I3305
Chapter 4
Design Patterns, Behavioral Patterns

Abed Safadi, Ph.D

Behavioral Patterns

General principles

- Enhance communication between objects :
  - Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects.
  - Define a communication model between objects
- Make the links between communicating objects more flexible
- ... and therefore increase the flexibility of the architecture

Plan

- Iterator
- Visitor
- Strategy
- Observer
- MVC
- Observer-MVC

Iterator pattern

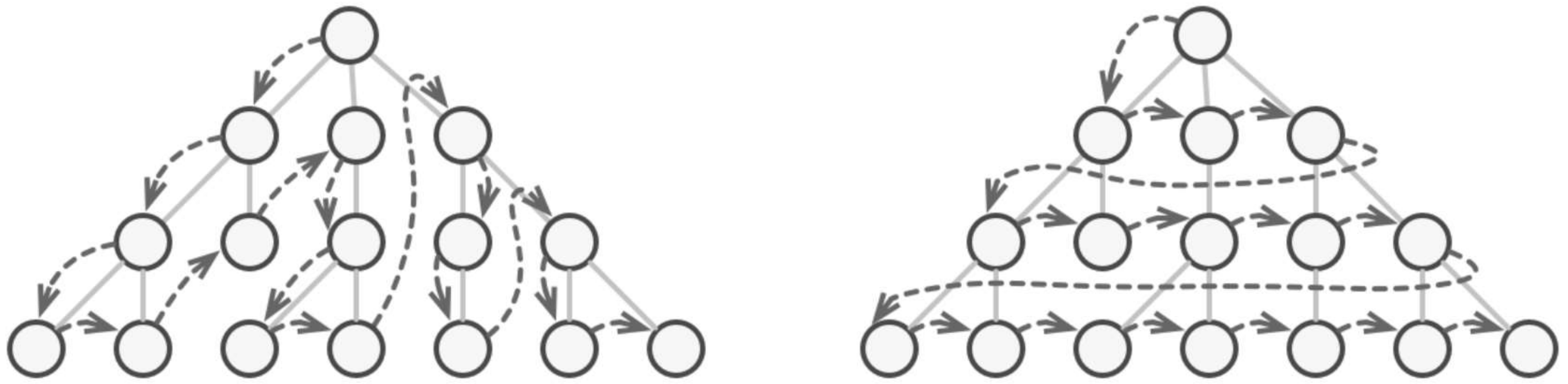
Intent:

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.). Iterator pattern is very commonly used design pattern in Java and .Net programming environment.

Example : Java collections iterators

Problem

Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.



The same collection can be traversed in several different ways.

If you have a collection based on a list. You just loop over all of the elements. But how do you sequentially traverse elements of a complex data structure, such as a tree?

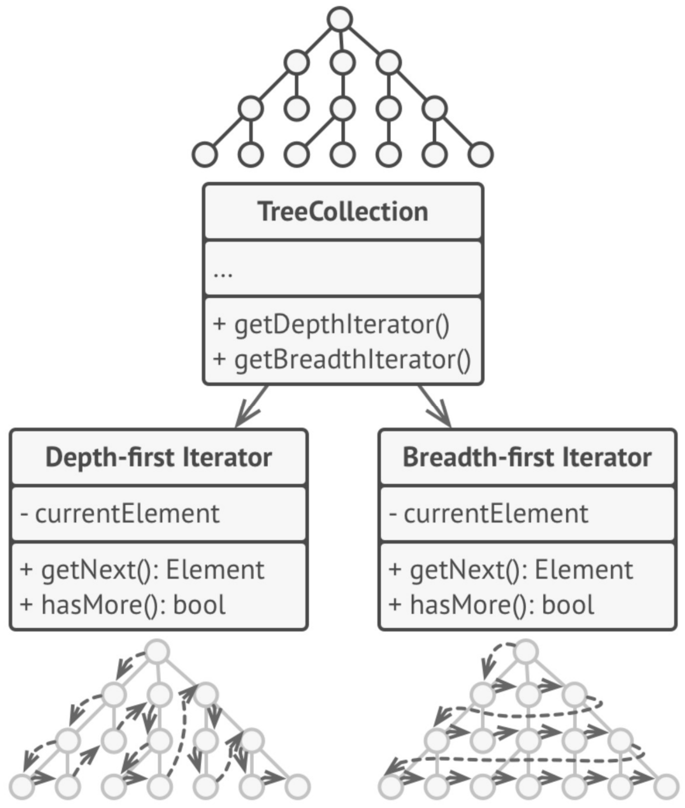
Solution

The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an *iterator*.

Usually, iterators provide one primary method for fetching elements of the collection.

All iterators must implement the same interface.

If you need a special way to traverse a collection, you just create a new iterator class, without having to change the collection or the client.



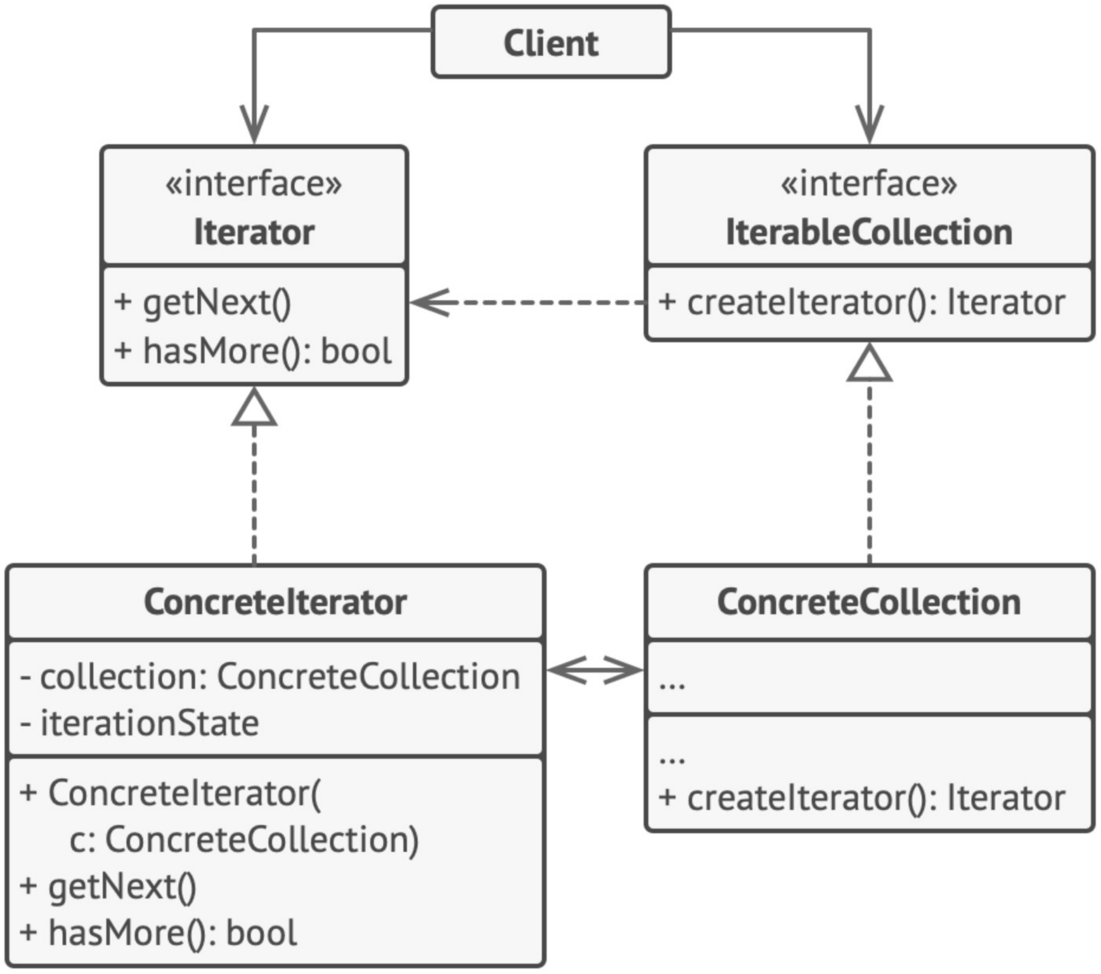
Iterators implement various traversal algorithms. Several iterator objects ca traverse the same collection at the same time.

Structure

The **Iterator** interface declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.

Concrete Iterators implement specific algorithms for traversing a collection.

The **Collection** interface declares one or multiple methods for getting iterators compatible with the collection. Note that the return type of the methods must be declared as the iterator interface so that the concrete collections can return various kinds of iterators.



Concrete Collections return new instances of a particular concrete iterator class each time the client requests one

Iterator pattern example

```java
public class Entreprise {
    private String nom;
    private List<Personne> employes;
    public Entreprise(String nom) {
        this.nom = nom; employes = new LinkedList<Personne>();
    }

    public void embaucher(Personne p){ employes.add(p); }

    public Iterateur creerIterateur(){
        return new IterateurDePersonnes(employes);
    }
}
```

```java
public class Personne {
    private String nom, prenom;
    public Personne(String nom, String prenom) {
        this.nom = nom; this.prenom = prenom;
    }

    public String toString() {
        return "Personne [nom=" + nom + "," + "prenom=" + prenom + "]";
    }
}
```

```java
public class Client {
    public static void main(String[] args) {
        Entreprise maBoite = new Entreprise("Poire Mordue");
        maBoite.embaucher(new Personne("Laporte", "Marc"););
        maBoite.embaucher(new Personne("Pain-Barre", "Cyril"));
        maBoite.embaucher(new Personne("Valicov", "Petru"));

        Iterateur it = maBoite.creerIterateur();
        for (it.premier(); !it.cestFini(); it.suivant()) {
            System.out.println(it.courant());
        }
    }
}
```

```java
public interface Iterateur {
    public Personne premier();
    public void suivant();
    public boolean cestFini();
    public Personne courant();
}
```

```java
public class IterateurDePersonnes implements Iterateur {
    private List<Personne> lesEmployes;
    private int position;

    public IterateurDePersonnes(List<Personne> employes) {
        lesEmployes = employes;
    }

    public void suivant() {
      position++;
    }

    public Personne premier() {
        position = 0;
        return lesEmployes.get(0);
    }

    public boolean cestFini() {
        if (position >= lesEmployes.size())
            return true;
        return false;
    }

    public Personne courant() {
        return lesEmployes.get(position);
    }
}
```

Plan

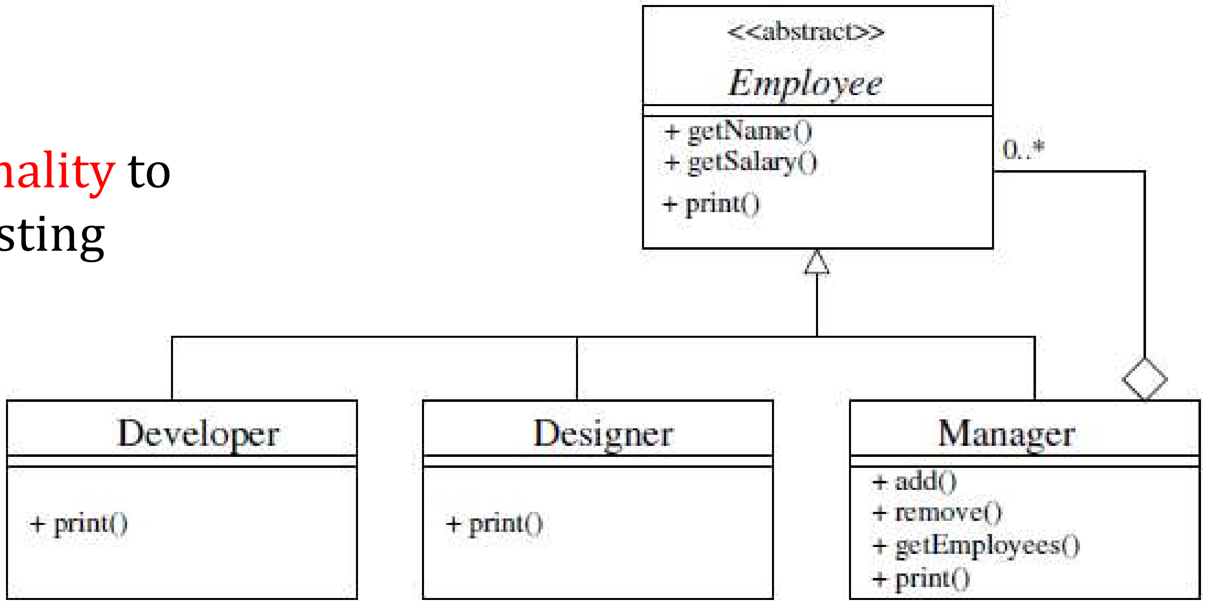
- Iterator
- **Visitor**
- Strategy
- Observer
- MVC
- Observer-MVC

Visitor

The purpose of a Visitor pattern is to define a new operation without introducing the modifications to an existing object structure.

Imagine that we have a composite object which consists of components. The object's structure is fixed – we either can't change it, or we don't plan to add new types of elements to the structure.

Now, how could we add new functionality to our code without modification of existing classes?

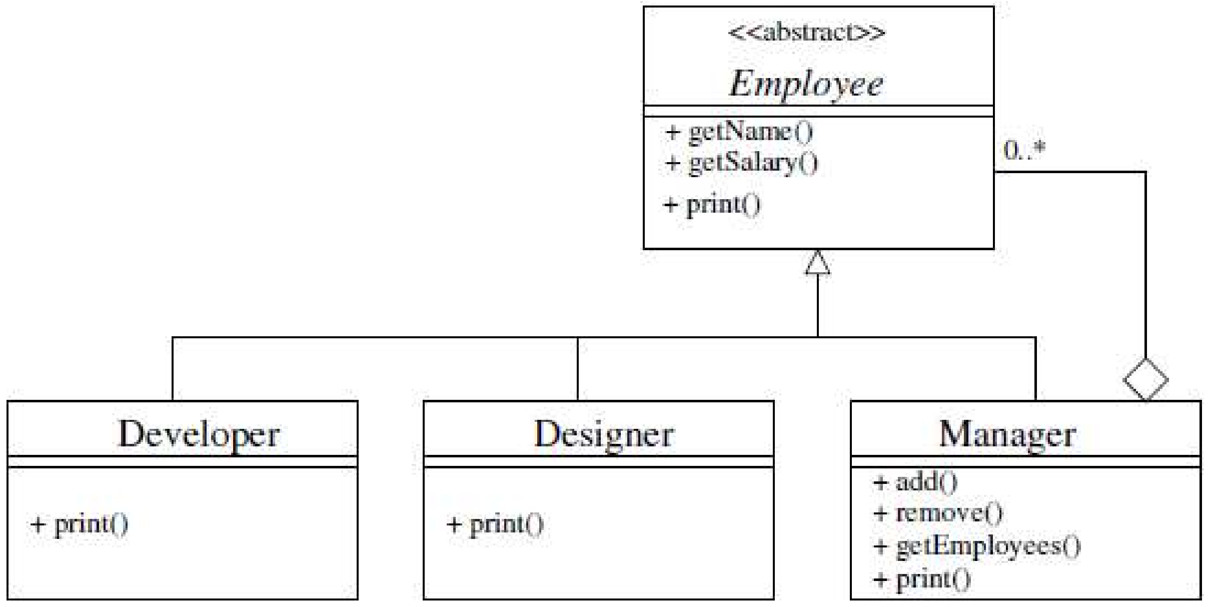


Visitor Pattern Problem

Example (Employee)

1. Print the names and their corresponding salaries of all the employees (except managers) of a given manager.

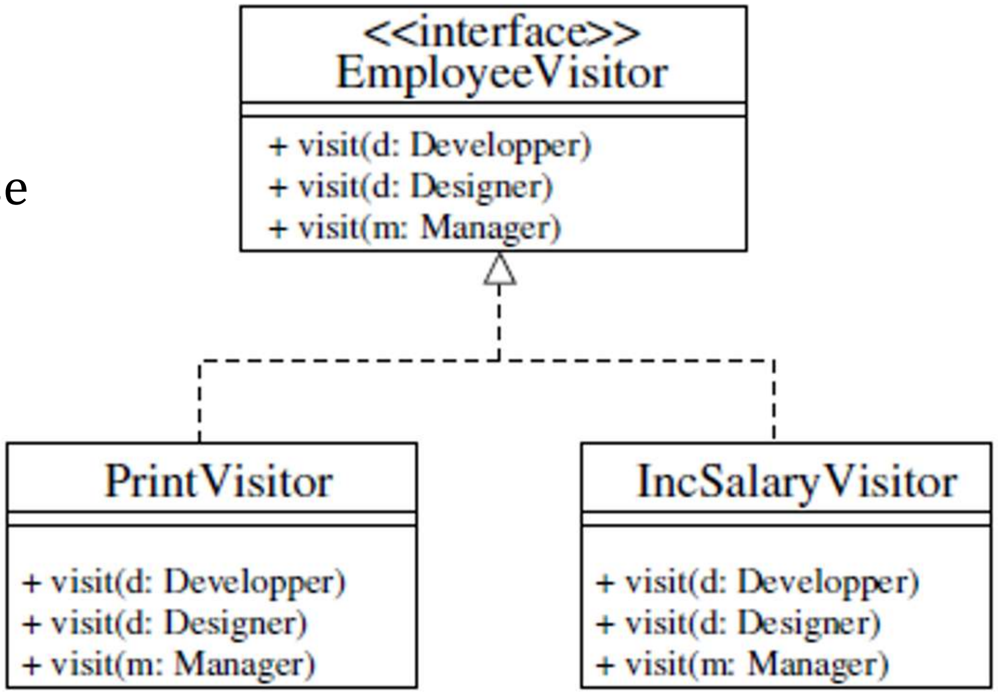
2. Increase the salary of all the employees of a given manager (included).



Solution

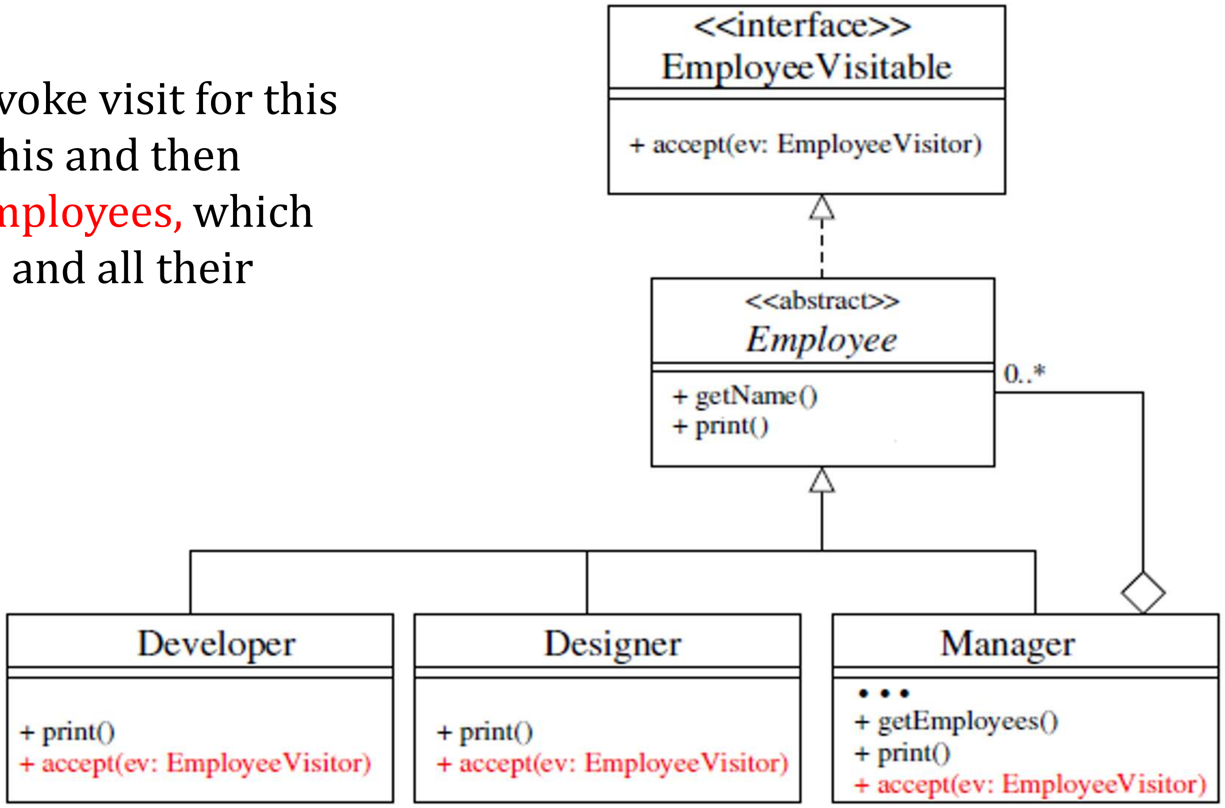
The Visitor pattern suggests that you place the new behavior into a separate class called *visitor*, instead of trying to integrate it into existing classes.

- EmployeeVisitor interface. Declares visit methods.
- Different implementations of EmployeeVisitor give you different visit methods
  - PrintVisitor: visit methods print name and salary
  - IncSalaryVisitor: visit methods increase salary



Solution

- EmployeeVisitable interface. Declares accept method.
- Our base class, Employee, implements EmployeeVisitable
- accept takes an EmployeeVisitor object as argument
- Implementations of accept invoke EmployeeVisitor.visit() as appropriate
  - accept in Developer and Designer invoke visit for this
  - accept in Manager invokes visit for this and then invokes accept for all subordinate employees, which will cause visit to be called for them, and all their subordinates, etc.



Java Translation

```java
public interface EmployeeVisitable {
    public void accept(EmployeeVisitor employeeVisitor);
}
```

```java
public abstract class Employee implements EmployeeVisitable {
    protected String name;
    protected int salary;

    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public int getSalary() {
        return salary;
    }
    public abstract void print();
}
```

Java Translation

```java
public class Developer extends Employee {
    public Developer(String name, int salary) {
        super(name, salary);
    }

    public void print() {
        System.out.println("Developer " + name);
    }

    public void accept(EmployeeVisitor employeeVisitor) {
        employeeVisitor.visit(this);
    }
}
```

```java
public class Designer extends Employee {
    public Designer(String name, int salary) {
        super(name, salary);
    }

    public void print() {
        System.out.println("Designer " + name);
    }

    public void accept(EmployeeVisitor employeeVisitor) {
        employeeVisitor.visit(this);
    }
}
```

Java Translation

```java
public class Manager extends Employee {
    protected LinkedList<Employee> employees = new LinkedList<Employee>();

    public Manager(String name, int salary) {
        super(name, salary);
    }

    ...

    public void accept(EmployeeVisitor employeeVisitor) {
        employeeVisitor.visit(this);
        for(Employee e: employees) {
            e.accept(employeeVisitor);
        }
    }
}
```

Java Translation

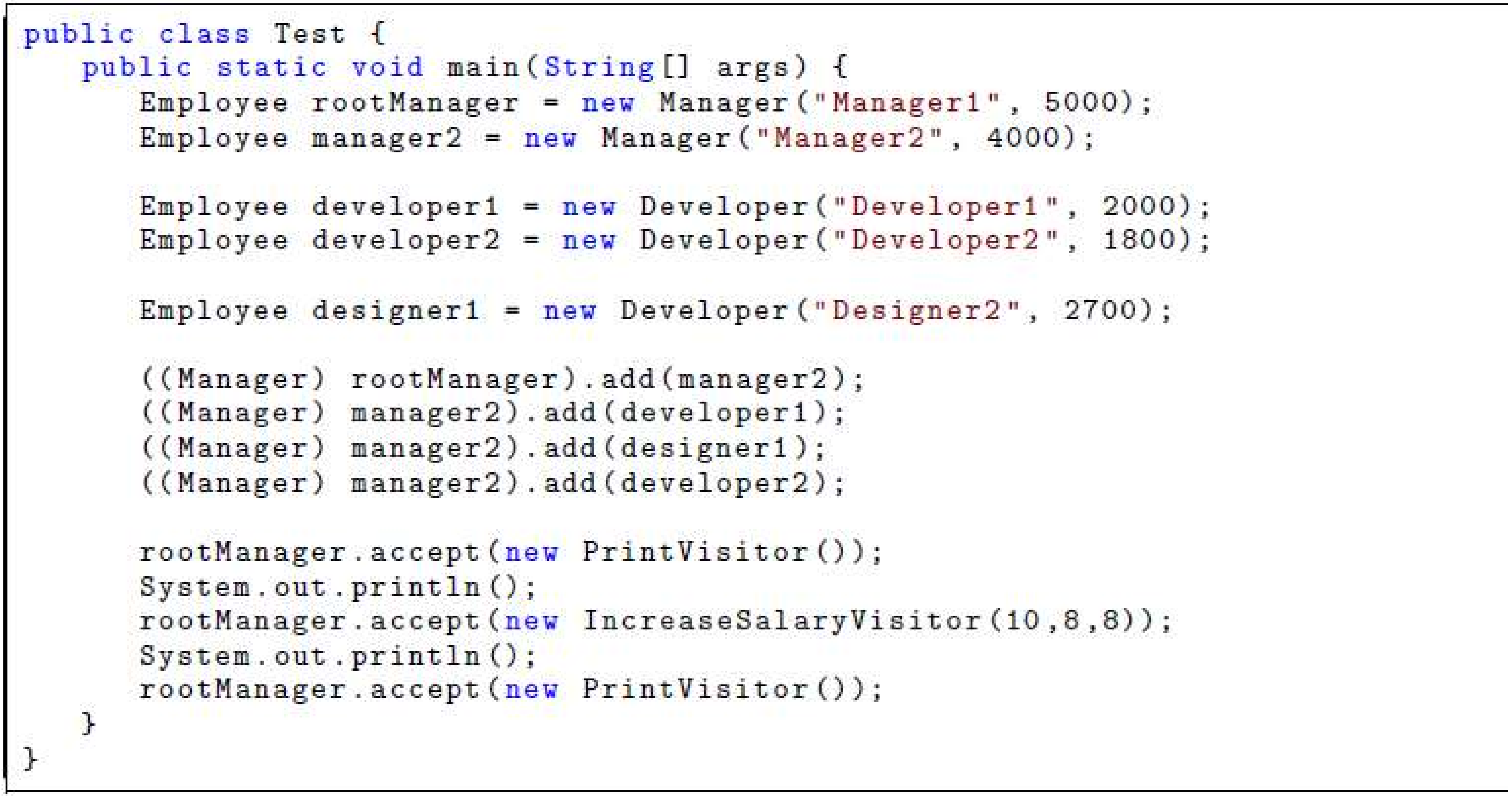
```
public interface EmployeeVisitor {
    public void visit(Manager manager);
    public void visit(Developer developer);
    public void visit(Designer designer);
}
```

```
public class PrintVisitor implements EmployeeVisitor{
    public void visit(Manager manager) {
        System.out.println(manager.getName() + " " + manager.getSalary());
    }

    public void visit(Developer developer) {
        System.out.println(developer.getName() + " " + developer.getSalary());
    }

    public void visit(Designer designer) {
        System.out.println(designer.getName() + " " + designer.getSalary());
    }
}
```

Java Translation

```java
public class IncreaseSalaryVisitor implements EmployeeVisitor {
   private int percentageManager;
   private int percentageDesigner;
   private int percentageDeveloper;

   public IncreaseSalaryVisitor(int pManager, int pDesigner, int pDevelper) {
      percentageManager = pManager;
      percentageDesigner = pDesigner;
      percentageDeveloper = pDevelper;
   }

   public void visit(Manager manager) {
      manager.salary *= 1 + percentageManager/100.0;
   }

   public void visit(Developer developer) {
      developer.salary *= 1 + percentageDeveloper/100.0;
   }

   public void visit(Designer designer) {
      designer.salary *= 1 + percentageDesigner/100.0;
   }
}
```

Java Translation

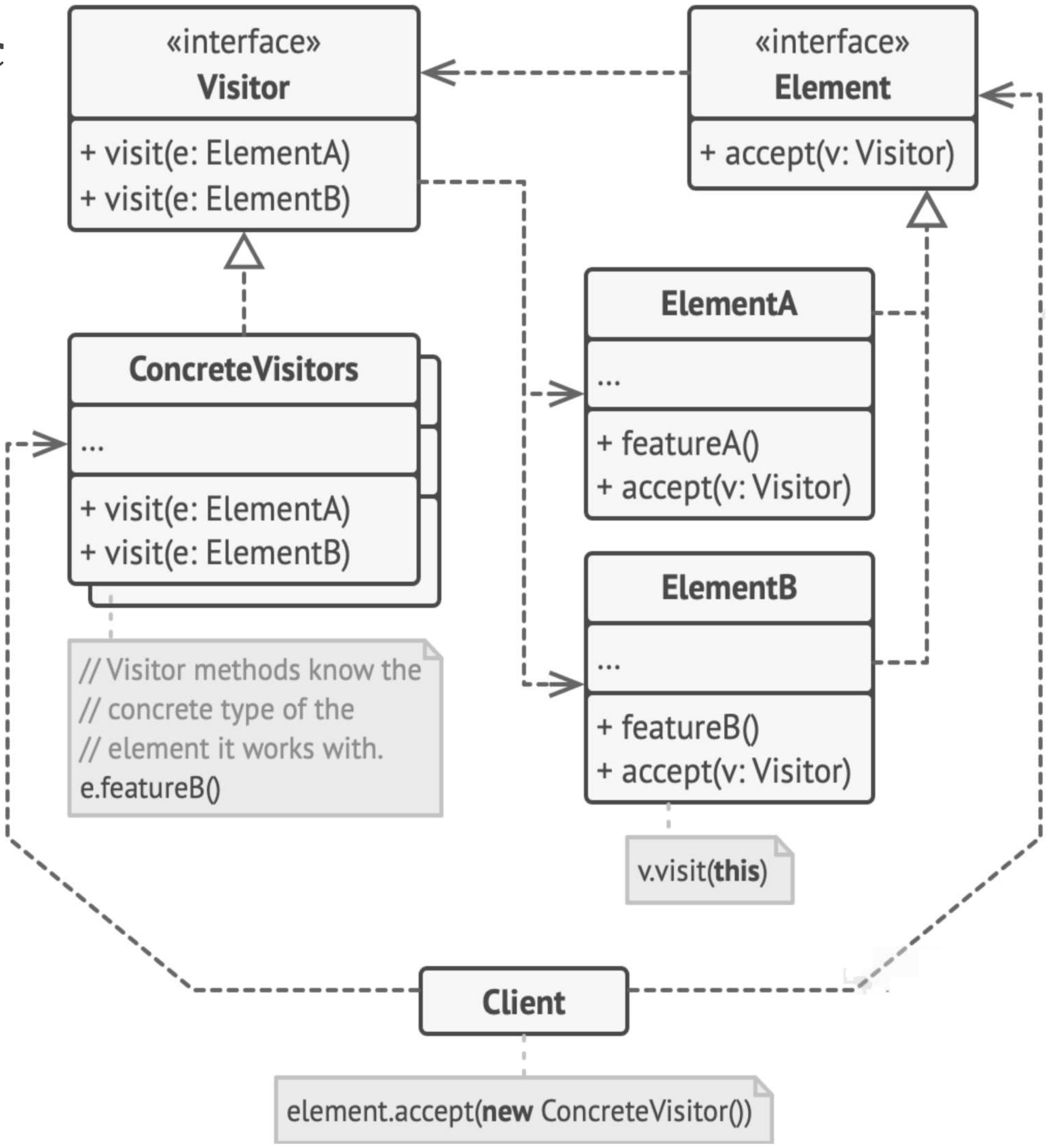


```java
public class Test {
    public static void main(String[] args) {
        Employee rootManager = new Manager("Manager1", 5000);
        Employee manager2 = new Manager("Manager2", 4000);

        Employee developer1 = new Developer("Developer1", 2000);
        Employee developer2 = new Developer("Developer2", 1800);

        Employee designer1 = new Developer("Designer2", 2700);

        ((Manager) rootManager).add(manager2);
        ((Manager) manager2).add(developer1);
        ((Manager) manager2).add(designer1);
        ((Manager) manager2).add(developer2);

        rootManager.accept(new PrintVisitor());
        System.out.println();
        rootManager.accept(new IncreaseSalaryVisitor(10,8,8));
        System.out.println();
        rootManager.accept(new PrintVisitor());
    }
}
```

Structure

The **Visitor** interfac e declares a set of visiting methods that can take concrete elements of an object structure as arguments.

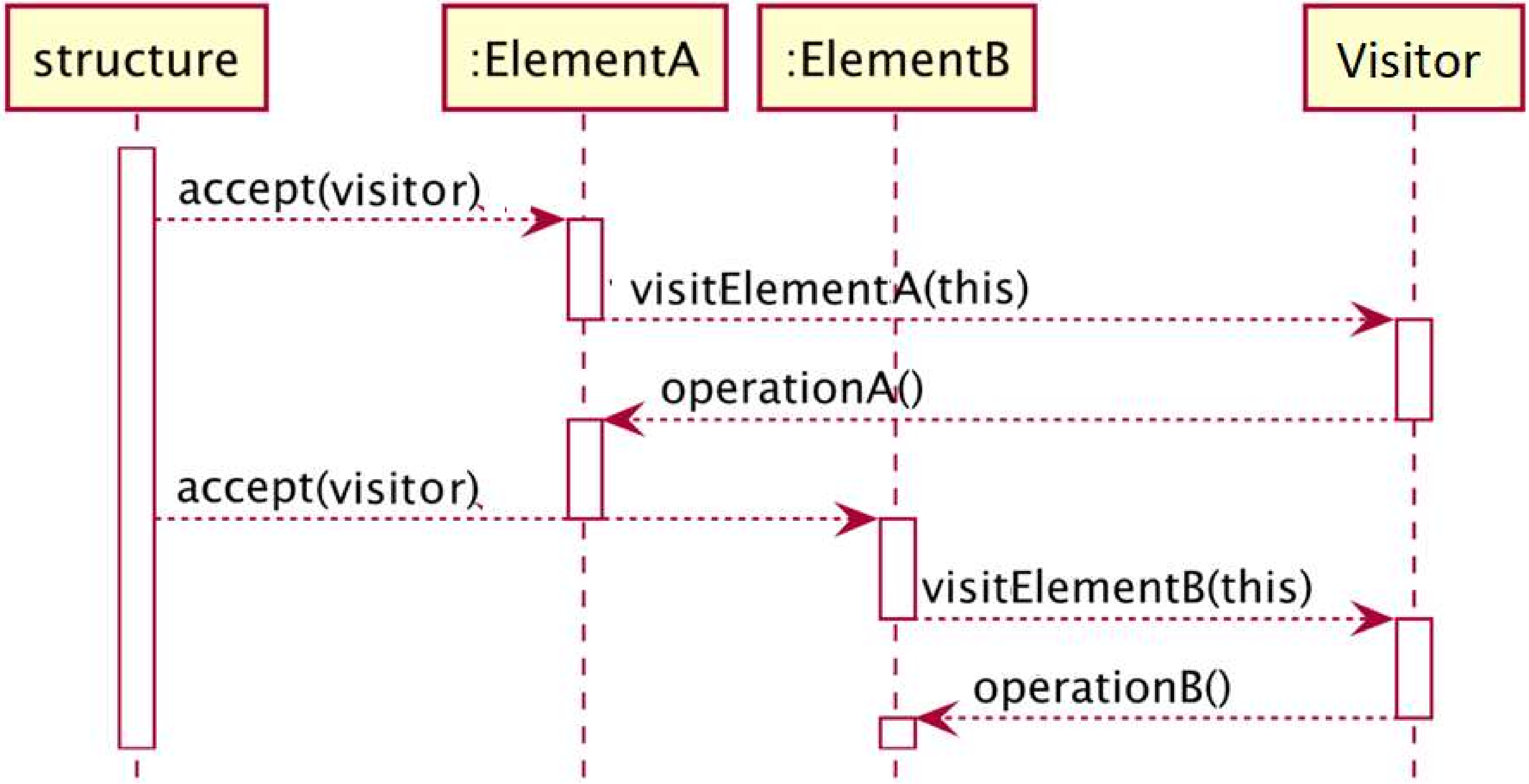
Each **Concrete Visitor** implemen ts several versions of the same behaviors, tailored for different concrete element classes.



The **Element** interfa ce declares a method for “accepting” visitors.

Each **Concrete Element** must implement the acceptance method. The purpose of this method is to redirect the call to the proper visitor’s method corresponding to the current element class.

Sequence Diagram



Plan

- Iterator
- Visitor
- Strategy
- Observer
- MVC
- Observer-MVC

Strategy

Intent

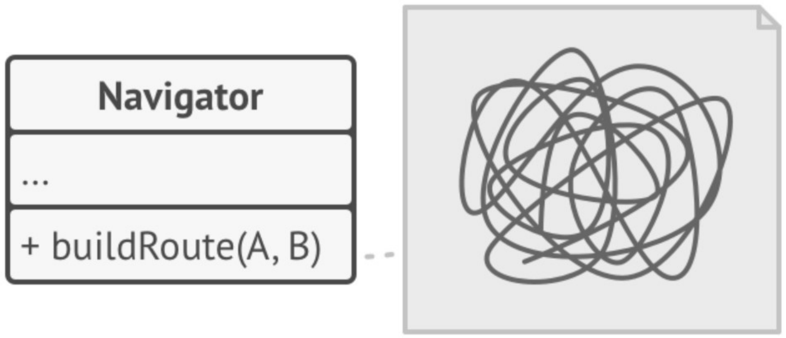
Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Problem

One day you decided to create a navigation app for casual travelers. The app was centered around a beautiful map which helped users quickly orient themselves in any city.

The first version of the app could only build the routes over roads. People who traveled by car were bursting with joy. But apparently, not everybody likes to drive on their vacation. So with the next update, you added an option to build walking routes. Right after that, you added another option to let people use public transport in their routes.

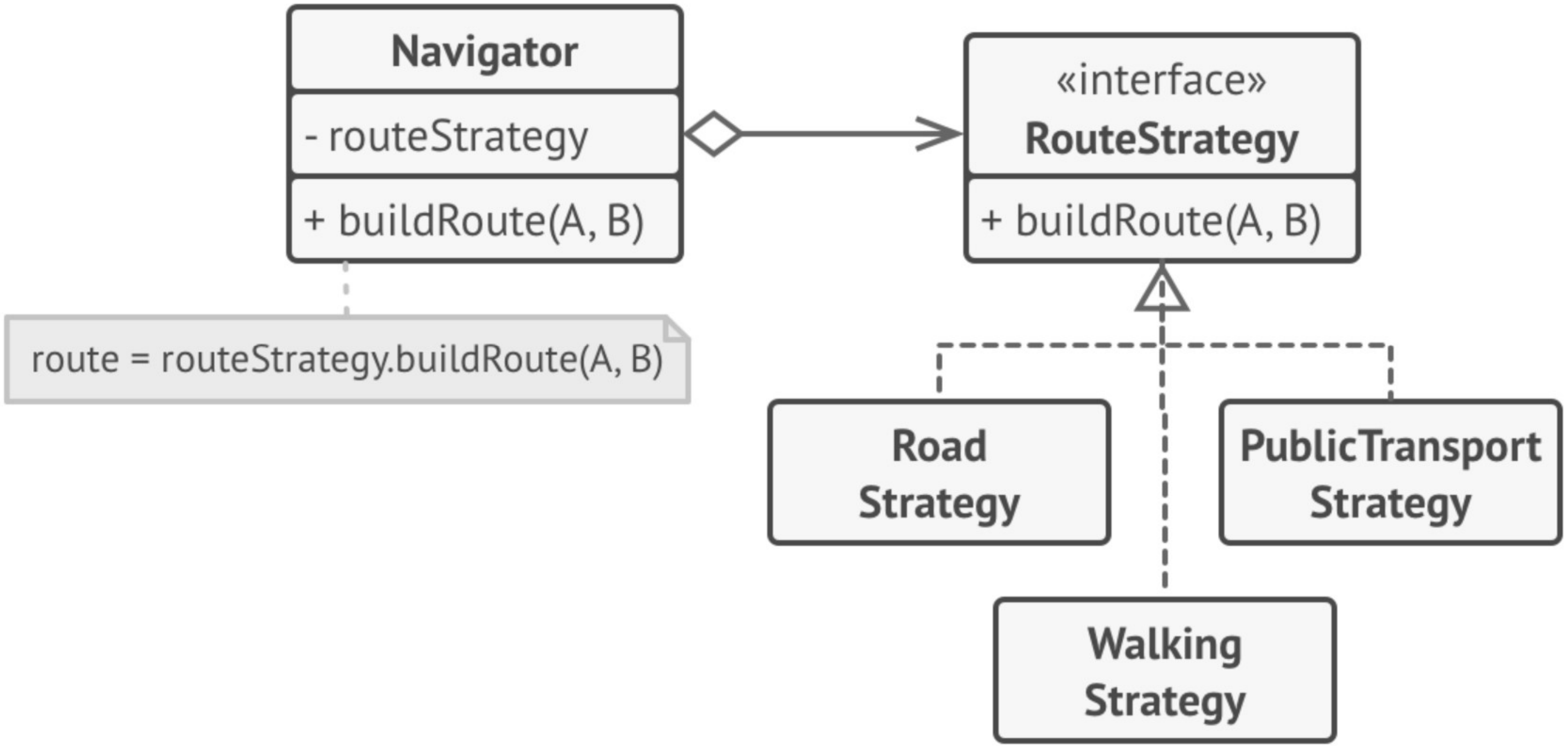
Each time you added a new routing algorithm, the main class of the navigator doubled in size. At some point, the beast became too hard to maintain.



Any change to one of the algorithms, whether it was a simple bug fix or a slight adjustment of the street score, affected the whole class, increasing the chance of creating an error in already-working code

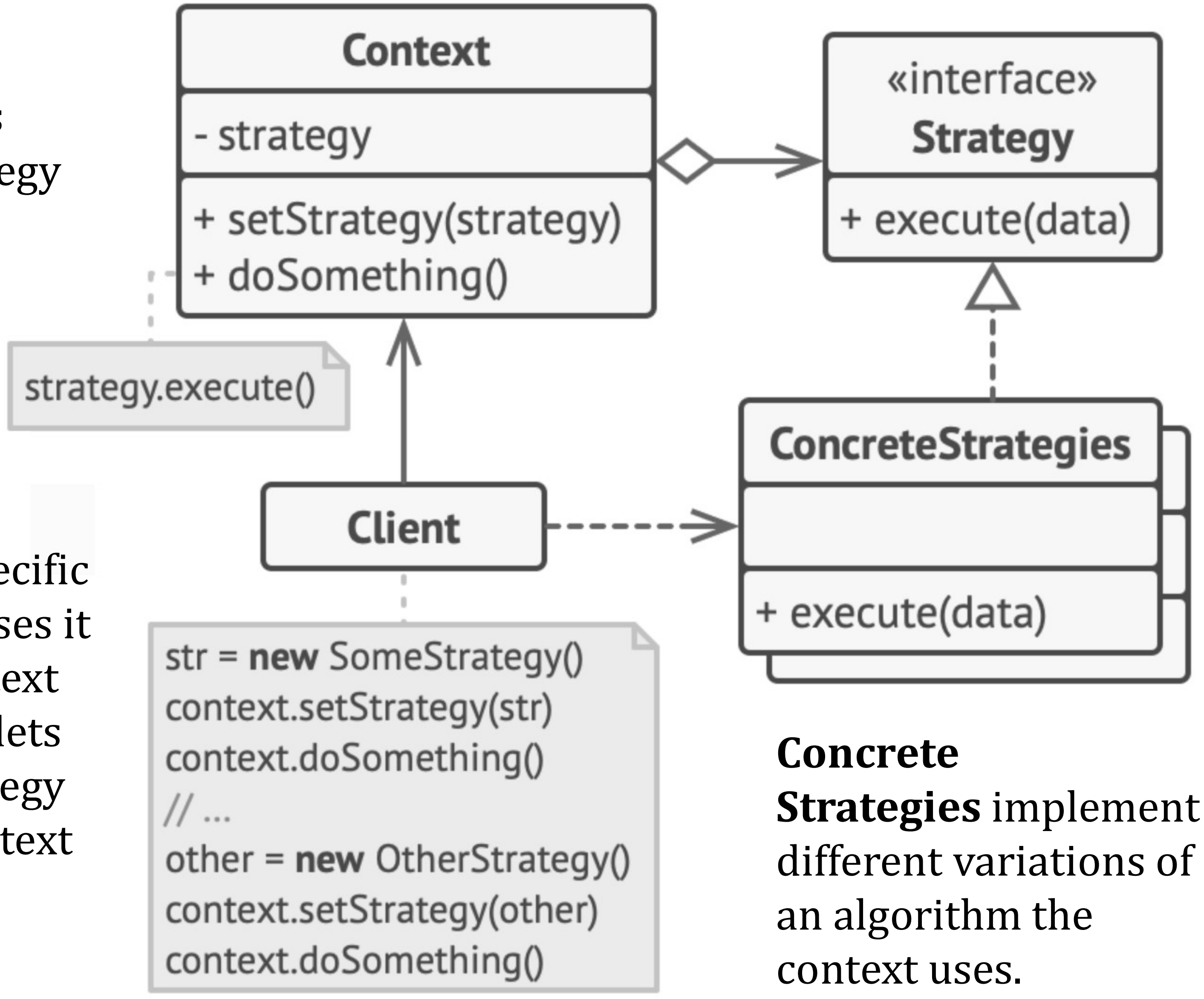
Solution

The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called *strategies*.



Structure

The **Context** maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface.

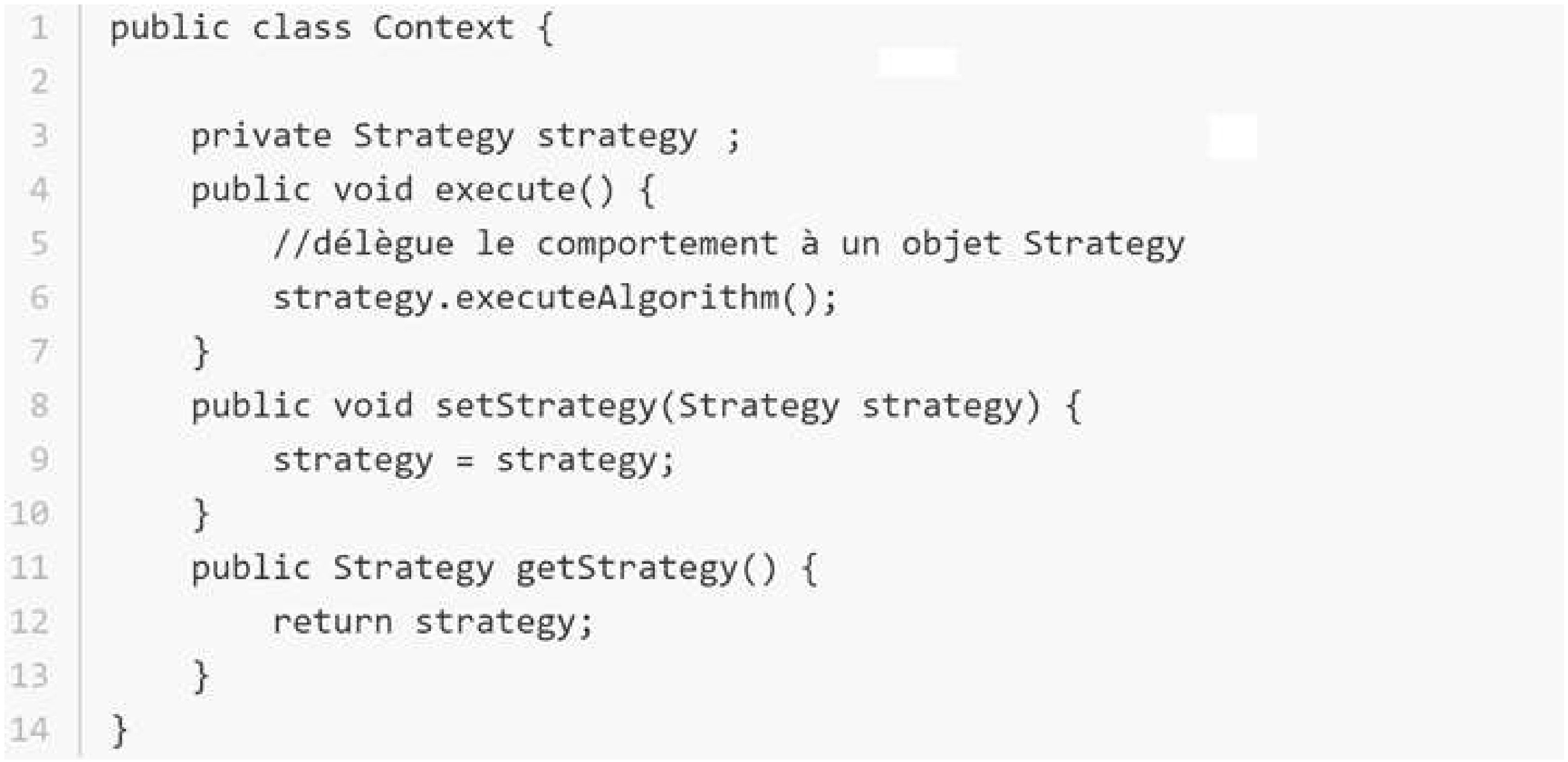


```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

The **Client** creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.

Concrete Strategies implement different variations of an algorithm the context uses.

Java Translation

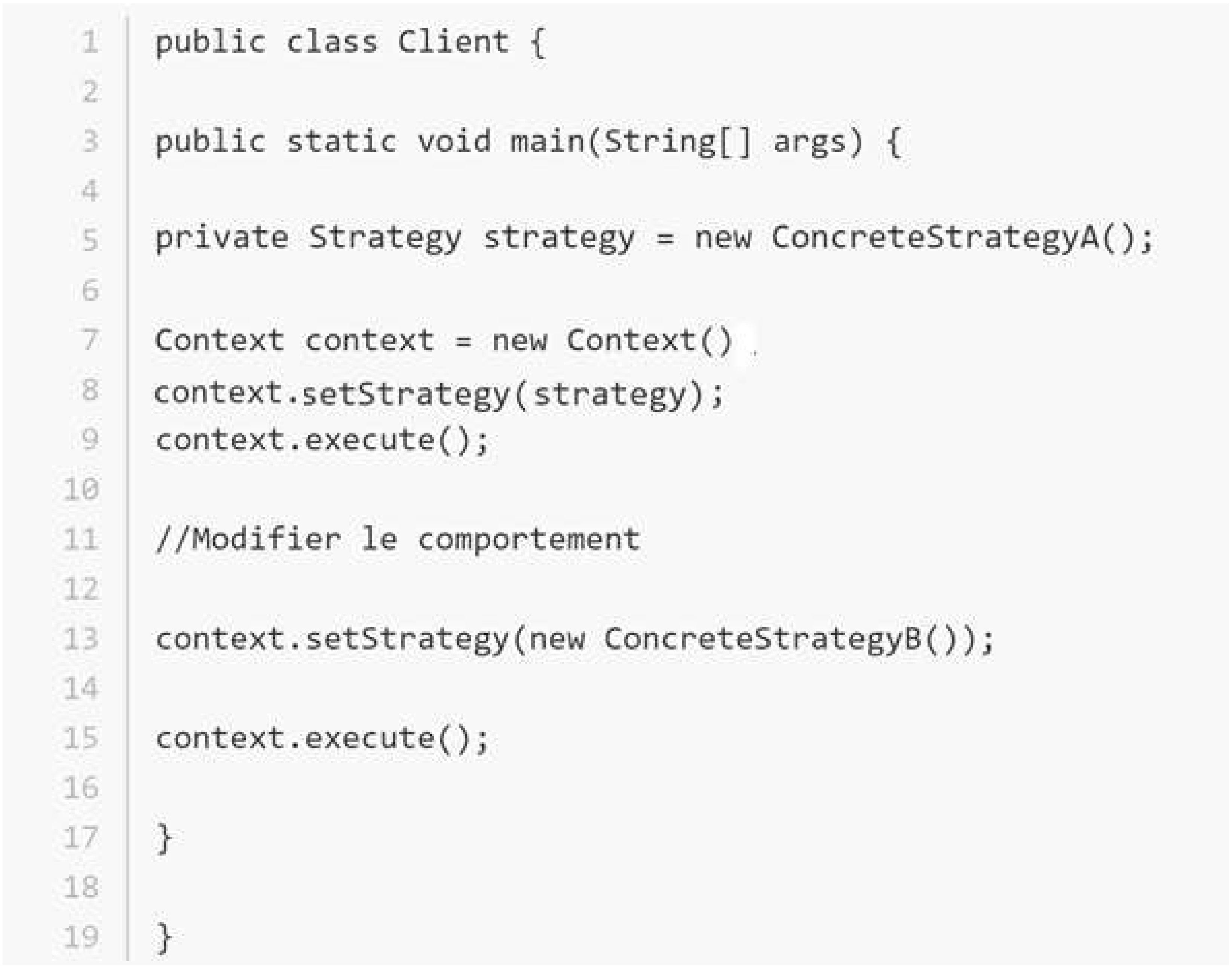


```java
public class Context {

    private Strategy strategy ;
    public void execute() {
        //délègue le comportement à un objet Strategy
        strategy.executeAlgorithm();
    }
    public void setStrategy(Strategy strategy) {
        strategy = strategy;
    }
    public Strategy getStrategy() {
        return strategy;
    }
}
```

Java Translation

```java
interface Strategy {
    public void executeAlgorithm();
}
class ConcreteStrategyA implements Strategy {
    public void executeAlgorithm() {
        System.out.println("Concrete Strategy A");
    }
}
class ConcreteStrategyB implements Strategy {
    public void executeAlgorithm() {
        System.out.println("Concrete Strategy B");
    }
}
```

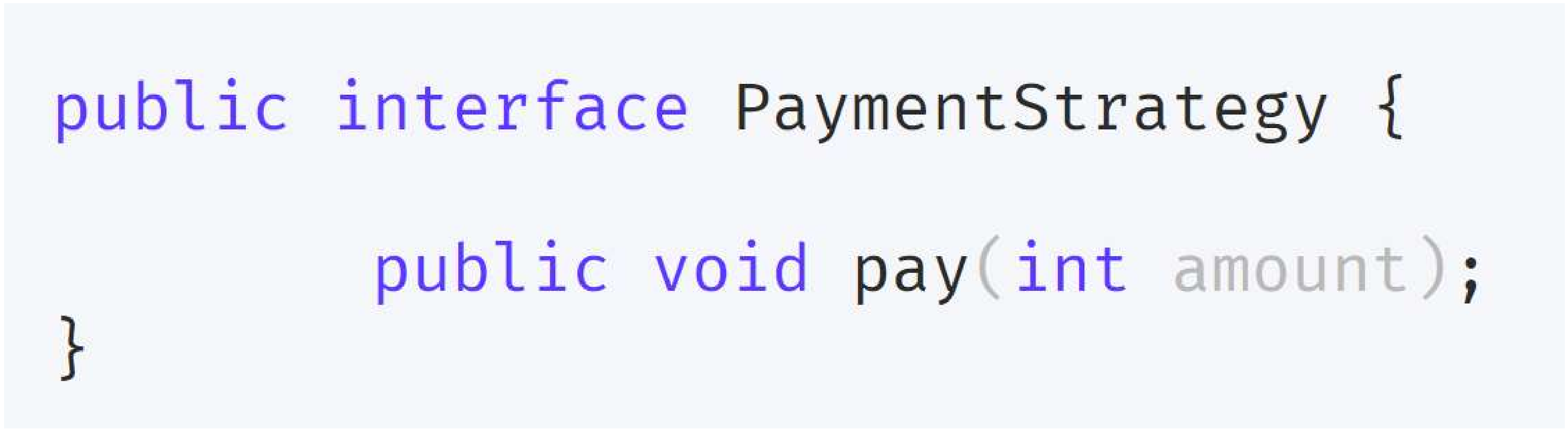
Java Translation



```java
public class Client {

public static void main(String[] args) {

private Strategy strategy = new ConcreteStrategyA();

Context context = new Context()
context.setStrategy(strategy);
context.execute();

//Modifier le comportement

context.setStrategy(new ConcreteStrategyB());

context.execute();


}

}
```

Strategy pattern Example

we will try to implement a simple Shopping Cart where we have two payment strategies – using Credit Card or using PayPal.



```
public interface PaymentStrategy {

        public void pay(int amount);
}
```

Java Translation

```
public class CreditCardStrategy implements
PaymentStrategy {

        private String name;
        private String cardNumber;
        private String cvv;
        private String dateOfExpiry;

        public CreditCardStrategy(String nm, String
ccNum, String cvv, String expiryDate){
                this.name=nm;
                this.cardNumber=ccNum;
                this.cvv=cvv;
                this.dateOfExpiry=expiryDate;
        }
        @Override
        public void pay(int amount) {
                System.out.println(amount +" paid with
credit/debit card");
        }

}
```

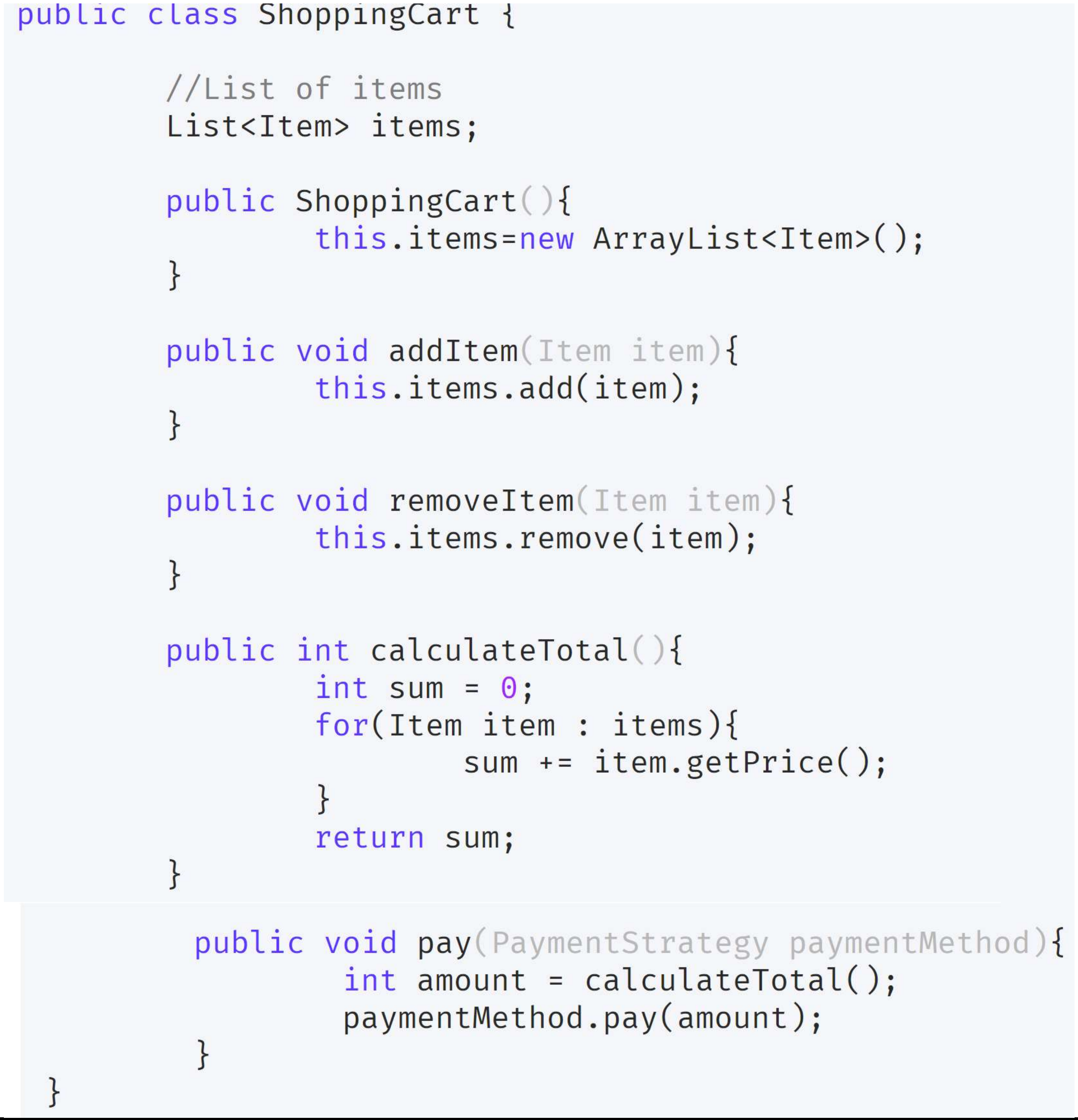
Java Translation

```
public class PaypalStrategy implements PaymentStrategy {

        private String emailId;
        private String password;

        public PaypalStrategy(String email, String pwd){
                this.emailId=email;
                this.password=pwd;
        }

        @Override
        public void pay(int amount) {
                System.out.println(amount + " paid using
Paypal.");
        }

}
```

Java Translation

```java
public class Item {

        private String upcCode;
        private int price;

        public Item(String upc, int cost){
                this.upcCode=upc;
                this.price=cost;
        }

        public String getUpcCode() {
                return upcCode;
        }

        public int getPrice() {
                return price;
        }

}
```

Java Translation



```java
public class ShoppingCart {

        //List of items
        List<Item> items;

        public ShoppingCart(){
                this.items=new ArrayList<Item>();
        }

        public void addItem(Item item){
                this.items.add(item);
        }

        public void removeItem(Item item){
                this.items.remove(item);
        }

        public int calculateTotal(){
                int sum = 0;
                for(Item item : items){
                        sum += item.getPrice();
                }
                return sum;
        }

        public void pay(PaymentStrategy paymentMethod){
                int amount = calculateTotal();
                paymentMethod.pay(amount);
        }
}
```

Java Translation

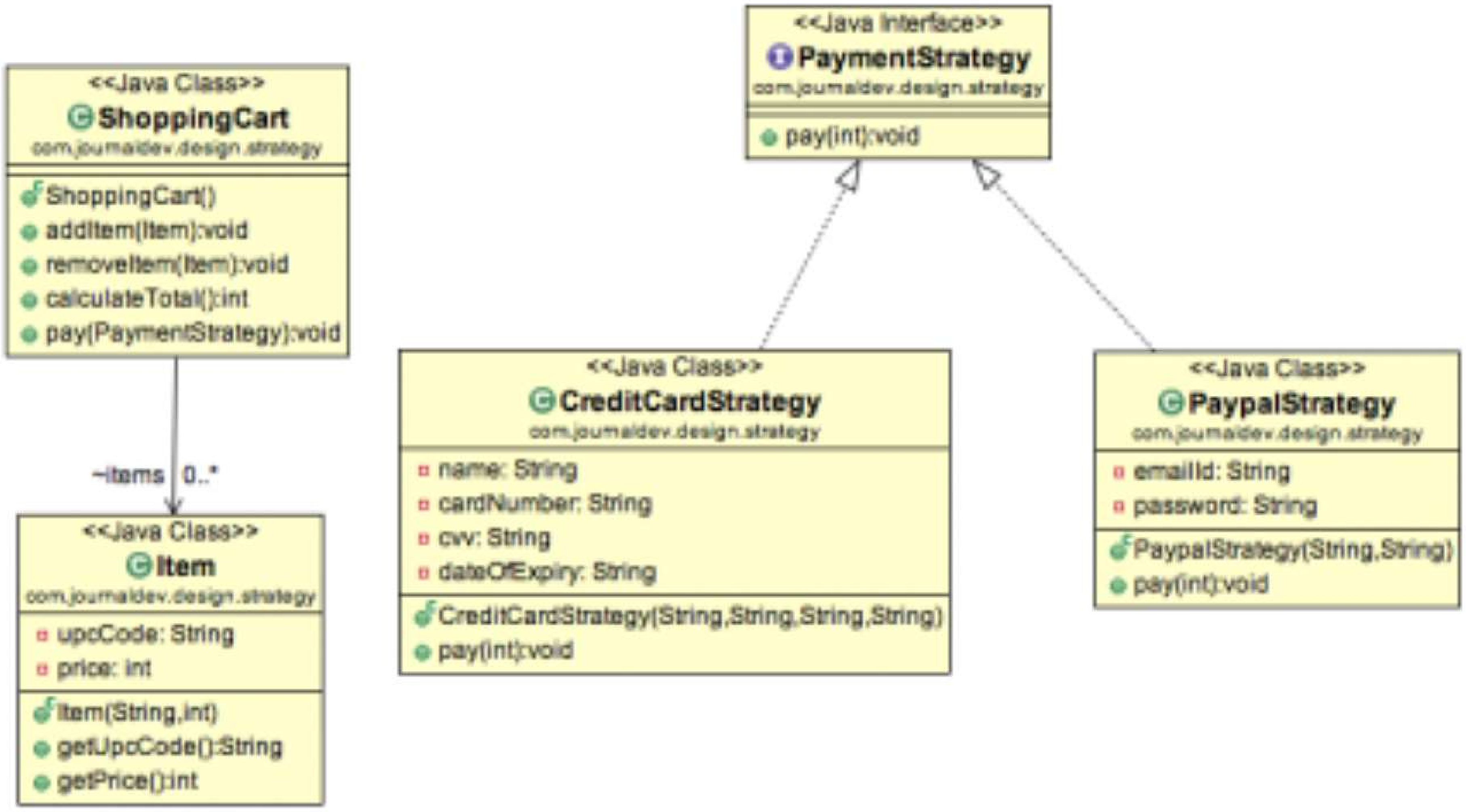
```
public class ShoppingCartTest {

        public static void main(String[] args) {
                ShoppingCart cart = new ShoppingCart();

                Item item1 = new Item("1234",10);
                Item item2 = new Item("5678",40);

                cart.addItem(item1);
                cart.addItem(item2);

                //pay by paypal
                cart.pay(new
PaypalStrategy("myemail@example.com", "mypwd"));

                //pay by credit card
                cart.pay(new CreditCardStrategy("Pankaj
Kumar", "1234567890123456", "786", "12/15"));
        }

}
```

Java Translation

Output of program is:

```
50 paid using Paypal.
50 paid with credit/debit card
```

Strategy design pattern class diagram



Plan

- Iterator
- Visitor
- Strategy
- Observer
- MVC
- Observer-MVC

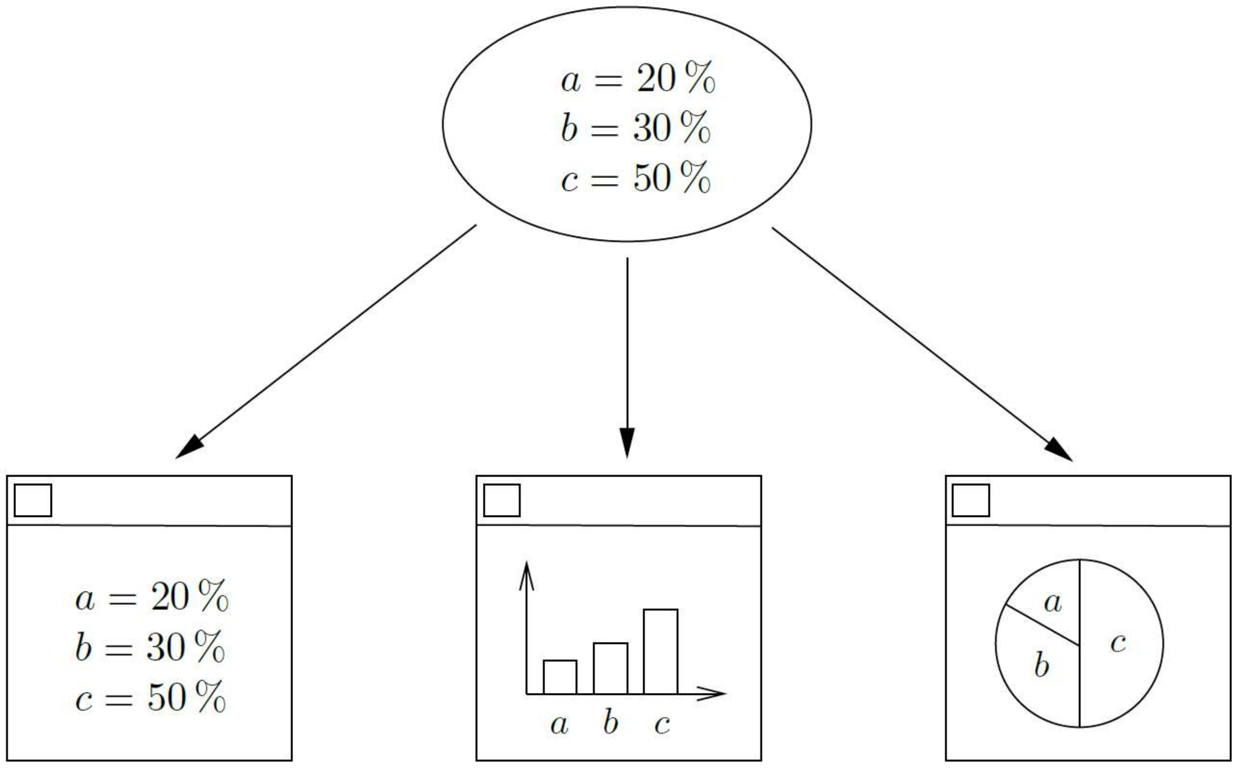
Observer design pattern

Intent

- **Observer** is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Problem

Imagine creating an application that can have several graphical representations for the same object.

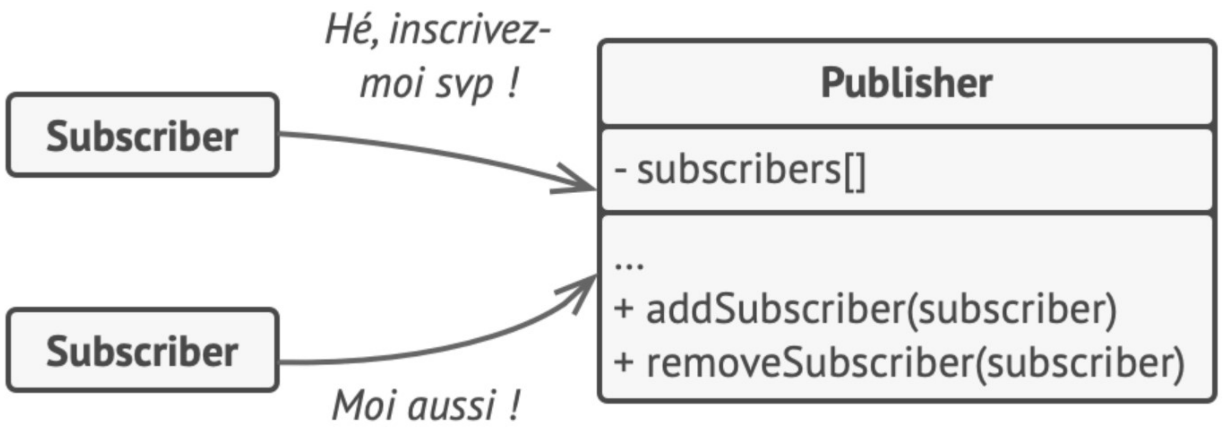


How to notify graphical representations of changes?

Solution

- The object that has some interesting state is often called *subject*, but since it's also going to notify other objects about the changes to its state, we'll call it *publisher*.

- All other objects that want to track changes to the publisher's state are called *subscribers*.



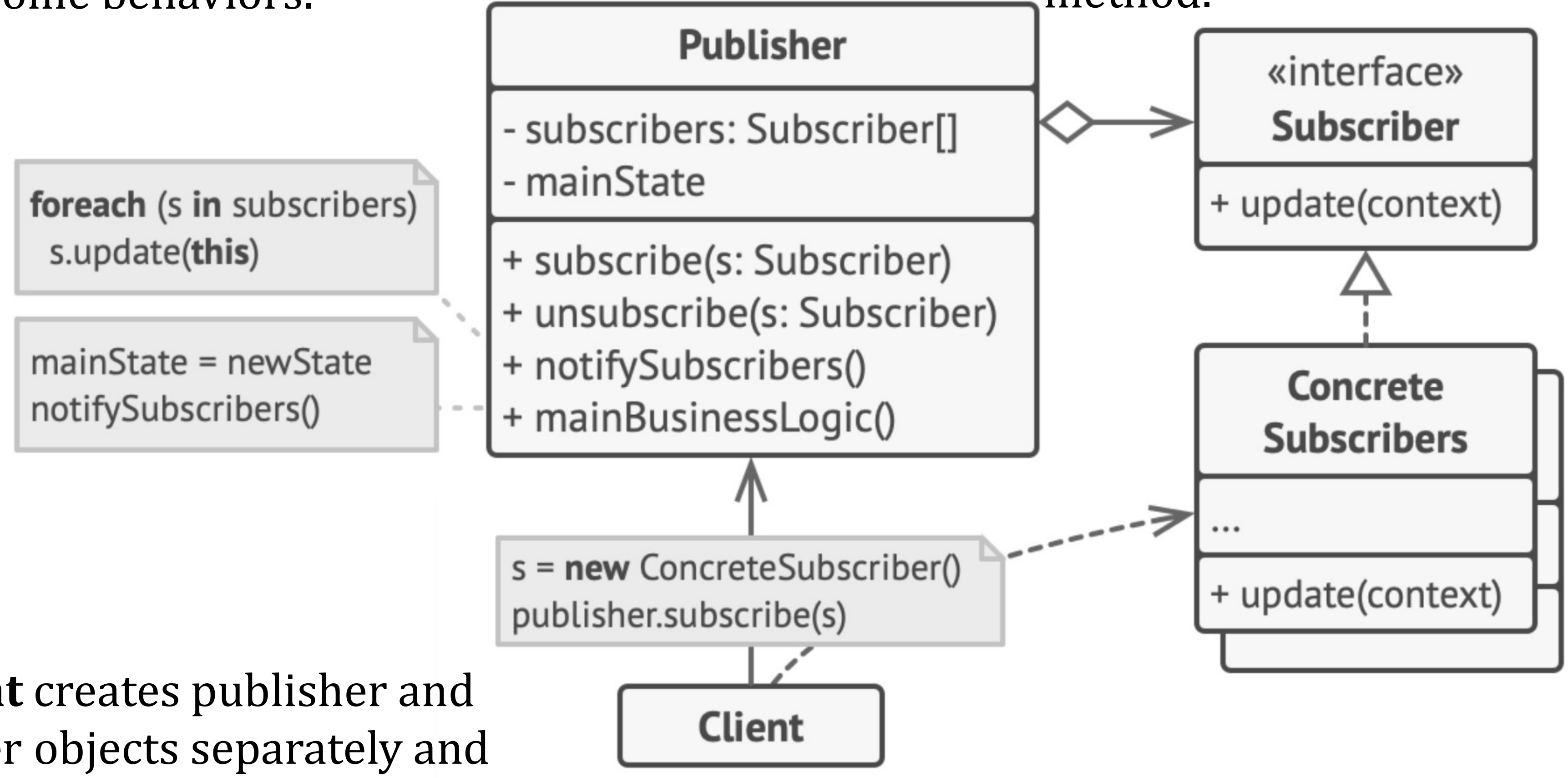
- The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher.

Structure

The **Publisher** issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors.

When a new event happens, the publisher goes over the subscription list and calls the notification method declared in the subscriber interface on each subscriber object.

The **Subscriber** interface declares the notification interface. In most cases, it consists of a single update method.

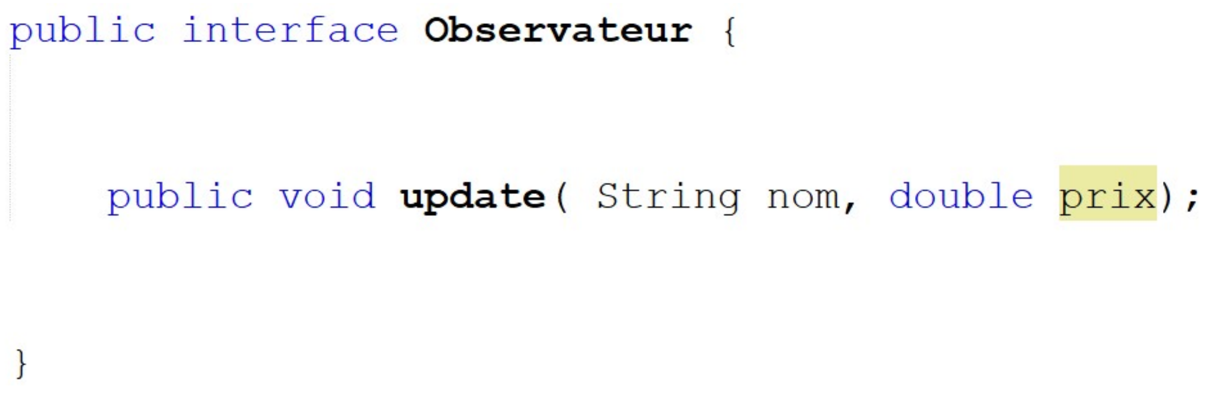


The **Client** creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

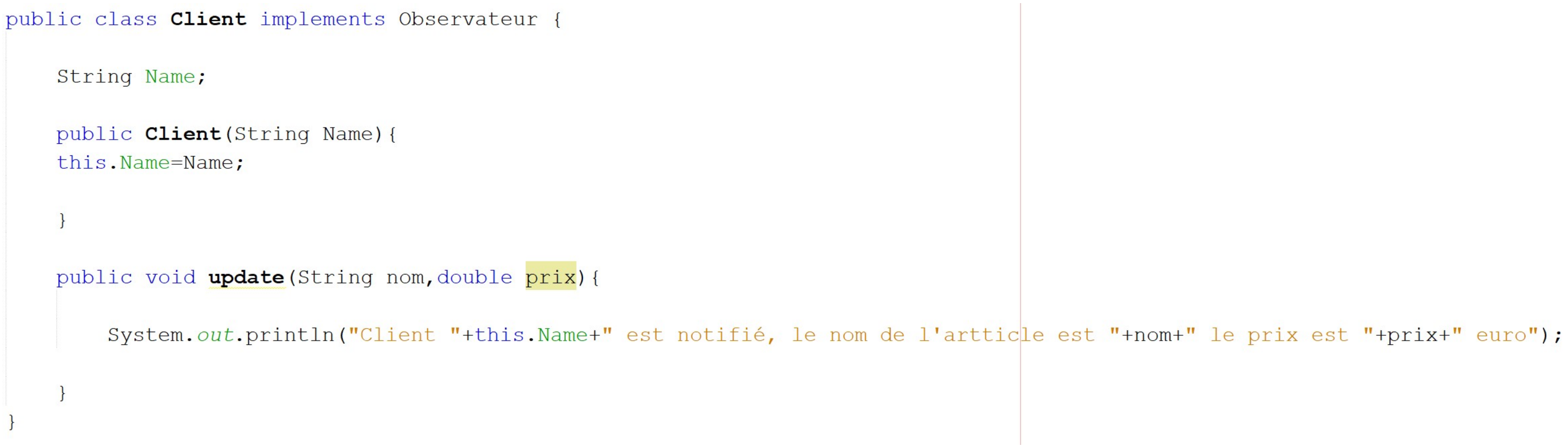
Observer pattern Example

Imagine that you have two types of objects: a Customer and a Store. The customer is very interested in a particular brand of product (say, it’s a new model of the iPhone) which should become available in the store very soon or is interested in the price of a particular item. If the price of a product offered for sale changes, customers should be able to be notified of that change.

Java Translation

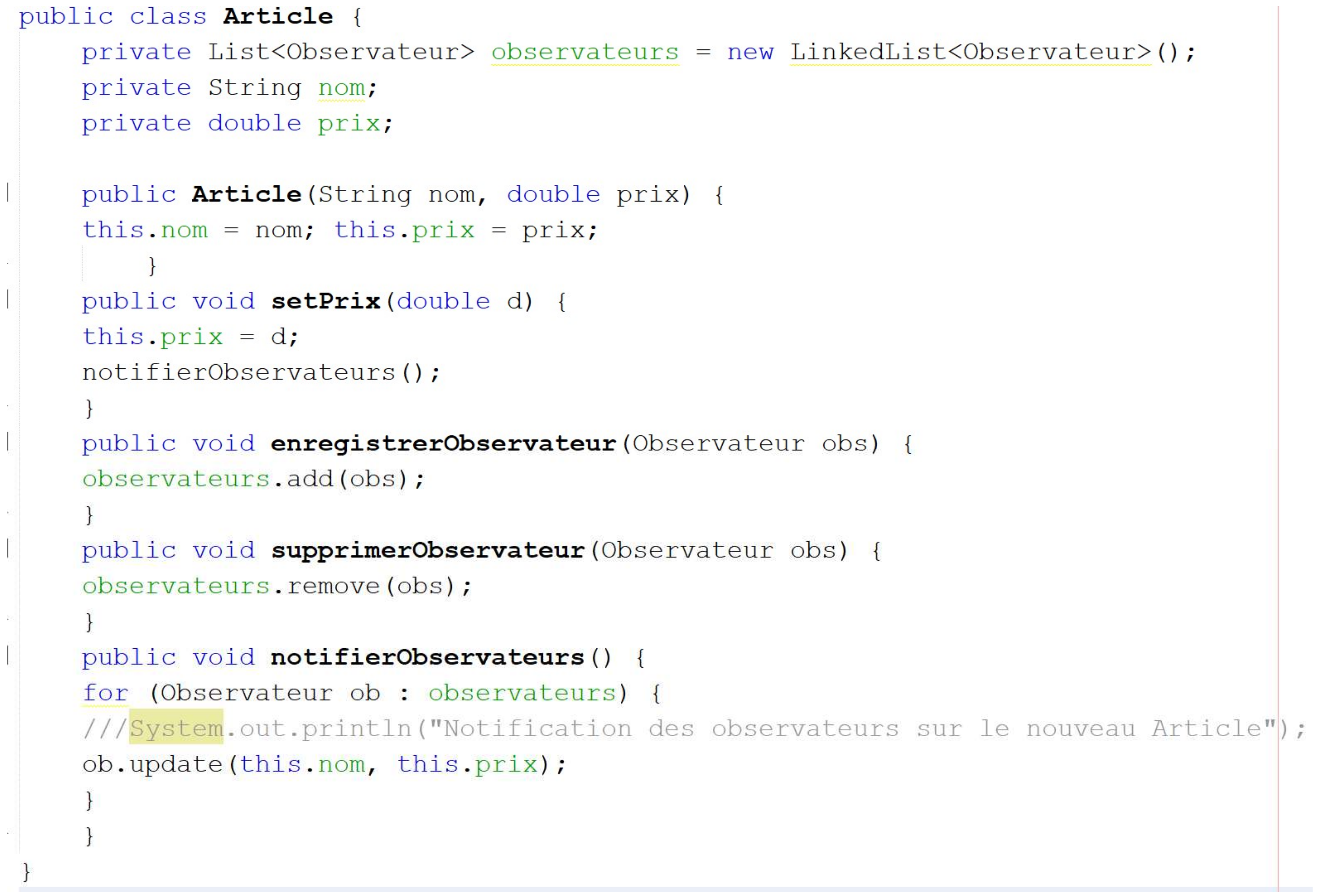


```java
public interface Observateur {


    public void update( String nom, double prix);


}
```



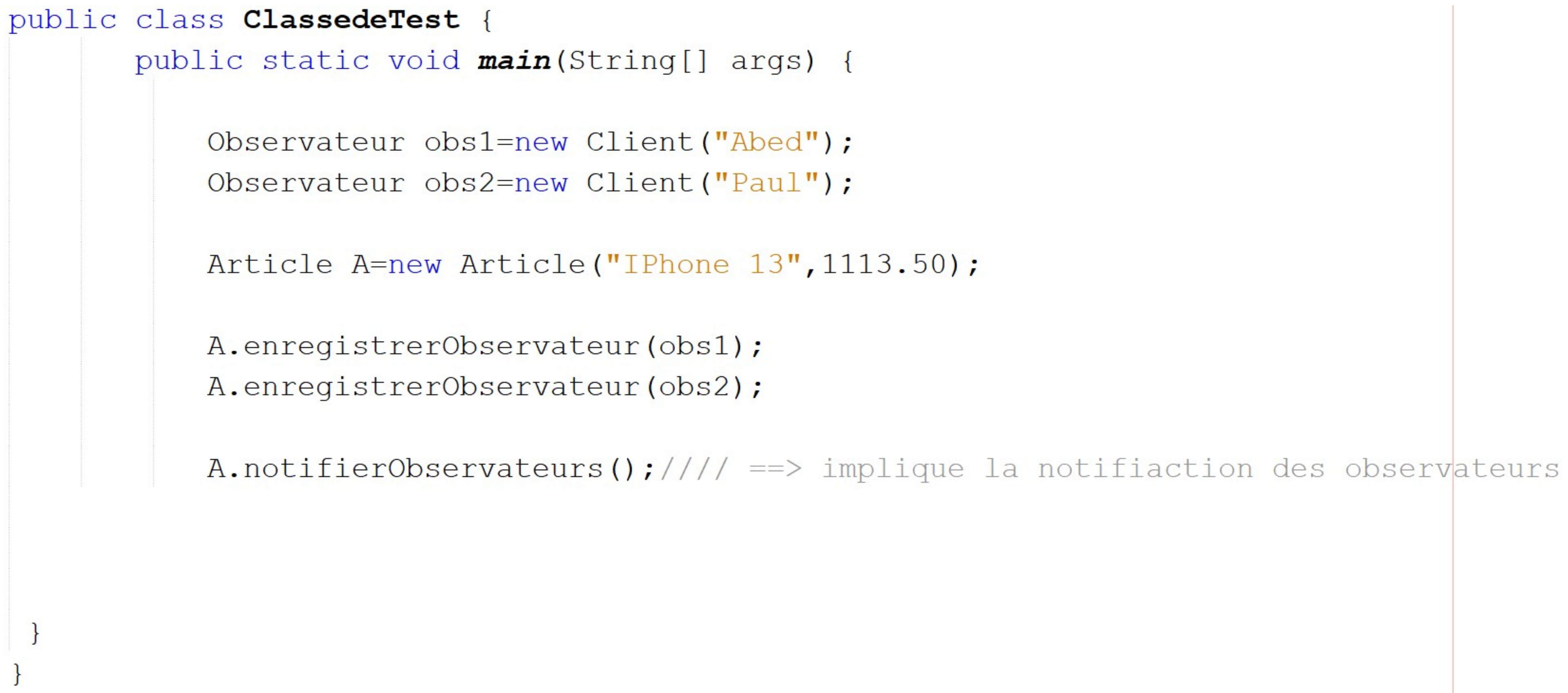
```java
public class Client implements Observateur {

    String Name;

    public Client(String Name){
    this.Name=Name;

    }

    public void update(String nom,double prix){

        System.out.println("Client "+this.Name+" est notifié, le nom de l'artticle est "+nom+" le prix est "+prix+" euro");

    }
}
```

Java Translation

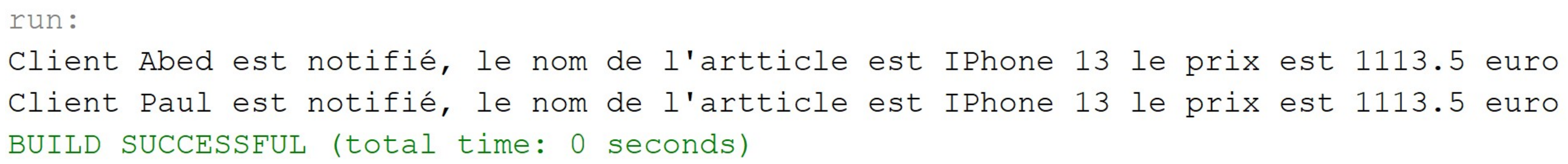


```java
public class Article {
    private List<Observateur> observateurs = new LinkedList<Observateur>();
    private String nom;
    private double prix;

    public Article(String nom, double prix) {
    this.nom = nom; this.prix = prix;
        }
    public void setPrix(double d) {
    this.prix = d;
    notifierObservateurs();
    }
    public void enregistrerObservateur(Observateur obs) {
    observateurs.add(obs);
    }
    public void supprimerObservateur(Observateur obs) {
    observateurs.remove(obs);
    }
    public void notifierObservateurs() {
    for (Observateur ob : observateurs) {
    ///System.out.println("Notification des observateurs sur le nouveau Article");
    ob.update(this.nom, this.prix);
    }
    }
}
```

Java Translation



```java
public class ClassedeTest {
        public static void main(String[] args) {

            Observateur obs1=new Client("Abed");
            Observateur obs2=new Client("Paul");

            Article A=new Article("IPhone 13",1113.50);

            A.enregistrerObservateur(obs1);
            A.enregistrerObservateur(obs2);

            A.notifierObservateurs();//// ==> implique la notifiaction des observateurs



  }
}
```



```
run:
Client Abed est notifié, le nom de l'artticle est IPhone 13 le prix est 1113.5 euro
Client Paul est notifié, le nom de l'artticle est IPhone 13 le prix est 1113.5 euro
BUILD SUCCESSFUL (total time: 0 seconds)
```

Plan

- Iterator
- Visitor
- Strategy
- Observer
- MVC
- Observer-MVC

MVC Architecture

MVC - Model View Controller

- The Model represents the business layer of the application.
- The View defines the presentation of the application
- The Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes.

These three components are usually implemented as separate classes.

The Model

- The Model is the part that does the work. It models the actual problem being solved
- The Model should be independent of both the Controller and the View
- But it provides methods for them to use
- Independence gives flexibility

The Controller

- The Controller decides what the model is to do
- Often, the user is put in control by means of a GUI, in this case, the GUI and the Controller are often the same (bad design)
- The Controller and the Model can always be separated (what to do versus how to do it)
- The Model should not depend on the Controller

The view

- Typically, the user has to be able to see, or view, what the program is doing
- The View describes the state of the Model
- The Model should be independent of the View, but it can provide access methods
- It is more flexible to let the View be independent of the model.

Combining Controller and View

- Sometimes the Controller and View are combined, especially in small programs

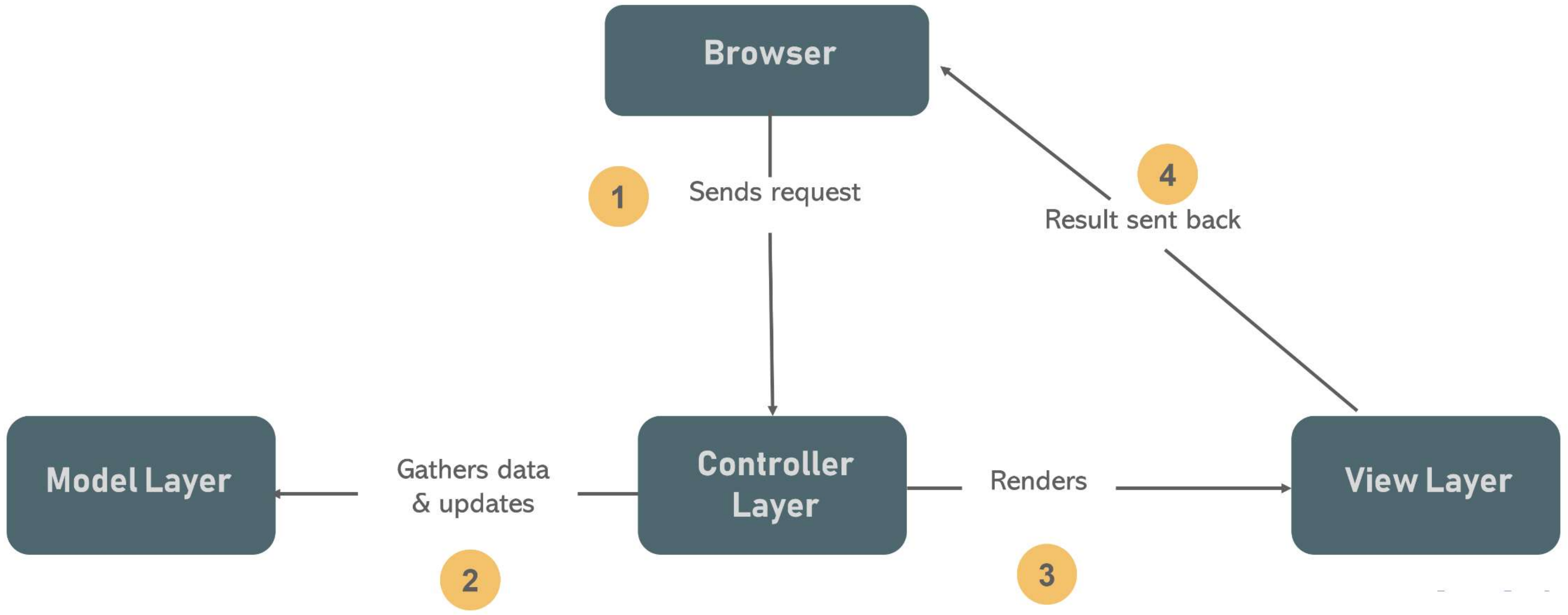
- Combining the Controller and View is appropriate if they are very interdependent

- The Model should still be independent

- Never mix Model code with GUI code!

MVC Architecture

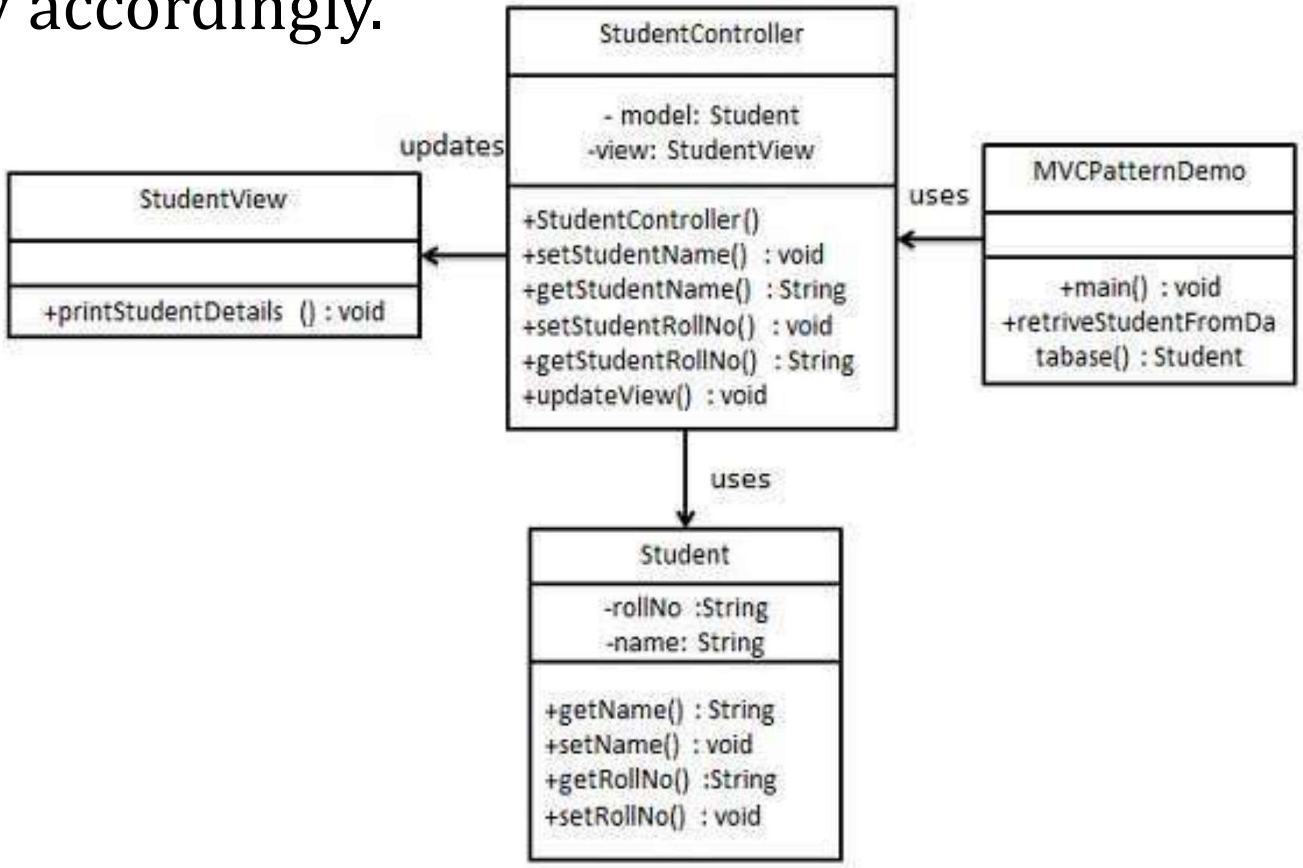
Due to this separation the user requests are processed as follows:



1.The browser on the client sends a request for a page to the controller present on the server
2.The controller performs the action of invoking the model, thereby, retrieving the data it needs in response to the request
3.The controller then gives the retrieved data to the view
4.The view is rendered and sent back to the client for the browser to display

Example

We are going to create a Student object acting as a model. StudentView will be a view class which can print student details on console and StudentController is the controller class responsible to store data in Student object and update view StudentView accordingly.



Model Layer

The Model in the MVC design pattern acts as a data layer for the application. It represents the business logic for application and also the state of application. The model object fetch and store the model state in the database. Using the model layer, rules are applied to the data that represents the concepts of application.

Step 1

```
public class Student {

 private String rollNo;
 private String name;

public String getRollNo() {
return rollNo; }

public void setRollNo(String rollNo) {
this.rollNo = rollNo; }

public String getName() {
return name; }

public void setName(String name) {
this.name = name; } }
```

View Layer

View represents the visualization of data received from the model. The view layer consists of output of application or user interface. It sends the requested data to the client, that is fetched from model layer by controller.

Step 2

```
public class StudentView {
public void printStudentDetails(String studentName, String studentRollNo){
 System.out.println("Student: ");
System.out.println("Name: " + studentName);
System.out.println("Roll No: " + studentRollNo); }

}
```

Controller Layer

The controller layer gets the user requests from the view layer and processes them, with the necessary validations.

It acts as an interface between Model and View. The requests are then sent to model for data processing. Once they are processed, the data is sent back to the controller and then displayed on the view.

Step 3

```
public class StudentController {

private Student model;
private StudentView view;

public StudentController(Student model, StudentView view){
 this.model = model;
this.view = view; }

public void setStudentName(String name){ model.setName(name); }
public String getStudentName(){ return model.getName(); }

public void setStudentRollNo(String rollNo){ model.setRollNo(rollNo); }
public String getStudentRollNo(){ return model.getRollNo(); }

public void updateView(){
 view.printStudentDetails(model.getName(), model.getRollNo()); } }
```

Step 4

```
public class MVCPatternDemo {

public static void main(String[] args) {

//fetch student record based on his roll no from the database
 Student model = retriveStudentFromDatabase();

//Create a view : to write student details on console
StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);
controller.updateView();

 //update model data
controller.setStudentName("John");
controller.updateView(); }

 private static Student retriveStudentFromDatabase(){
 Student student = new Student();
 student.setName("Robert");
student.setRollNo("10");
return student; }
 }
```

Plan

- Iterator
- Visitor
- Strategy
- Observer
- MVC
- Observer-MVC

MVC in Java: Observer and Observable

Observable

- An Observable is an object that can be observed
- An Observer is notified when an object that it is observing announces a change
- When an Observable wants the world to know about what it has done, it executes:

```
setChanged ();
notifyObservers ();
```

- The Observable doesn't know or care who is looking
- But you have attach an Observer to the Observable with:

```
myObservable . addObserver ( myObserver );
```

- This is best done in the controller class - not in the model class!

MVC in Java: Observer and Observable

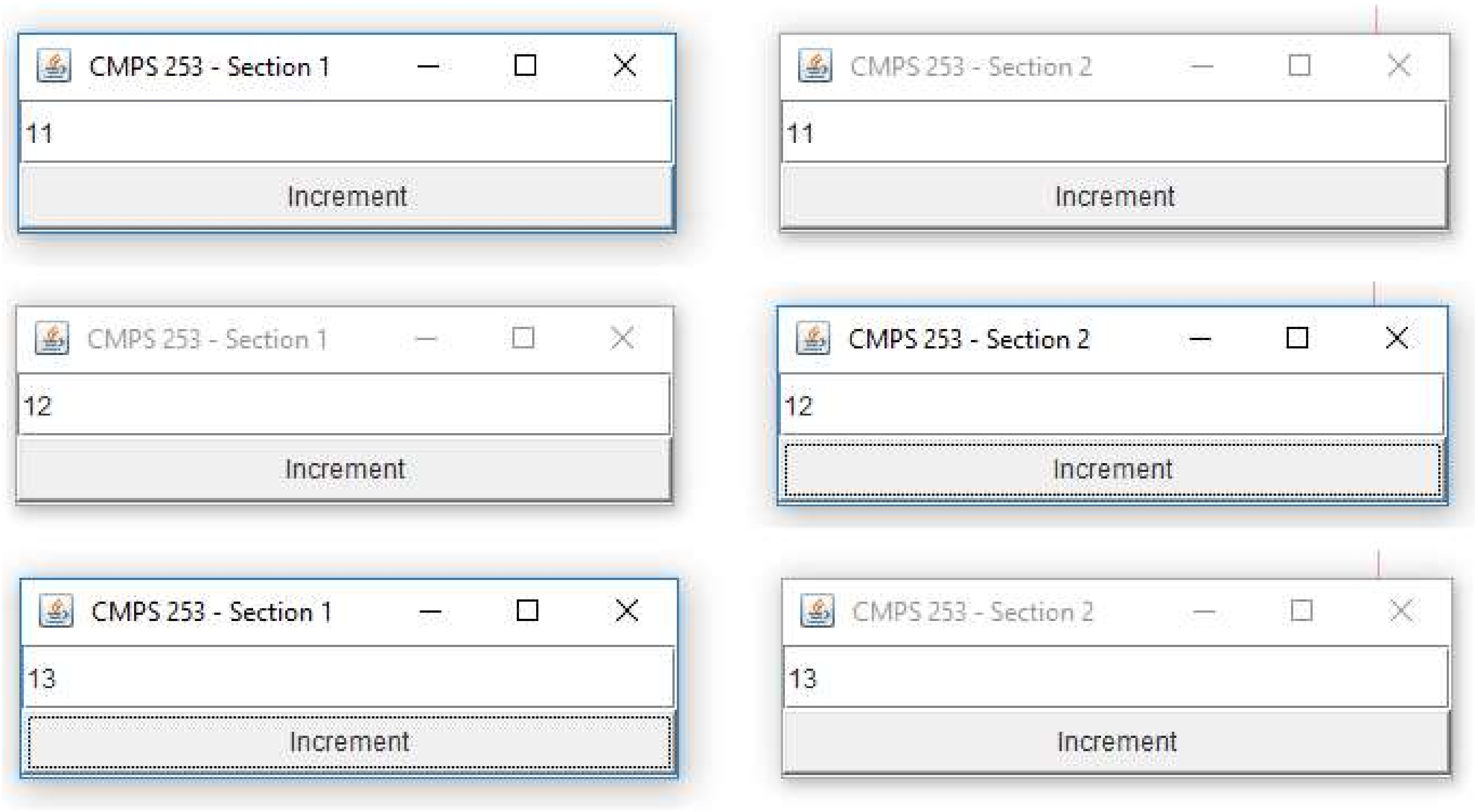
Observer

- Observer is an interface
- An Observer implements:

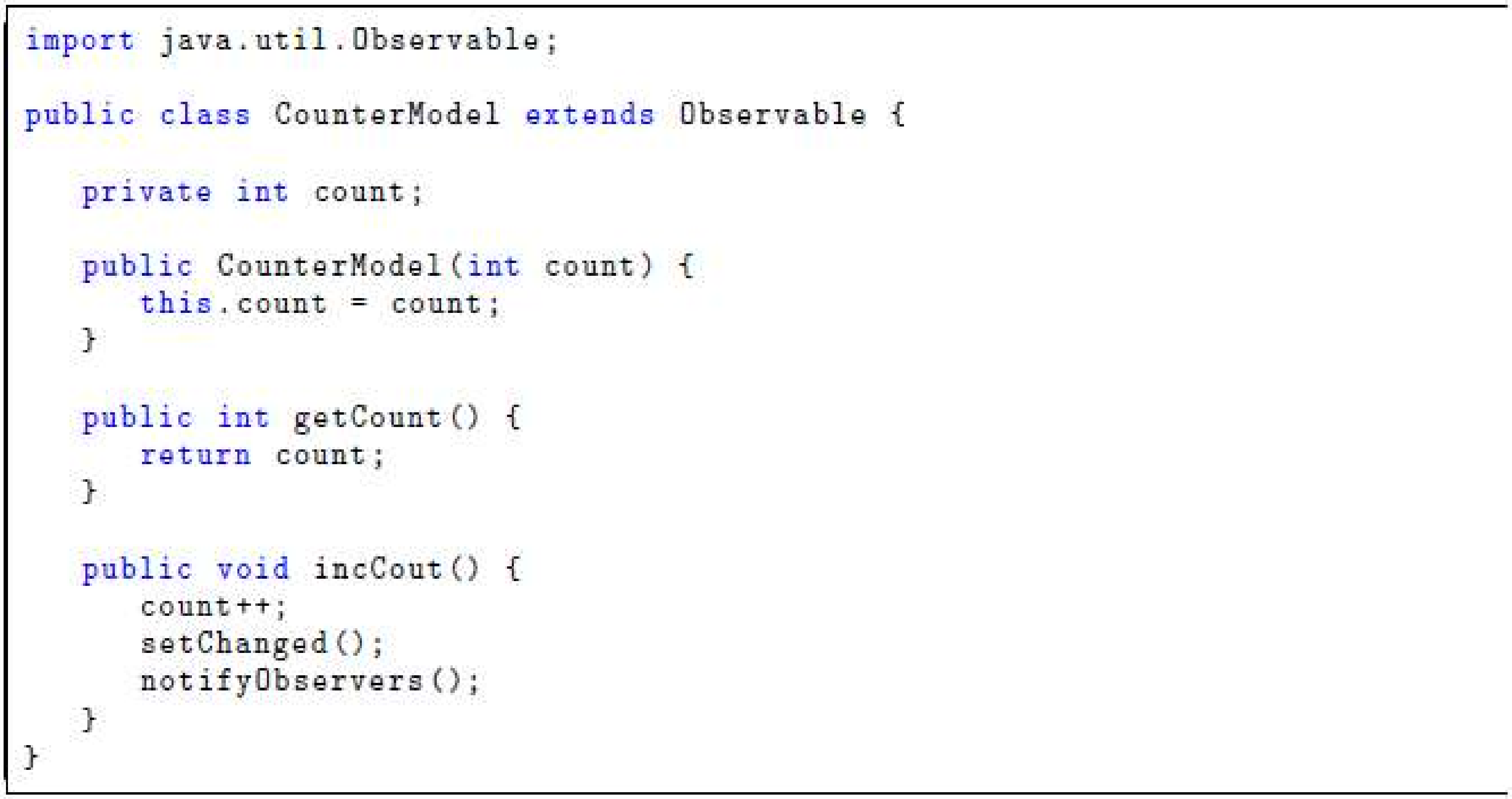
```
public void update ( Observable obs , Object arg)
```

- This method is invoked whenever an Observable that it is listening to does an notifyObservers([obs]) and the Observable object called setChanged()
- The obs argument is a reference to the observable object itself.

Counter Example



Model-Counter Model

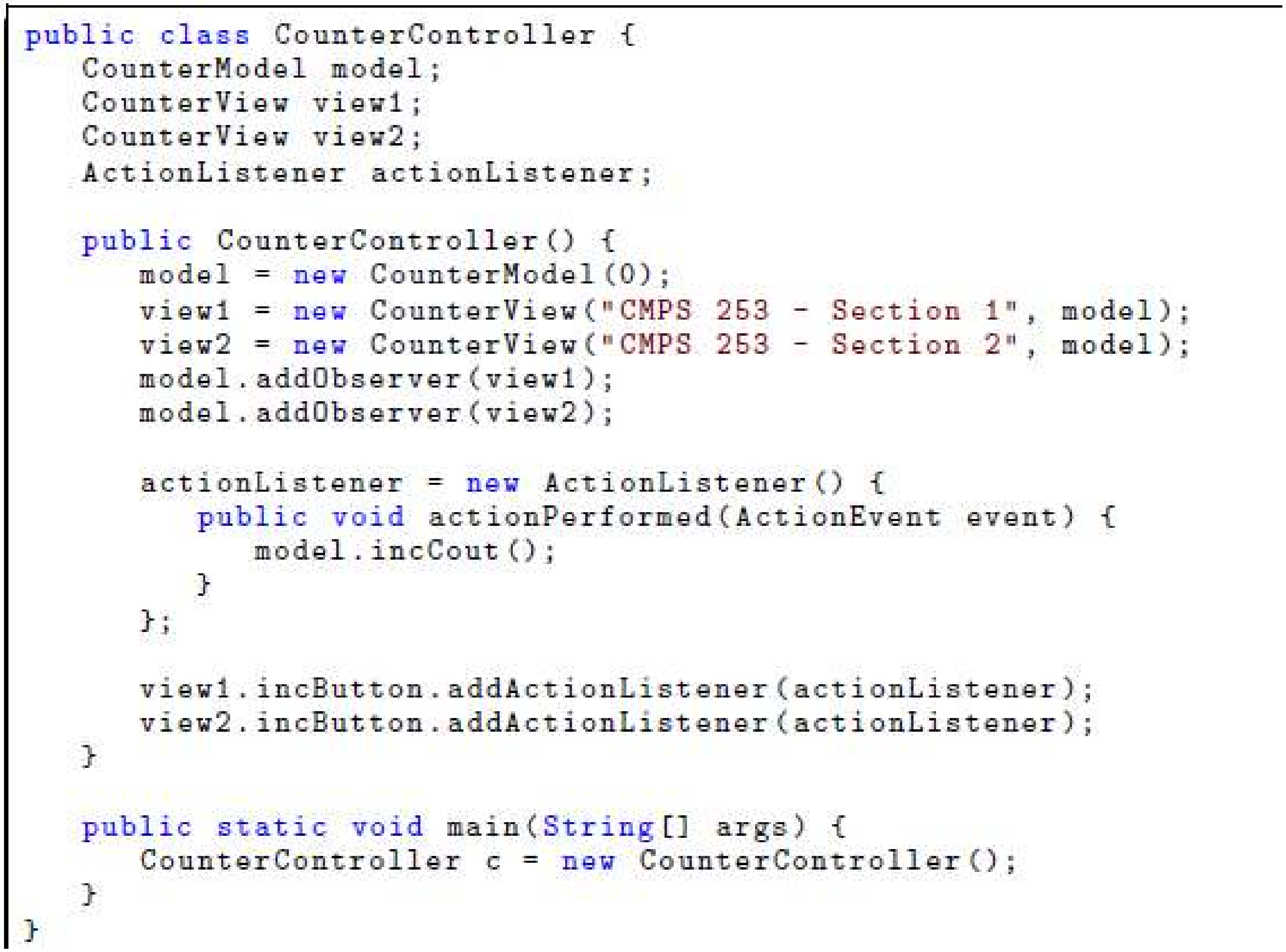


```java
import java.util.Observable;

public class CounterModel extends Observable {

   private int count;

   public CounterModel(int count) {
      this.count = count;
   }

   public int getCount() {
      return count;
   }

   public void incCout() {
      count++;
      setChanged();
      notifyObservers();
   }
}
```

View-CounterView

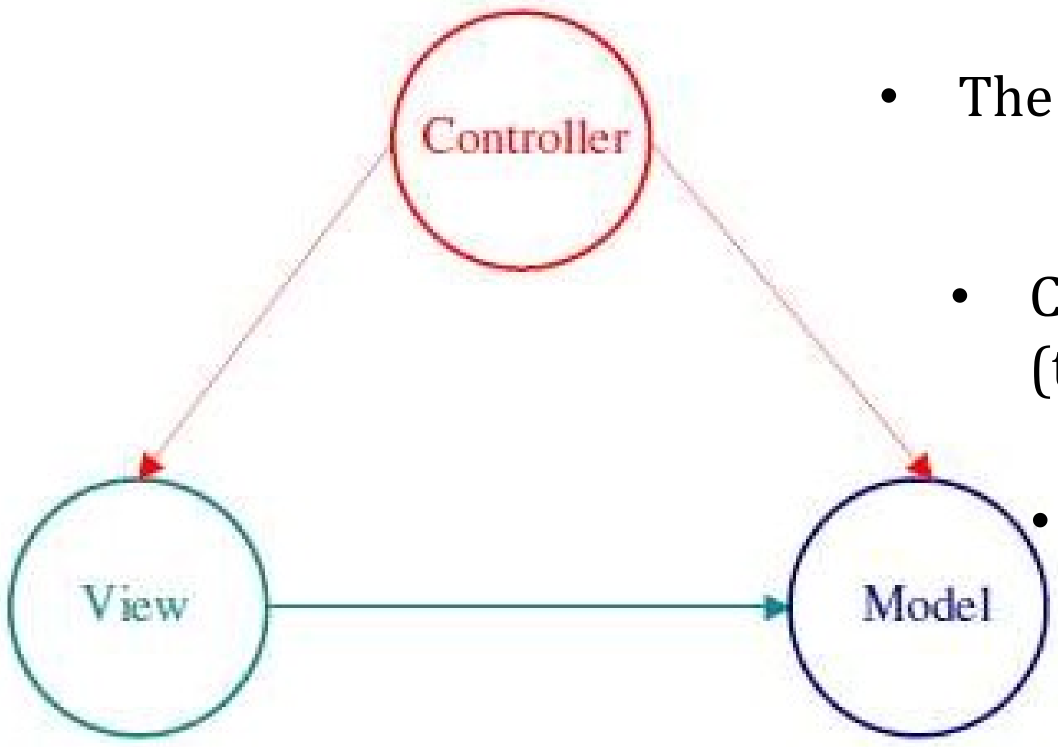
```java
import java.util.Observable;
import java.util.Observer;
...
public class CounterView extends JFrame implements Observer {

   private JTextField tf = new JTextField(10);
   private CounterModel model;
   Button incButton = new Button("Increment");;

   public CounterView(String title, CounterModel m) {
      setTitle(title);
      setSize(200,100);
      setLayout(new GridLayout(2,1));

      model = m;

      add(tf);
      add(incButton);
      setVisible(true);
   }

   @Override
   public void update(Observable o, Object arg) {
      if(o == model)
         tf.setText(((CounterModel) model).getCount() + "");
   }
}
```

Controller - CounterController



```java
public class CounterController {
    CounterModel model;
    CounterView view1;
    CounterView view2;
    ActionListener actionListener;

    public CounterController() {
        model = new CounterModel(0);
        view1 = new CounterView("CMPS 253 - Section 1", model);
        view2 = new CounterView("CMPS 253 - Section 2", model);
        model.addObserver(view1);
        model.addObserver(view2);

        actionListener = new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                model.incCout();
            }
        };

        view1.incButton.addActionListener(actionListener);
        view2.incButton.addActionListener(actionListener);
    }

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

Counter Example



- The user interacts with the view.
- Controller calls methods of the model (to update or to get some information)
- Controller sends the data to the viewer.

Questions?