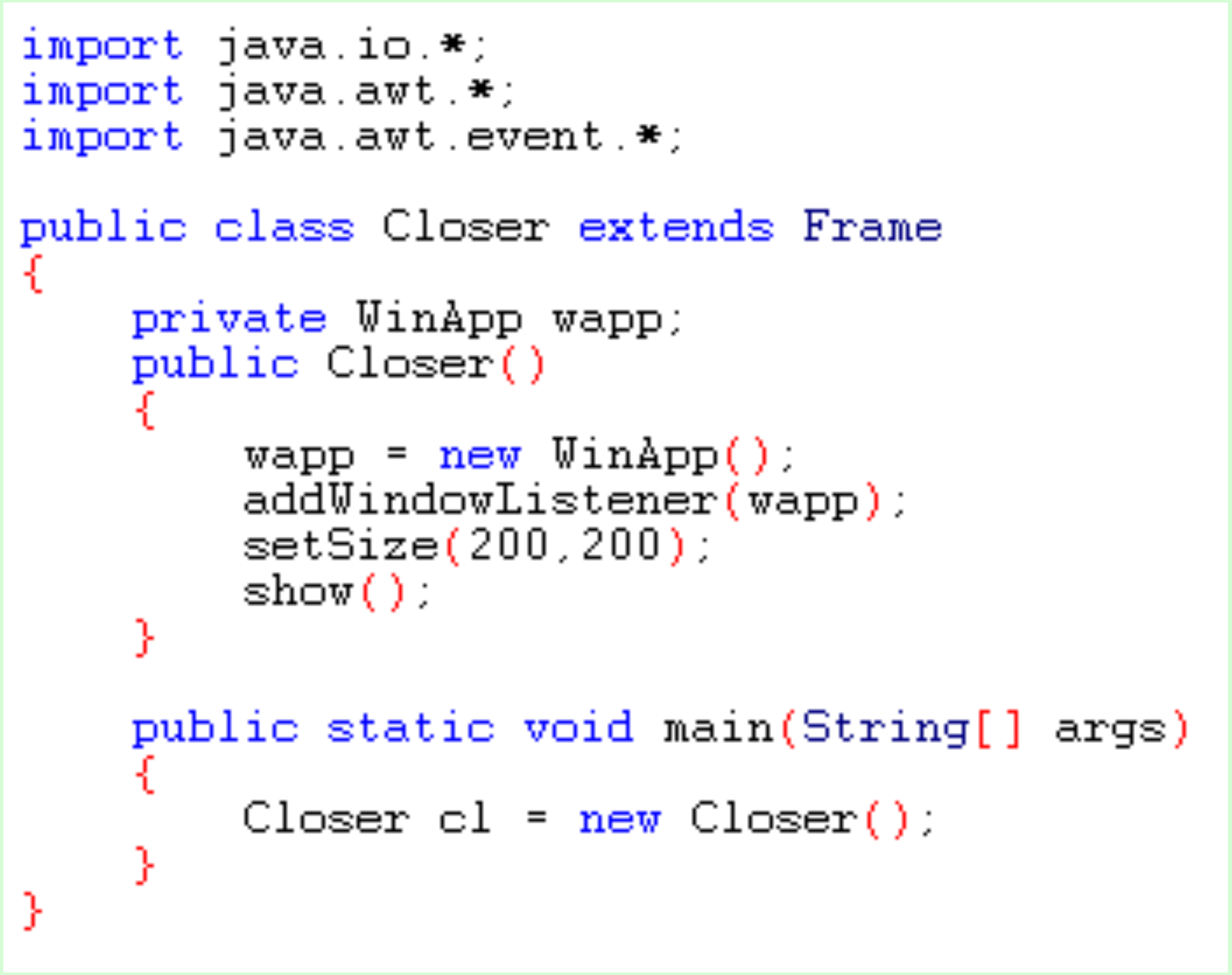
Adapters in Java

- As you can see this code is awkward and hard to read. The Window Adapter class is provided to simplify this procedure. This class contains empty implementations of all **seven** of the previous WindowEvents. You then need only to override the windowClosing event and insert the appropriate exit code.



```
import java.io.*;
import java.awt.*;
import java.awt.event.*;

public class WinApp extends WindowAdapter
{
    public void windowClosing(WindowEvent e){
        System.exit(0);
    }

}
```

Adapters in Java



```
import java.io.*;
import java.awt.*;
import java.awt.event.*;

public class Closer extends Frame
{
    private WinApp wapp;
    public Closer()
    {
        wapp = new WinApp();
        addWindowListener(wapp);
        setSize(200,200);
        show();
    }

    public static void main(String[] args)
    {
        Closer cl = new Closer();
    }
}
```

Adapters in Java

- Adapters like these are common in Java when a simple class can be used to encapsulate a number of events
  .
- They include ComponentAdapter, ContainerAdapter, FocusAdapter,KeyAdapter,MouseAdapter, and MouseMotionAdapter.

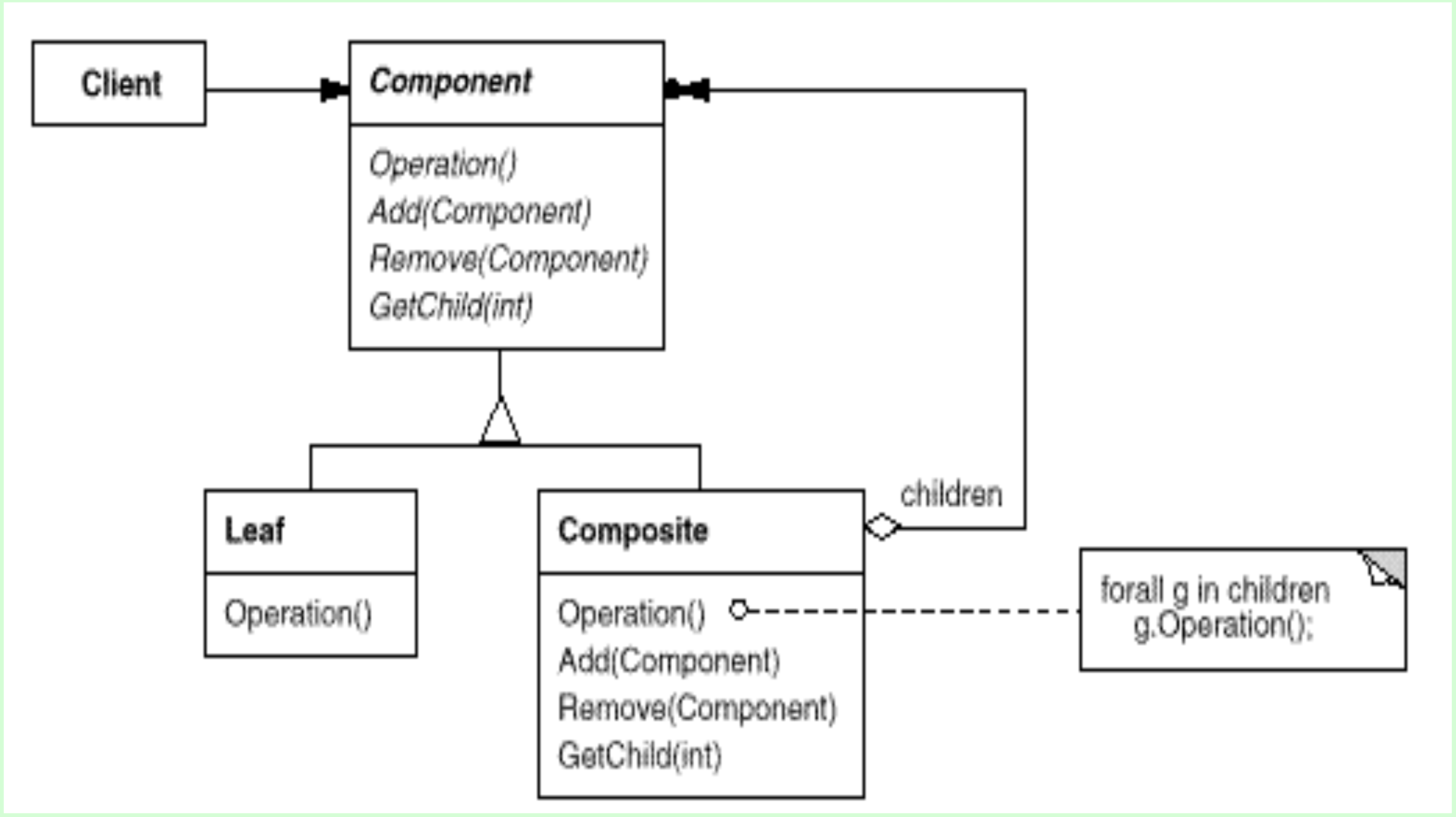
Composite Pattern

The Composite Pattern

- **Intent:**

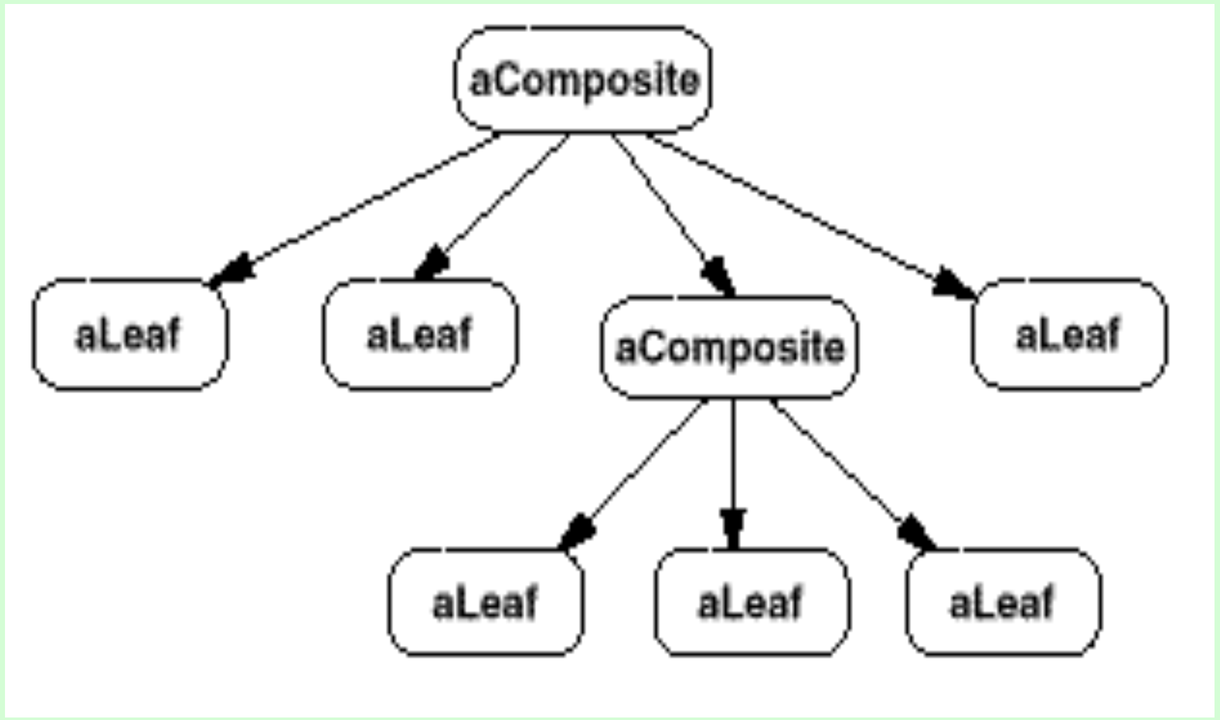
  Compose **objects** into **tree structures** to **represent part-whole hierarchies**.
  Composite lets clients treat individual objects and compositions of objects uniformly.

Composite Pattern Structure



Composite Pattern Structure

- A typical Composite object structure might look like this:



Composite Pattern Participants

- **Component (Abstract Employee)**
  - declares the **interface for objects in the composition**.
  - implements default behavior for the interface common to all classes, as appropriate.
  - declares **an interface** for **accessing and managing** its child components.
  - (optional) defines an interface for **accessing a component's parent** in the recursive structure, and implements it if that's appropriate.

Composite Pattern Participants

- **Leaf (Employee)**
  - represents **leaf objects** in the composition. A leaf **has no children.**
  - defines behavior for primitive objects in the composition.

Composite Pattern Participants

- **Composite (Boss)**
  - A group of objects in which **some objects may contain others**, thus one object may represent groups and an other may represent an individual item
  - 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.

The Composite Pattern

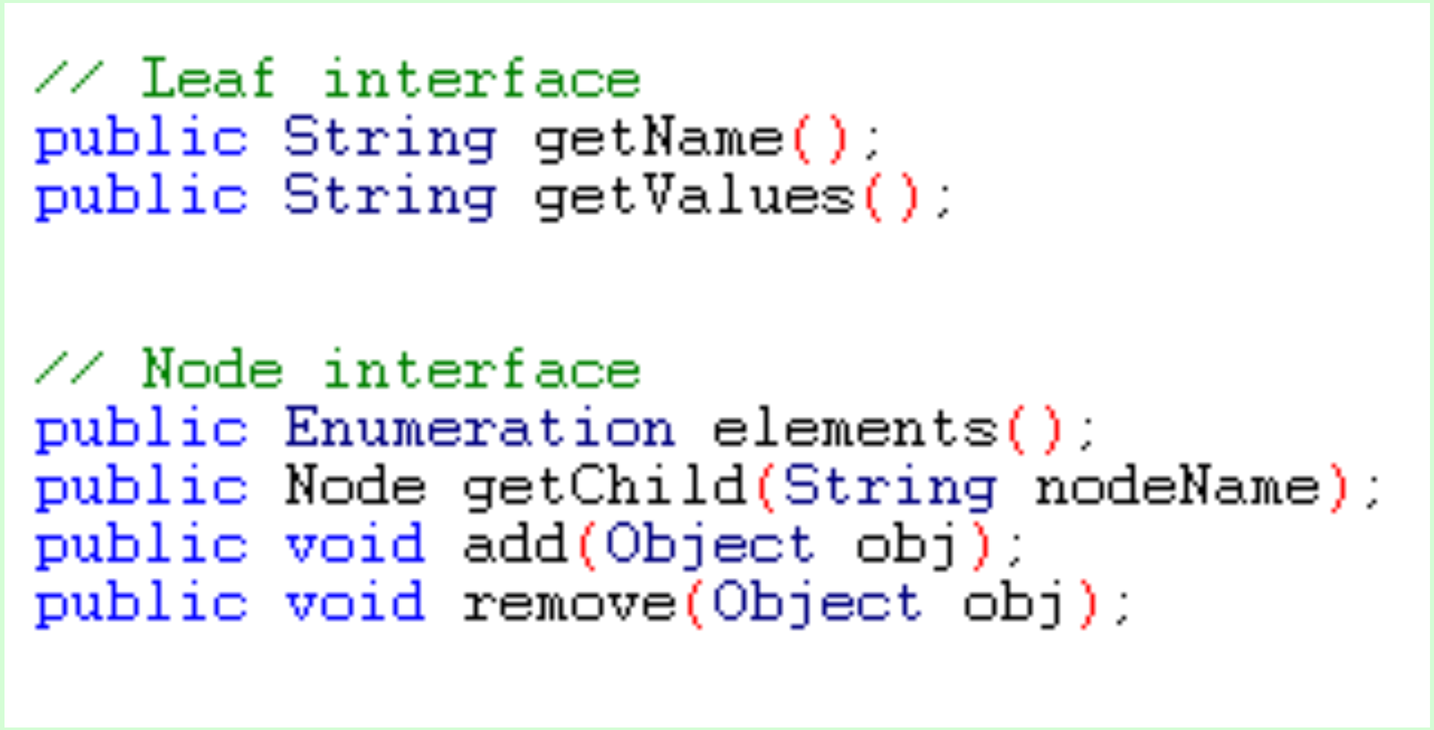
- Programmers often develop systems in which a component may be an **individual object** or **may represent a collection of objects**. The Composite pattern is designed to accommodate both cases. You can use it to **build part-whole hierarchies** or to construct data **representations of trees.**

- In summary a composite is a collection of objects, any one of which may be either a **composite** or a **primitive object**. In tree terminology, some objects may be nodes with additional branches, and some may be leaves.

- A problem that develops is the ability to distinguish between nodes and leaves. **Nodes have children** and can have children added to them, while **leaves do not at the moment have children**.

The Composite Pattern

- Some developers have suggested creating a separate interface for nodes and leaves, where a leaf could have the following methods:



```
// Leaf interface
public String getName();
public String getValues();


// Node interface
public Enumeration elements();
public Node getChild(String nodeName);
public void add(Object obj);
public void remove(Object obj);
```

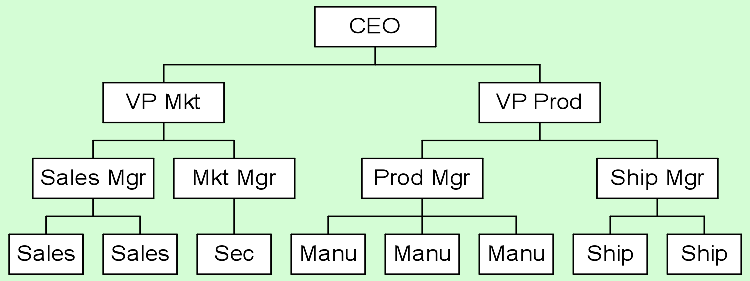
The Composite Pattern

- This then leaves us with the problem of deciding which elements will be which when we construct the composite.

- Java makes this quite easy for us, since every node or leaf can return an **Enumeration of the contents of the Vector** where the children are stored. If there **are no children**, the *hasMoreElements* method **returns false** at once.
  Thus if we simply obtain the Enumeration from each element, we can quickly determine whether it has any children by checking the return values of *hasMoreElements*.

Composite Example

- Lets consider a small company that was started by one person who got the business going. Then he hired a couple of people to handle the marketing and manufacturing. Soon each of them hired some assistants and the company expanded into the organization shown below:



Composite Example

- If the company is successful, each of the employee receives a salary. At any point in time we could be asked for the control span of any employee.

- We define the control span cost as the salary of that person as well as the combined salaries of all of his or her subordinates. Here is an ideal example for a composite:
  - **The cost of an individual employee is that employee's salary**
  - **The cost of an employee who heads a department is that employee's salary plus the salaries of all employees that the employee manages**.

Composite Example

- We would like a single interface that will produce the salary totals correctly, whether or not the employee has subordinates.
  - **public float getSalaries();**

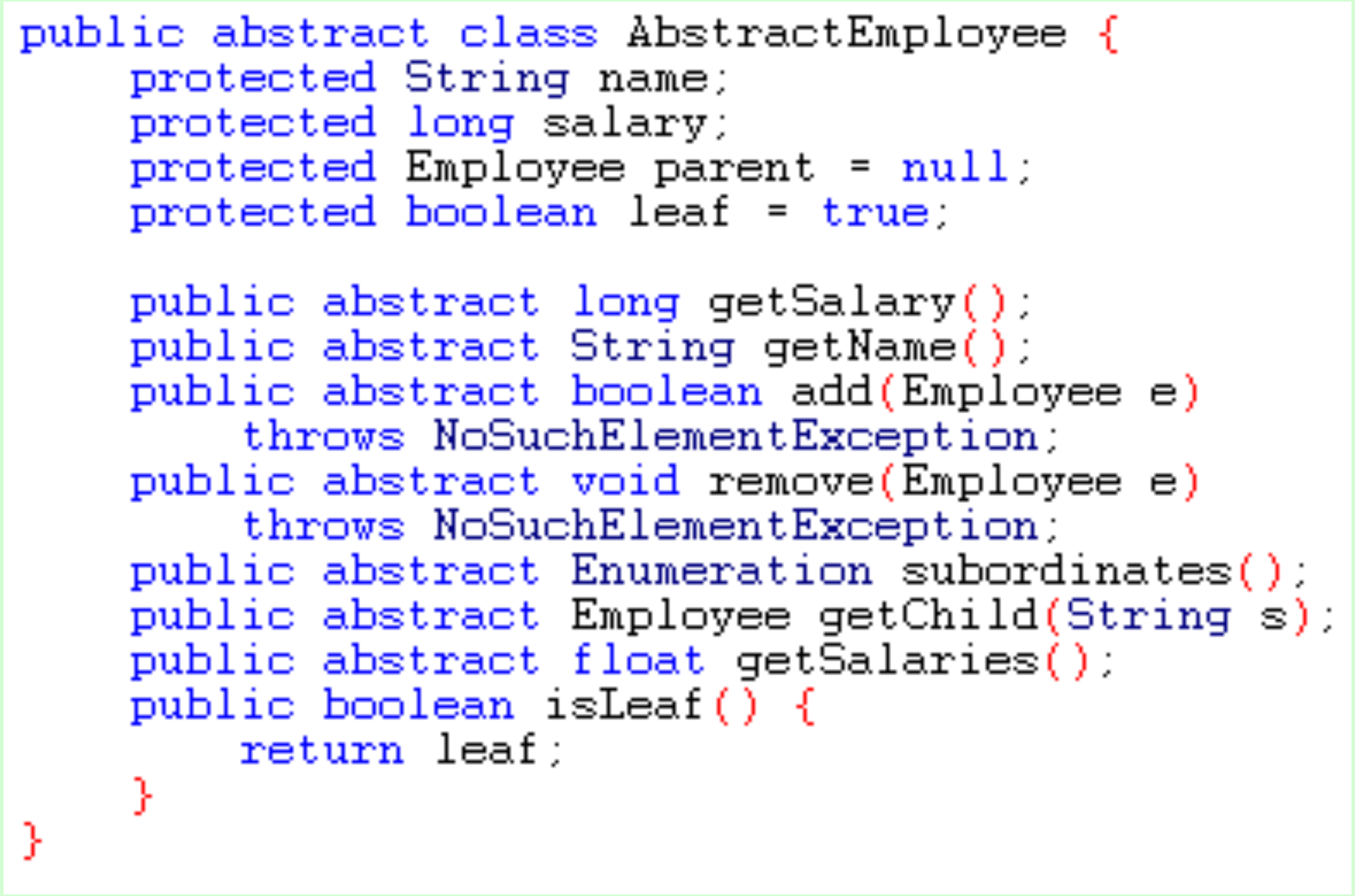
- We could now imagine representing the company as a composite made up of node: managers and employees. We could use a single class to represent all employees, but since each level might have different properties, defining at least two classes might be more useful: ***Employees*** and ***Bosses***.

  - **Employees are leaf nodes**

  - **Bosses are nodes that may have Employee nodes under them**

- We'll start with the AbstractEmployee class and derive our concrete employee classes from it.

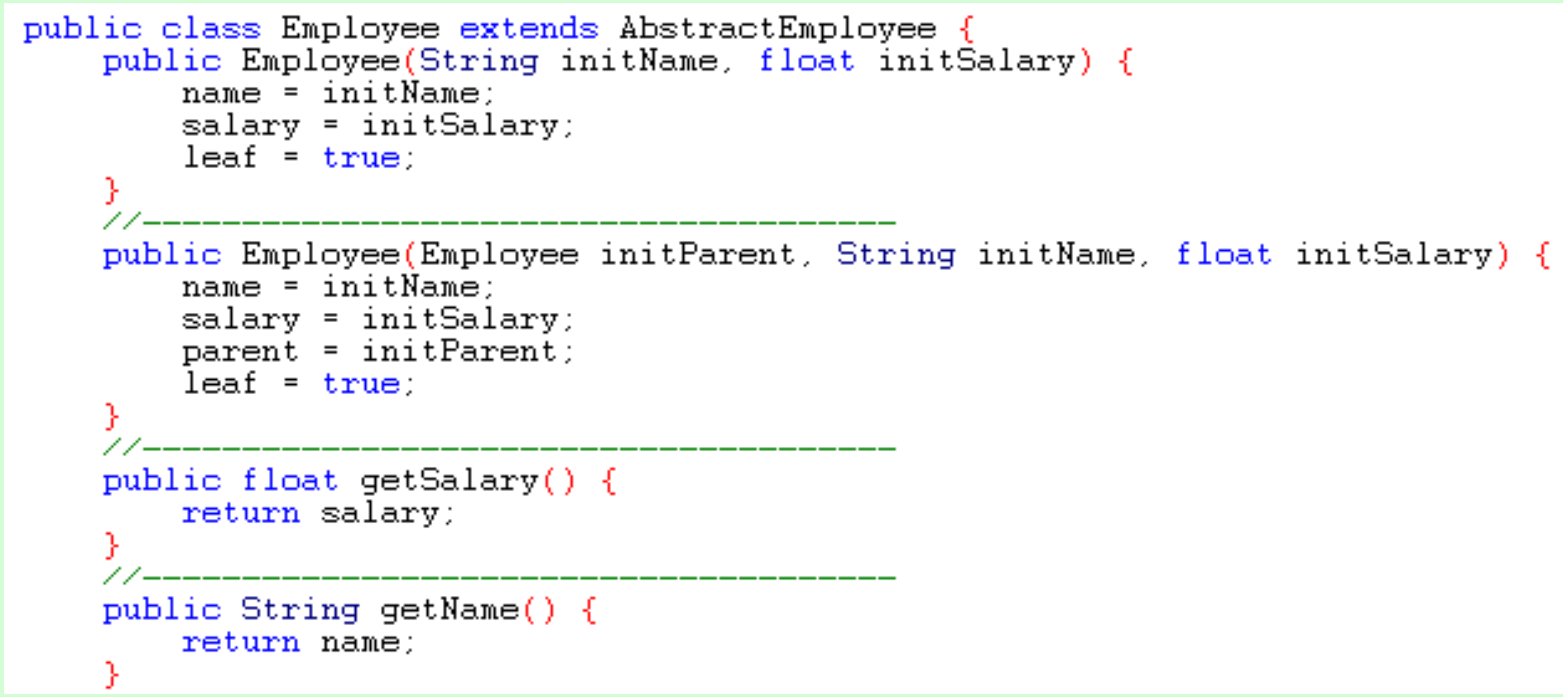
Composite Example



```java
public abstract class AbstractEmployee {
    protected String name;
    protected long salary;
    protected Employee parent = null;
    protected boolean leaf = true;

    public abstract long getSalary();
    public abstract String getName();
    public abstract boolean add(Employee e)
        throws NoSuchElementException;
    public abstract void remove(Employee e)
        throws NoSuchElementException;
    public abstract Enumeration subordinates();
    public abstract Employee getChild(String s);
    public abstract float getSalaries();
    public boolean isLeaf() {
        return leaf;
    }
}
```

Employee Class

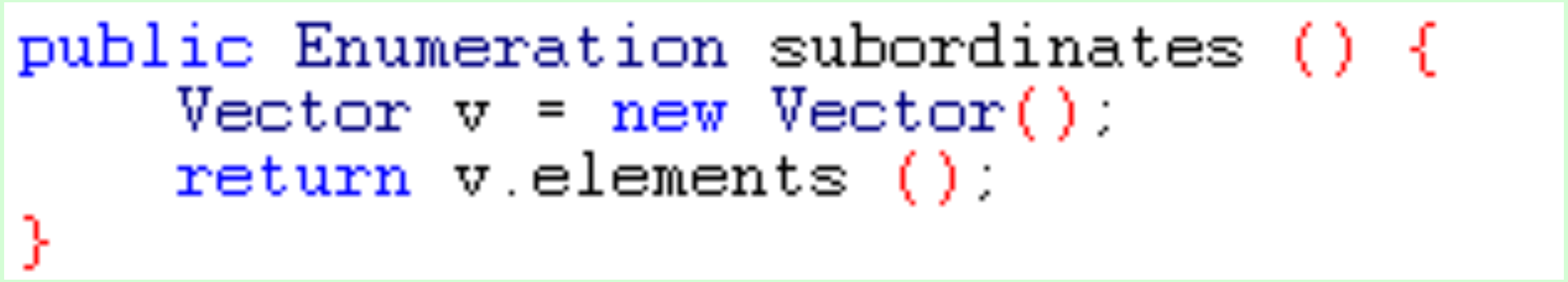
- Our concrete Employee class stores the name and salary of each employee and allows us to fetch them as needed.



```java
public class Employee extends AbstractEmployee {
    public Employee(String initName, float initSalary) {
        name = initName;
        salary = initSalary;
        leaf = true;
    }
    //-------------------------------------
    public Employee(Employee initParent, String initName, float initSalary) {
        name = initName;
        salary = initSalary;
        parent = initParent;
        leaf = true;
    }
    //-------------------------------------
    public float getSalary() {
        return salary;
    }
    //-------------------------------------
    public String getName() {
        return name;
    }
```

Employee Class

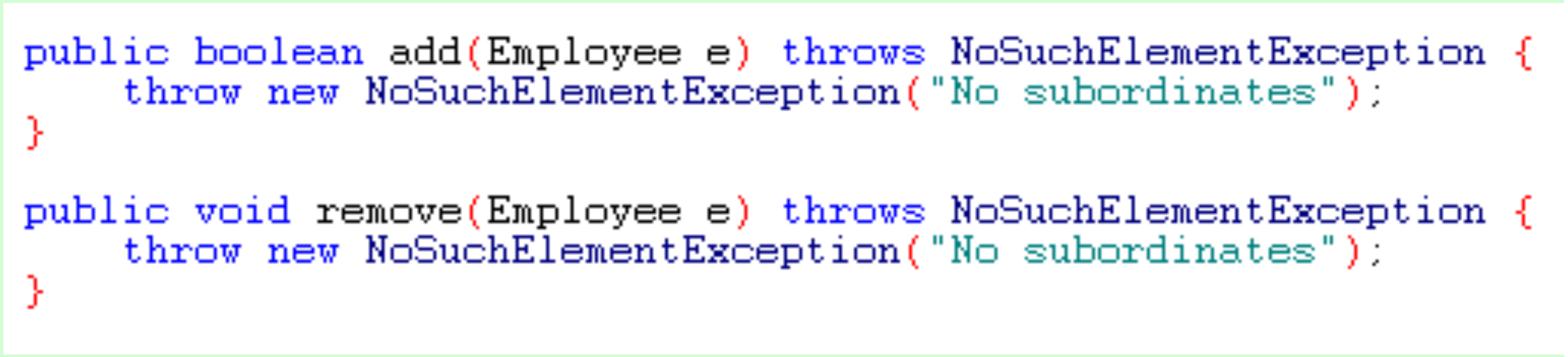
- The Employee class must have **concrete implemenations** of the add, remove, and getChild.
- Since the Employee is a leaf, all of these will return some sort of error indication. For example, *subordinates* could return null, but it would be better if they return an empty Enumeration.



```
public Enumeration subordinates () {
    Vector v = new Vector();
    return v.elements ();
}
```

Employee Class

- The *add* and *remove* methods must generate errors, since members of the basic Employee class cannot have subordinates.



```
public boolean add(Employee e) throws NoSuchElementException {
    throw new NoSuchElementException("No subordinates");
}

public void remove(Employee e) throws NoSuchElementException {
    throw new NoSuchElementException("No subordinates");
}
```

Boss Class

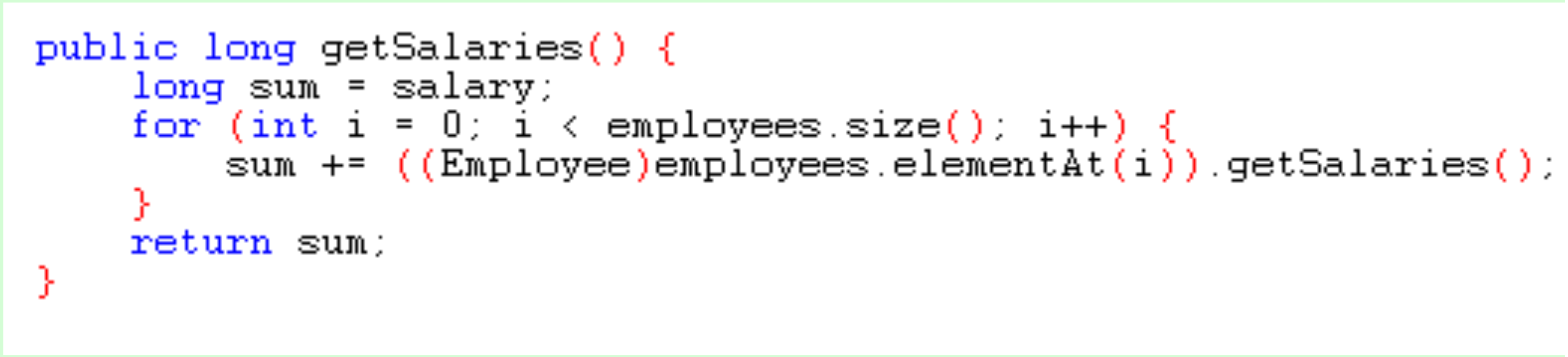
- The Boss class is a **subclass of Employee** and allows us to store subordinate employees as well.

- We'll store them in a **Vector called subordinates**, which we return through an Enumeration. Thus if a particular Boss has temporarily run out of Employees, the Enumeration will be empty.

```java
public class Boss extends Employee {
    Vector employees;

    public Boss(String initName, long initSalary) {
        super(initName, initSalary);
        leaf = false;
        employees = new Vector();
    }

    public Boss(Employee initParent, String initName, long initSalary) {
        super(initParent, initName, initSalary);
        leaf = false;
        employees = new Vector();
    }

    public Boss(Employee emp) {
        //promotes an employee position to a Boss
        //and thus allows it to have employees
        super(emp.getName (), emp.getSalary());
        employees = new Vector();
        leaf = false;
    }

    public boolean add(Employee e) throws NoSuchElementException {
        employees.add(e);
        return true;
    }

    public void remove(Employee e) throws NoSuchElementException {
        employees.removeElement(e);
    }

    public Enumeration subordinates () {
        return employees.elements ();
    }
```

Boss Class

- If you want to get a list of employees of a given manager, you obtain an Enumeration of them directly from the subordinates Vector. Similiarly, you can use this same Vector to return a sum of salaries for any employee and his or her subordinates.

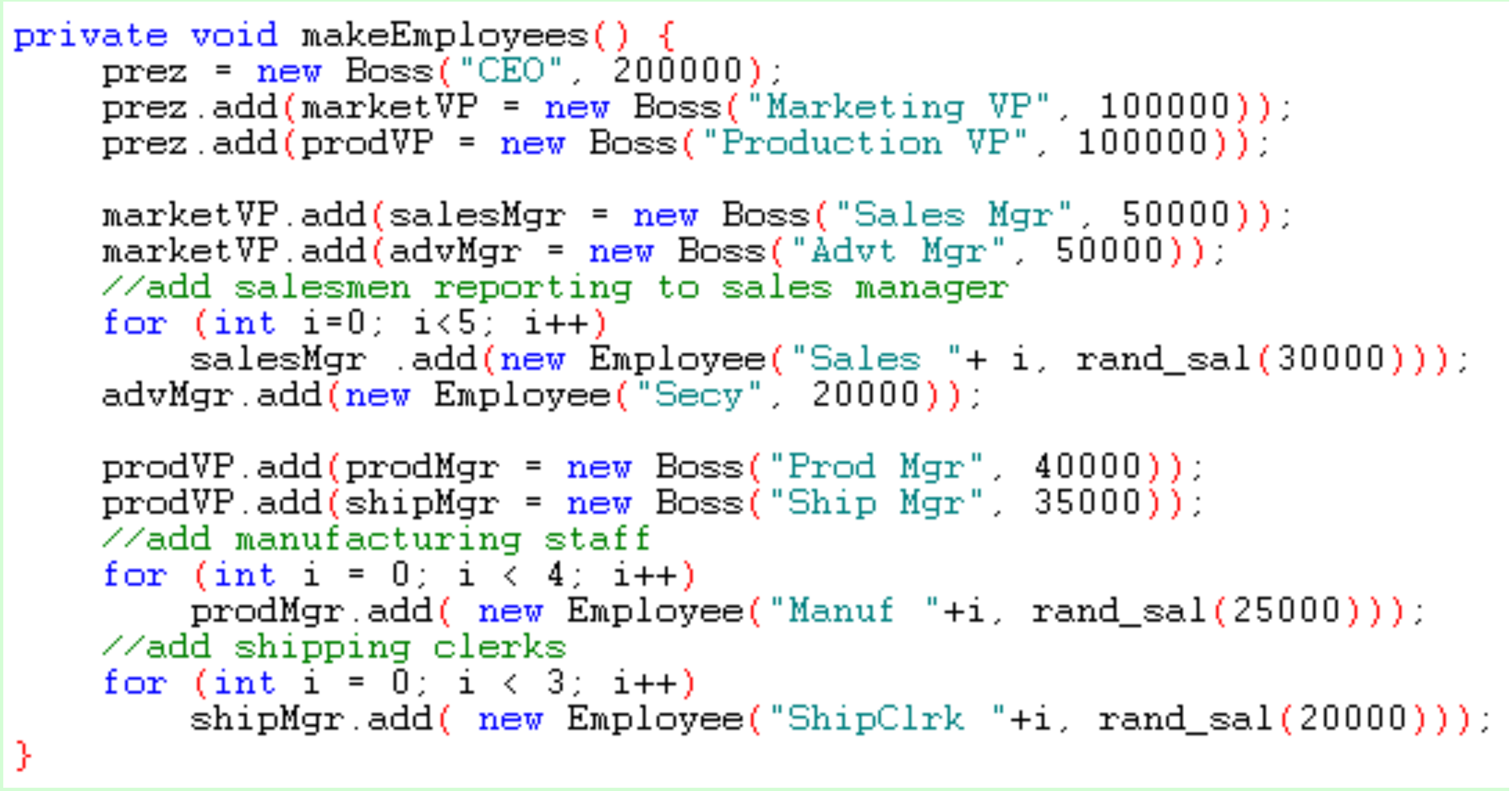


```
public long getSalaries() {
    long sum = salary;
    for (int i = 0; i < employees.size(); i++) {
        sum += ((Employee)employees.elementAt(i)).getSalaries();
    }
    return sum;
}
```

- Note that this method starts with the salary of the current Employee and then calls the *getSalaries* method on each subordinate. This is of course recursive, and any employees who themselves have subordinates will be included.

Creating the Composite

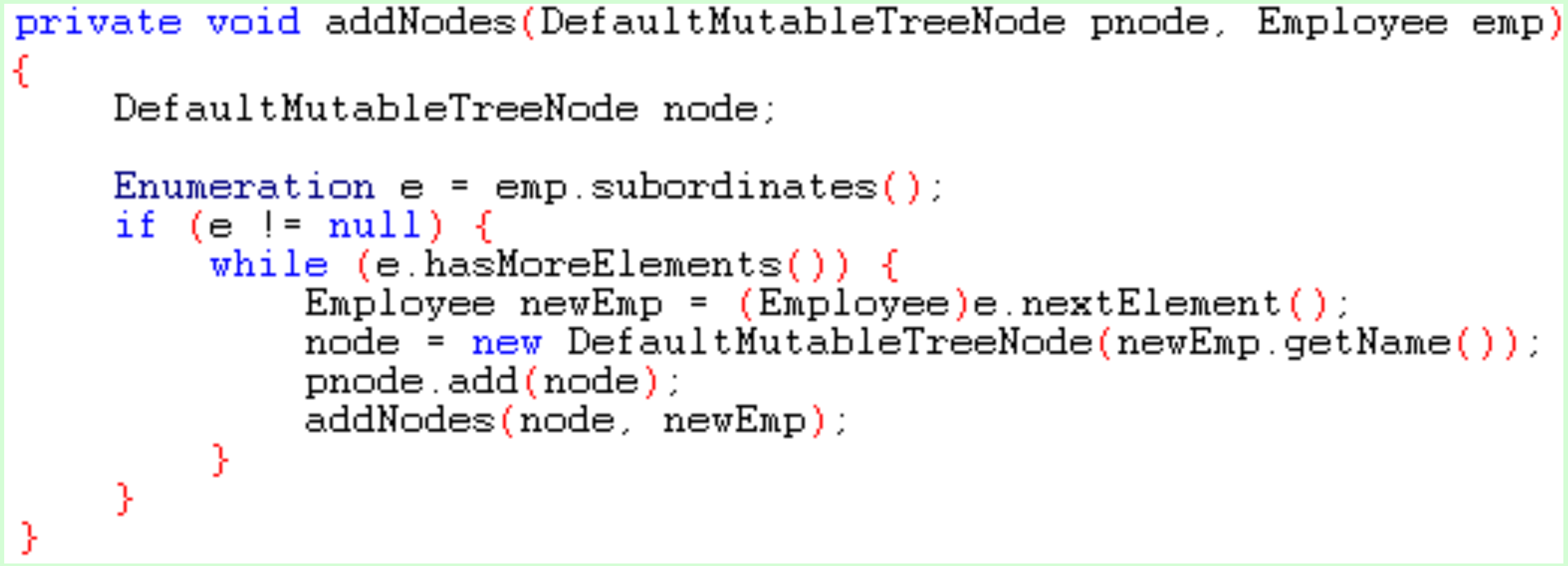
- We start by creating a CEO Employee and then add the employees subordinates and their subordinates as follows:



```
private void makeEmployees() {
    prez = new Boss("CEO", 200000);
    prez.add(marketVP = new Boss("Marketing VP", 100000));
    prez.add(prodVP = new Boss("Production VP", 100000));

    marketVP.add(salesMgr = new Boss("Sales Mgr", 50000));
    marketVP.add(advMgr = new Boss("Advt Mgr", 50000));
    //add salesmen reporting to sales manager
    for (int i=0; i<5; i++)
        salesMgr .add(new Employee("Sales "+ i, rand_sal(30000)));
    advMgr.add(new Employee("Secy", 20000));

    prodVP.add(prodMgr = new Boss("Prod Mgr", 40000));
    prodVP.add(shipMgr = new Boss("Ship Mgr", 35000));
    //add manufacturing staff
    for (int i = 0; i < 4; i++)
        prodMgr.add( new Employee("Manuf "+i, rand_sal(25000)));
    //add shipping clerks
    for (int i = 0; i < 3; i++)
        shipMgr.add( new Employee("ShipClrk "+i, rand_sal(20000)));
}
```

Creating the Composite

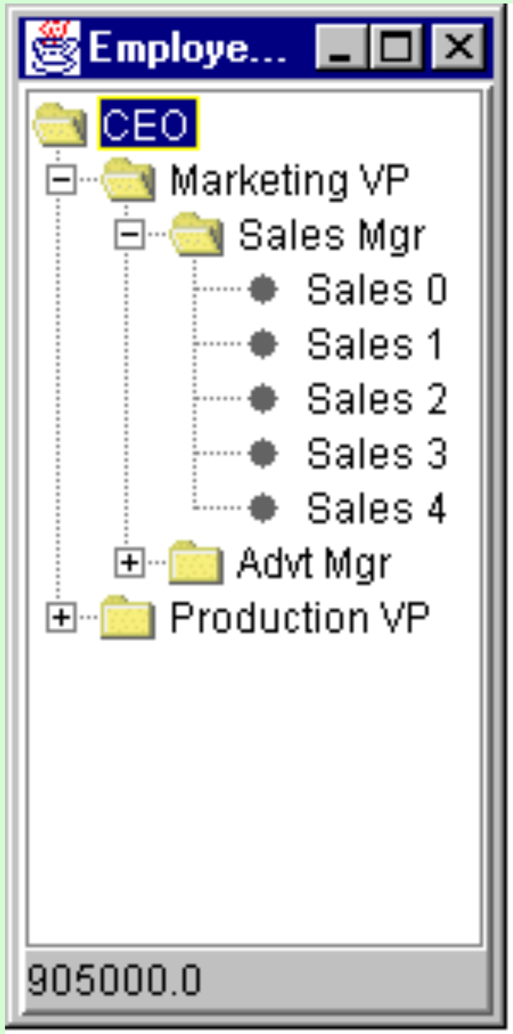
- Once we have constructed this Composite structure, we can load a visual JTree list by starting at the top node and calling the *addNode* method recursively until all of the leaves in each node are processed.



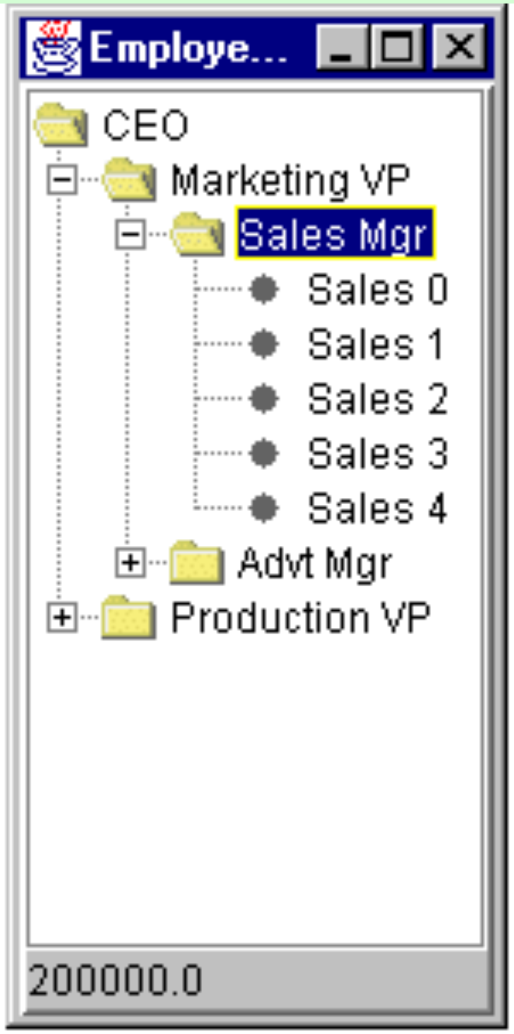
```
private void addNodes(DefaultMutableTreeNode pnode, Employee emp)
{
    DefaultMutableTreeNode node;

    Enumeration e = emp.subordinates();
    if (e != null) {
        while (e.hasMoreElements()) {
            Employee newEmp = (Employee)e.nextElement();
            node = new DefaultMutableTreeNode(newEmp.getName());
            pnode.add(node);
            addNodes(node, newEmp);
        }
    }
}
```

Final Program

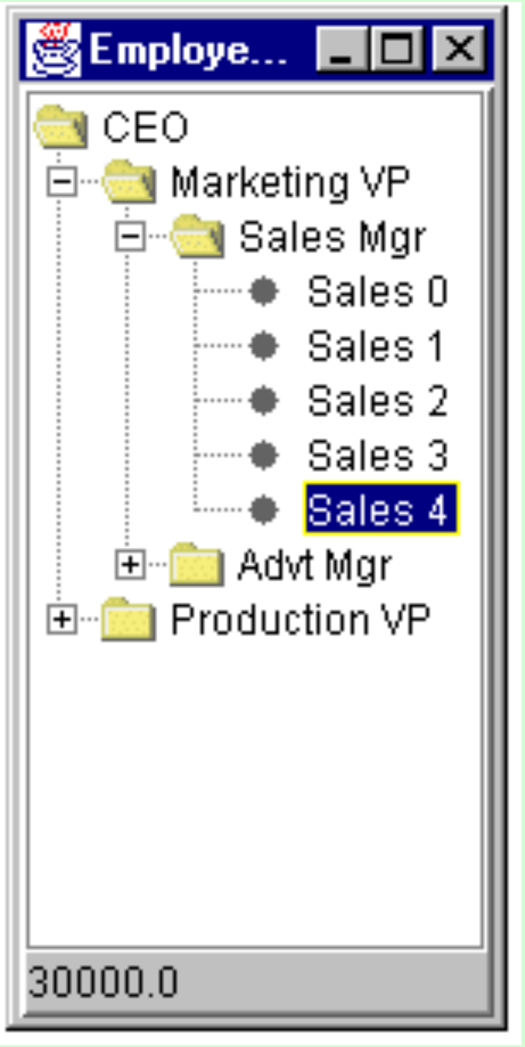
- Final Program



```
905000.0
```



```
200000.0
```



```
30000.0
```

Composite Consequences

- The composite pattern allows you to **define a class hierarchy of simple objects** and more-complex composite objects so that they appear to be the same to the client program.

- Because of this simplicity, the client can be that much simpler since nodes and leaves are handled in the same way.

Facade

- **Intent:**

  Provide a **unified interface to a set of interfaces in a subsystem**.
  Facade defines a higher-level interface that makes the subsystem easier to use.

Basically, this is saying that we need a **new way** to **interact with a system** that is **easier** than the current way, or

We need to use the system in a particular way.

We can build such a method of interaction because we only need to **use a subset** of the system in question.

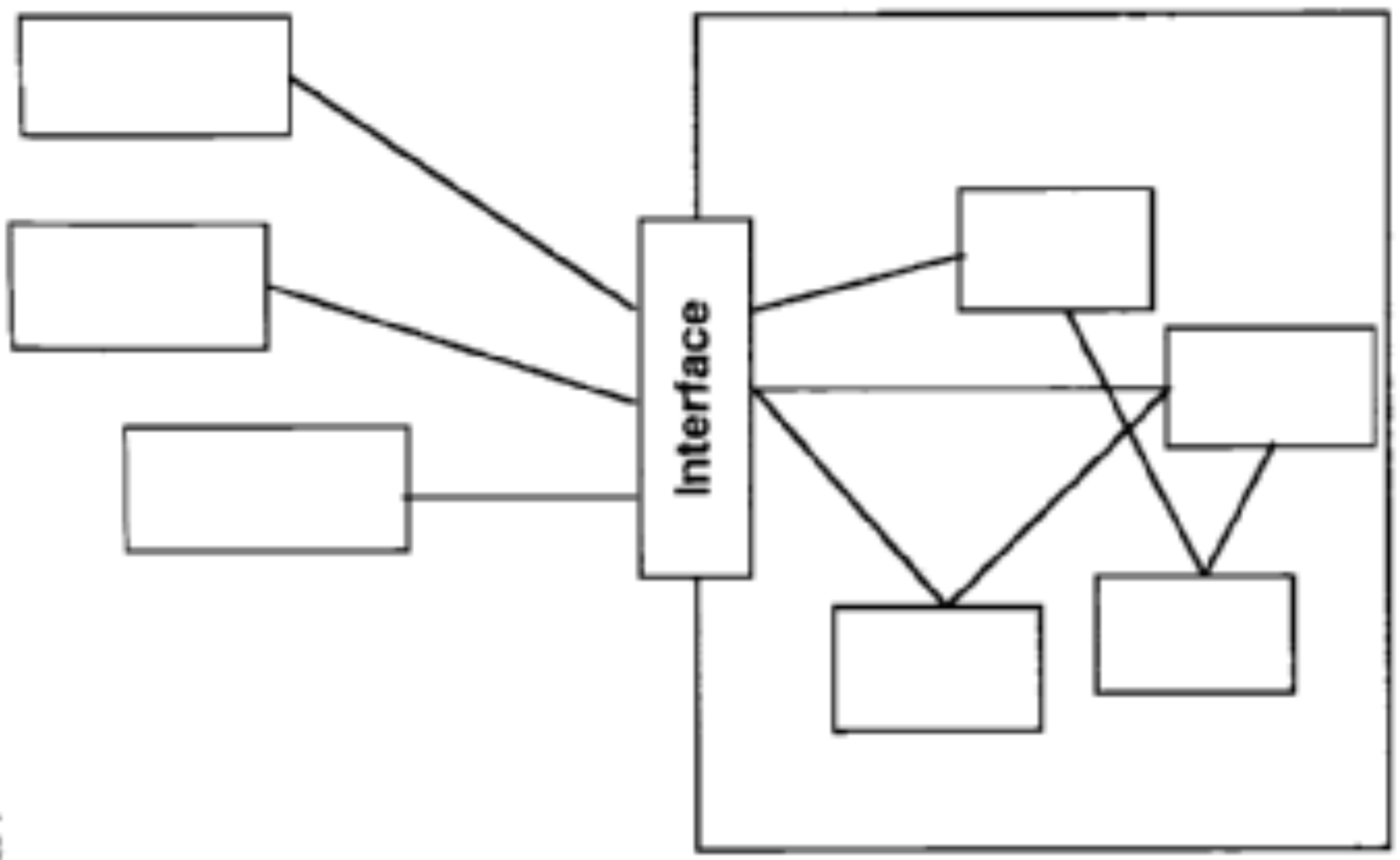
Facade

- Example
  Let’s say that you are working in an IT company where all of the company documentation for a particular system (CAD/CAM) consists of 8 feet of manuals with each page 9 by 11 inches and in small print. This is a pretty complex document system. Suppose that you were asked to learn about this system – there would be a fair bit of reading in 8 feet of manuals.
  Now, if you and say 5 other people were working on a project that needed to use the system, not all of ye would have to learn the entire thing.
  Rather than waste everyone’s time, you would probably draw straws, and the loser would have to write routines that the rest of ye would use to interface with the system

Facade

- This person would determine how you and others on our team were going to use the system and what APIs would be best for our particular needs.
  He/She would create a new class or classes that had the interface you required. Then you and the rest of the programming community could use this new interface without having to learn the entire complicated system (see figure on next page)

Facade

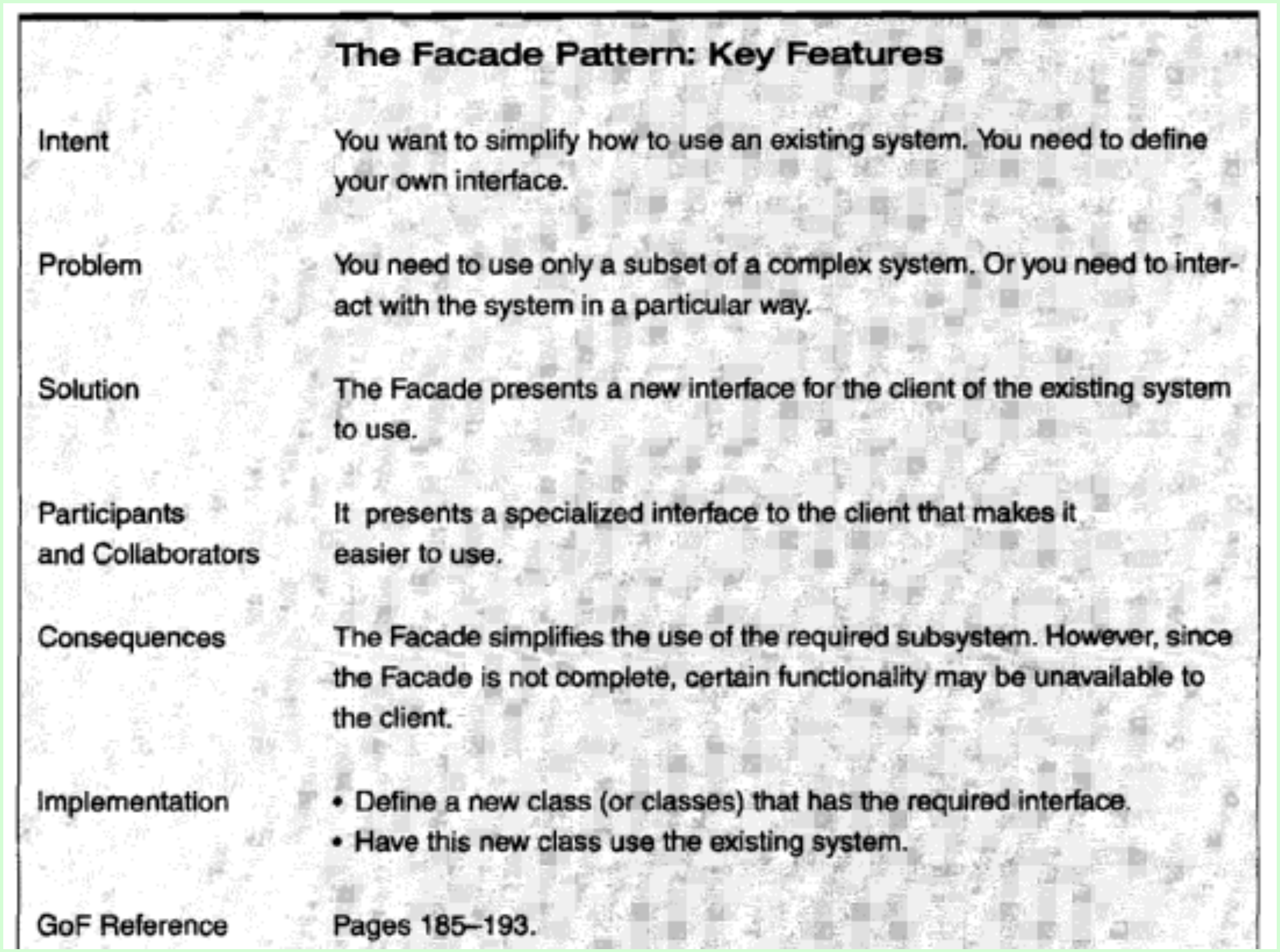


gure 6-2 Insulating clients from the subsystem.

Facade

- This approach only works when **using a subset of the system's capabilities** or when interacting with it in a particular way.
  If **everything** in the system needs to be used, it is **unlikely** that you can come up with a simpler interface.
- This is the **Facade pattern**.
  It enables us to **use a complex system more easily**, either to use just a subset of the system or use the system in a particular way.
  We have a **complicated system** of which we need to **use only a part**. We end up with a **simpler, easier-to-use system** or one that is customized to our needs.

Facade



	The Facade Pattern: Key Features
Intent	You want to simplify how to use an existing system. You need to define your own interface.
Problem	You need to use only a subset of a complex system. Or you need to interact with the system in a particular way.
Solution	The Facade presents a new interface for the client of the existing system to use.
Participants and Collaborators	It presents a specialized interface to the client that makes it easier to use.
Consequences	The Facade simplifies the use of the required subsystem. However, since the Facade is not complete, certain functionality may be unavailable to the client.
Implementation	• Define a new class (or classes) that has the required interface. • Have this new class use the existing system.
GoF Reference	Pages 185–193.

Facade - Participants

- **Facade**
  - knows which **subsystem classes** are **responsible** for a request.
  - **delegates client requests** to appropriate subsystem objects.

- **Subsystem classes**
  - **implement** subsystem functionality.
  - **handle work** assigned by the Facade object.
  - have **no knowledge** of the facade; that is, they keep no references to it.

Facade versus Adapter

- In both cases there are **preexisting class(es)** that contain the interface needed.
- In both cases a new object/class has a desired interface
- Both these patterns are wrappers...but they are different kinds of wrappers.

- **Facade**
  - **Not conforming** to an existing interface e.g. won't help to maintain polymorphic behaviour

- **Adapter**
  - Conforming to an existing interface
  - Not simplifying the interface but **trying to fit into existing one**

Facade Consequences

- It **shields clients** from subsystem components, thereby **reducing the number of objects** that clients deal with and making the subsystem easier to use.
- It **promotes weak coupling** between the subsystem and its clients. Weak coupling lets you vary the components of the subsystem without affecting its clients.
- It **doesn't prevent applications** from using subsystem classes if they need to. Thus you can choose between ease of use and generality.