Graphical Interface and Application(I3305)

Chapter 3: Behavioral Patterns

Lebanese University
Faculty of Science 1 - Department of Computer Science
Abed EL Safadi

Outline

- Introduction
- Creational patterns
- Structural patterns
- Behavioral patterns

Introduction

Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

Examples

1. The Visitor pattern adds function to a class

2. The MVC pattern is used to separate the logic of different layers in a program in independent units.

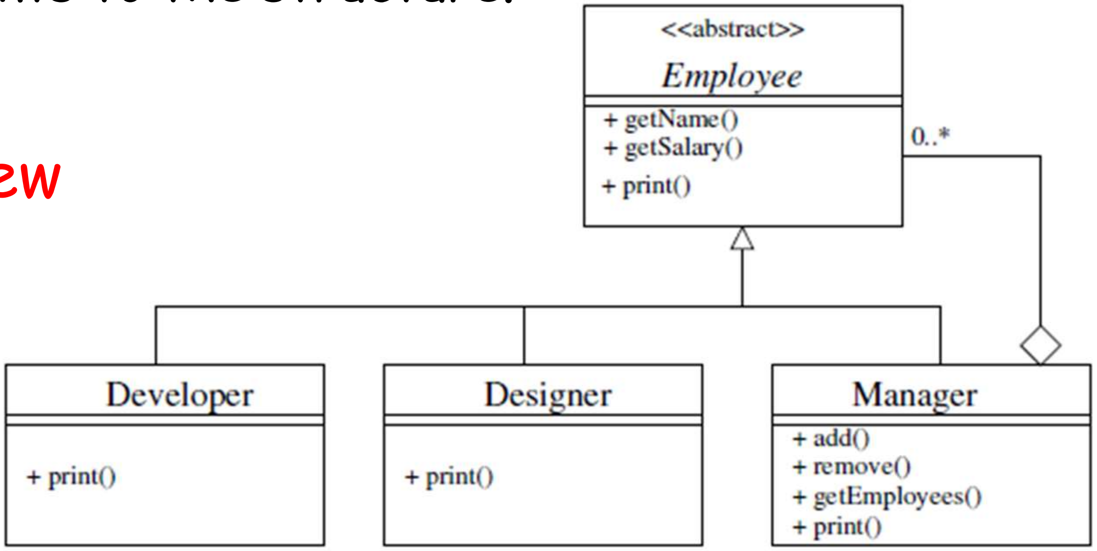
3. The Observer pattern defines the way a number of classes can be notified of a change

Visitor Pattern

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?

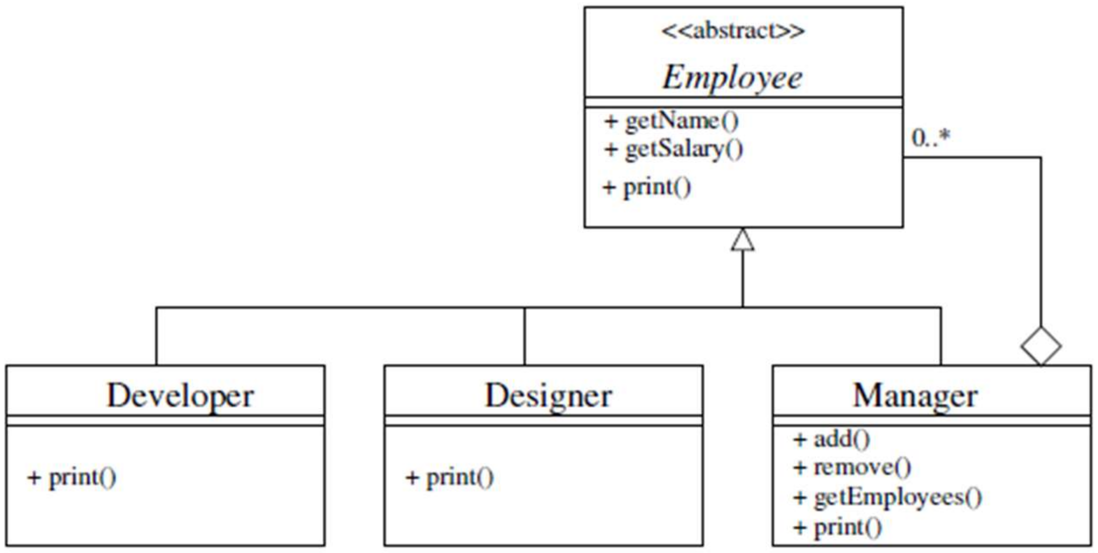


Visitor Pattern

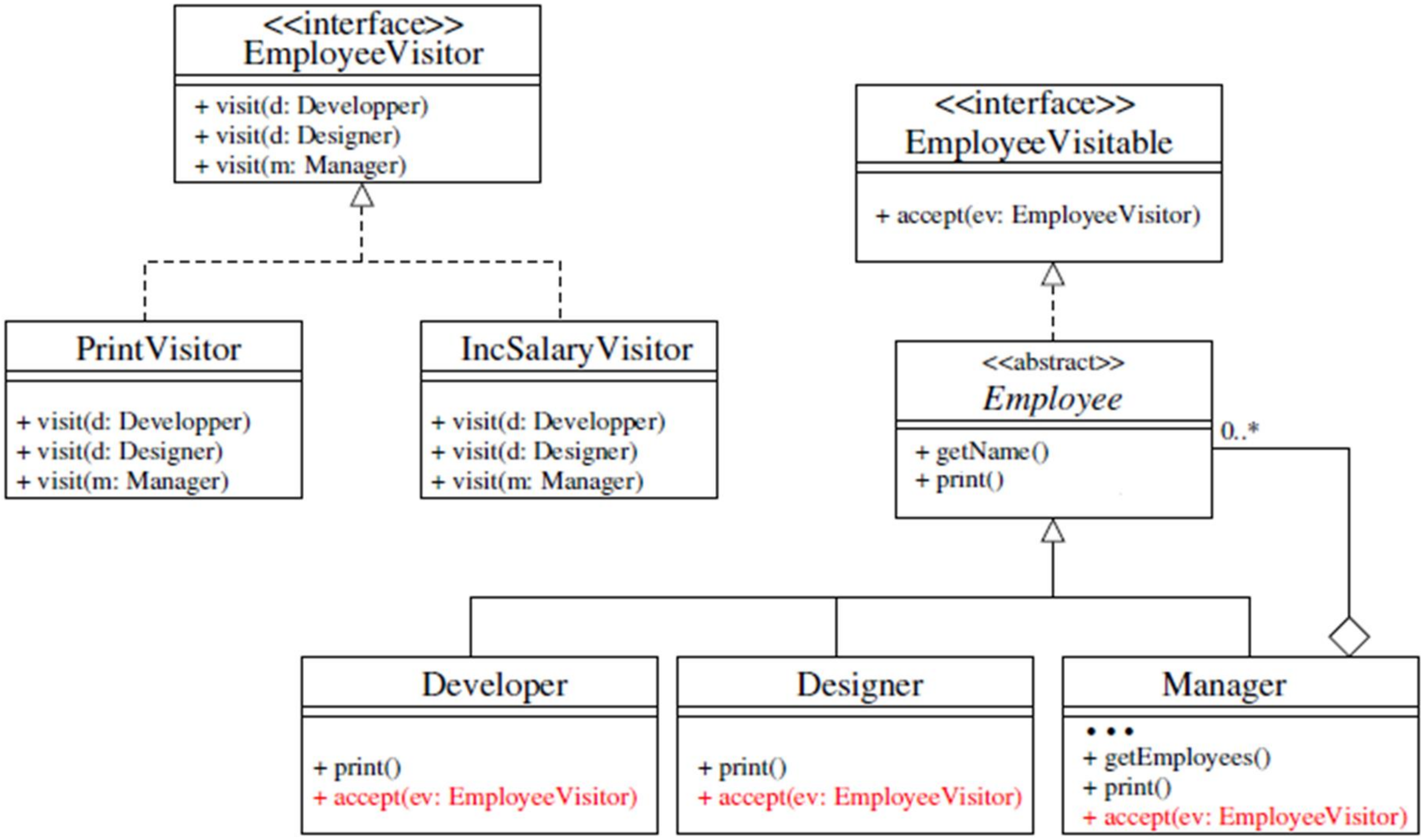
Example (Employee)

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).



Visitor Pattern



Visitor Pattern

- 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
- 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.

Visitor Pattern -Employee Example

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

```
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();
}
```

Visitor Pattern -Employee Example

```
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);
      }
   }
}
```

Visitor Pattern -Employee Example

```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);
   }
}
```

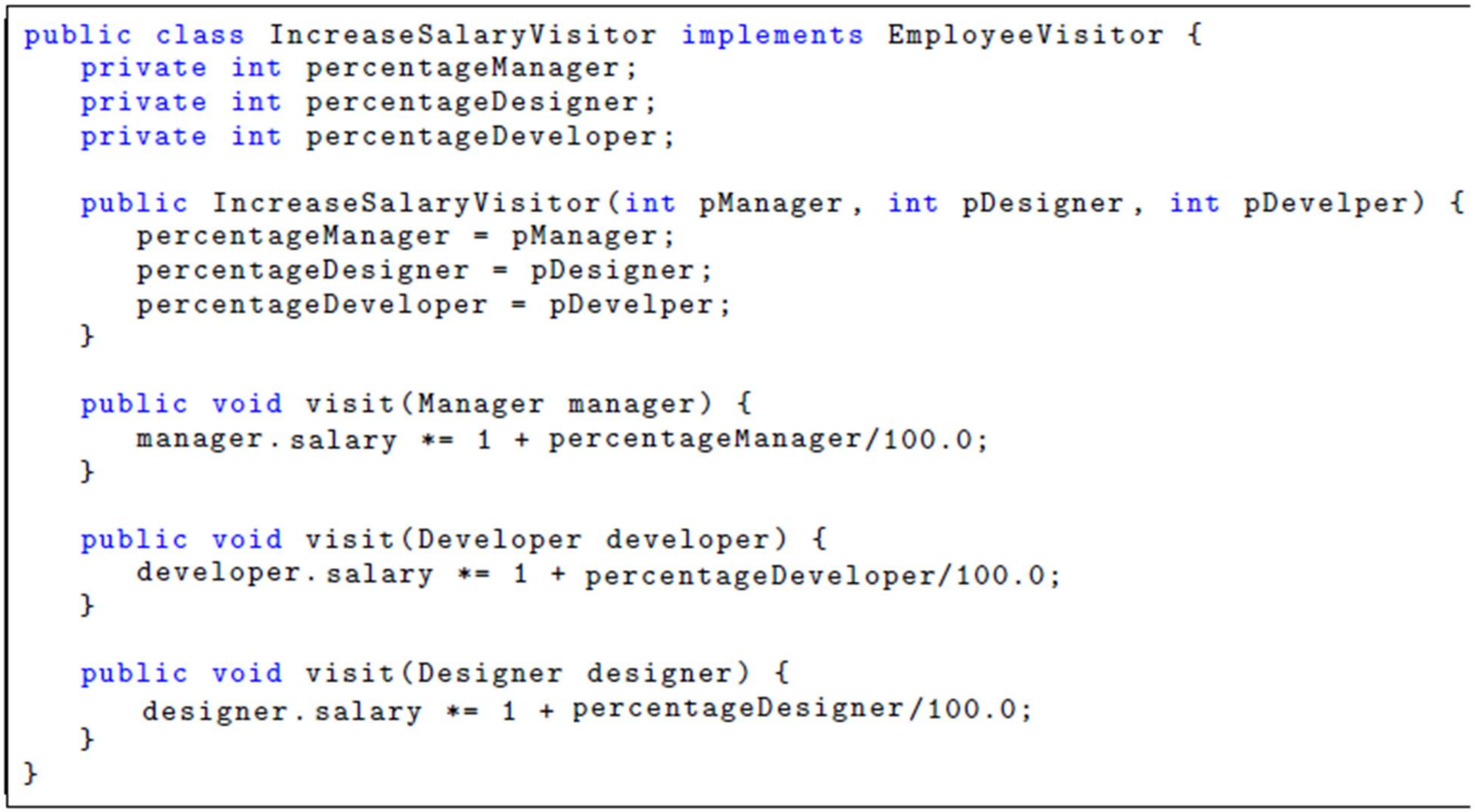
```java
public class Designer extends Employee {
   public Designer(String name, int salary) {
      super(name, salary);
   }

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

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

Visitor Pattern -Employee Example

```
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());
   }
}
```

Visitor Pattern -Employee Example

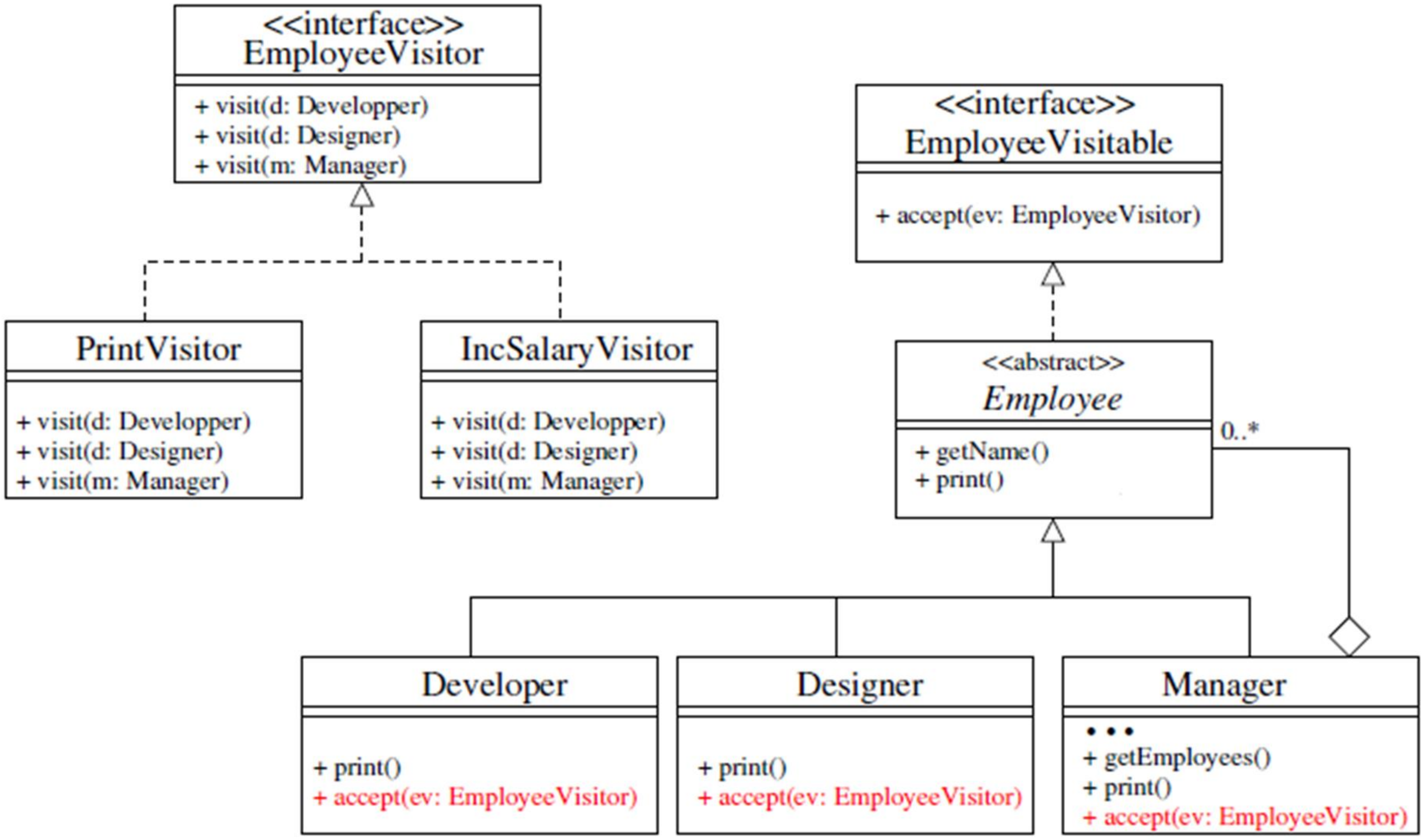


```
public class IncreaseSalaryVisitor implements EmployeeVisitor {
   private int percentageManager;
   private int percentageDesigner;
   private int percentageDeveloper;

   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;
   }
}
```

Visitor Pattern -Employee Example

```
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());
    }
}
```

Visitor Pattern Overview



MVC Pattern

MVC - Model View Controller

- The Model is the actual internal representation.
- The View is a way of looking at or displaying the model
- The Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data

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 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 (); /* or */ notifyObservers ( arg );
// The arg can be any object
```

- 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

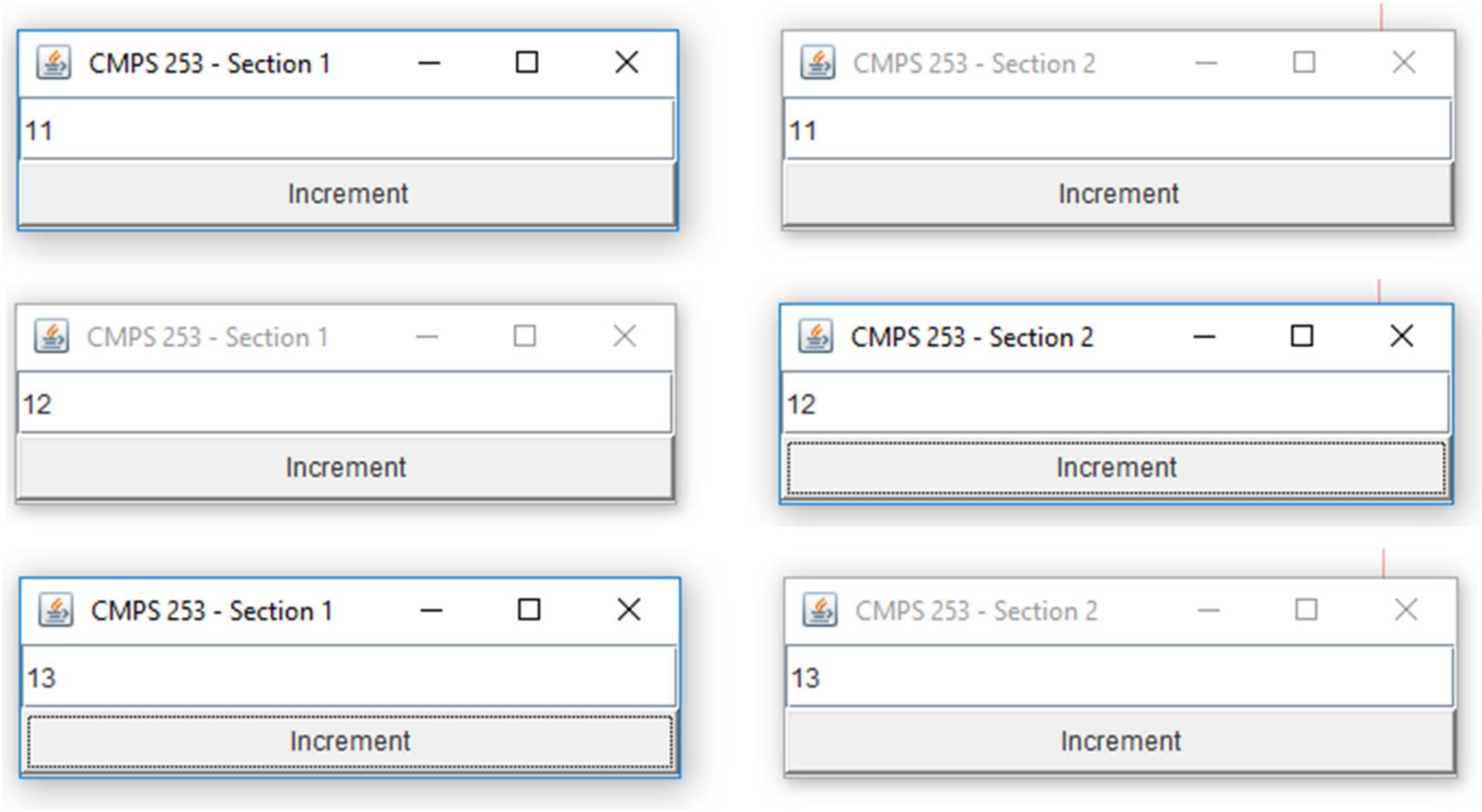
- Observer is an interface
- An Observer implements:

```
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



CMPS 253 - Section 1

11

Increment

CMPS 253 - Section 2

11

Increment

CMPS 253 - Section 1

12

Increment

CMPS 253 - Section 2

12

Increment

CMPS 253 - Section 1

13

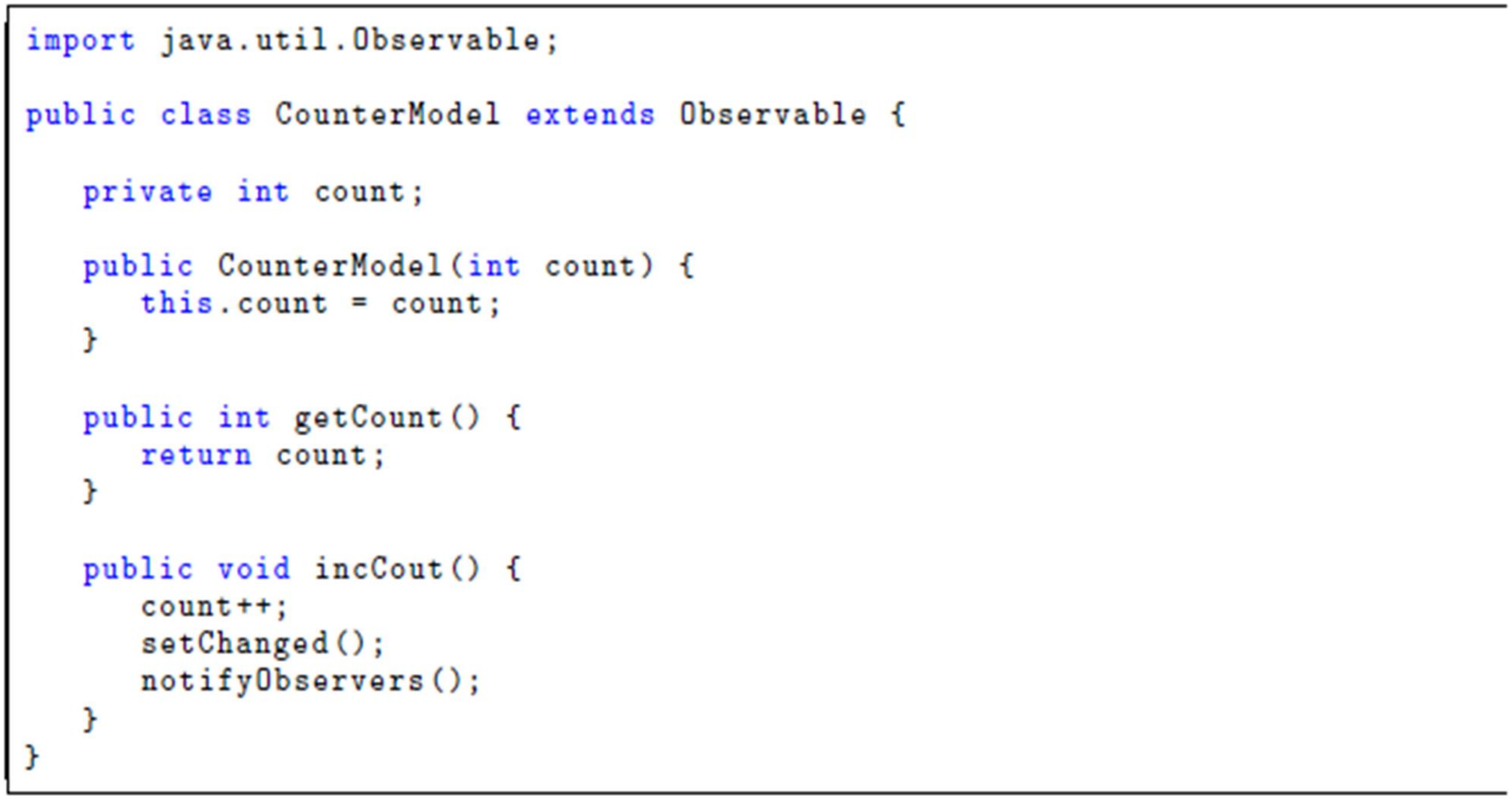
Increment

CMPS 253 - Section 2

13

Increment

Model - CounterModel

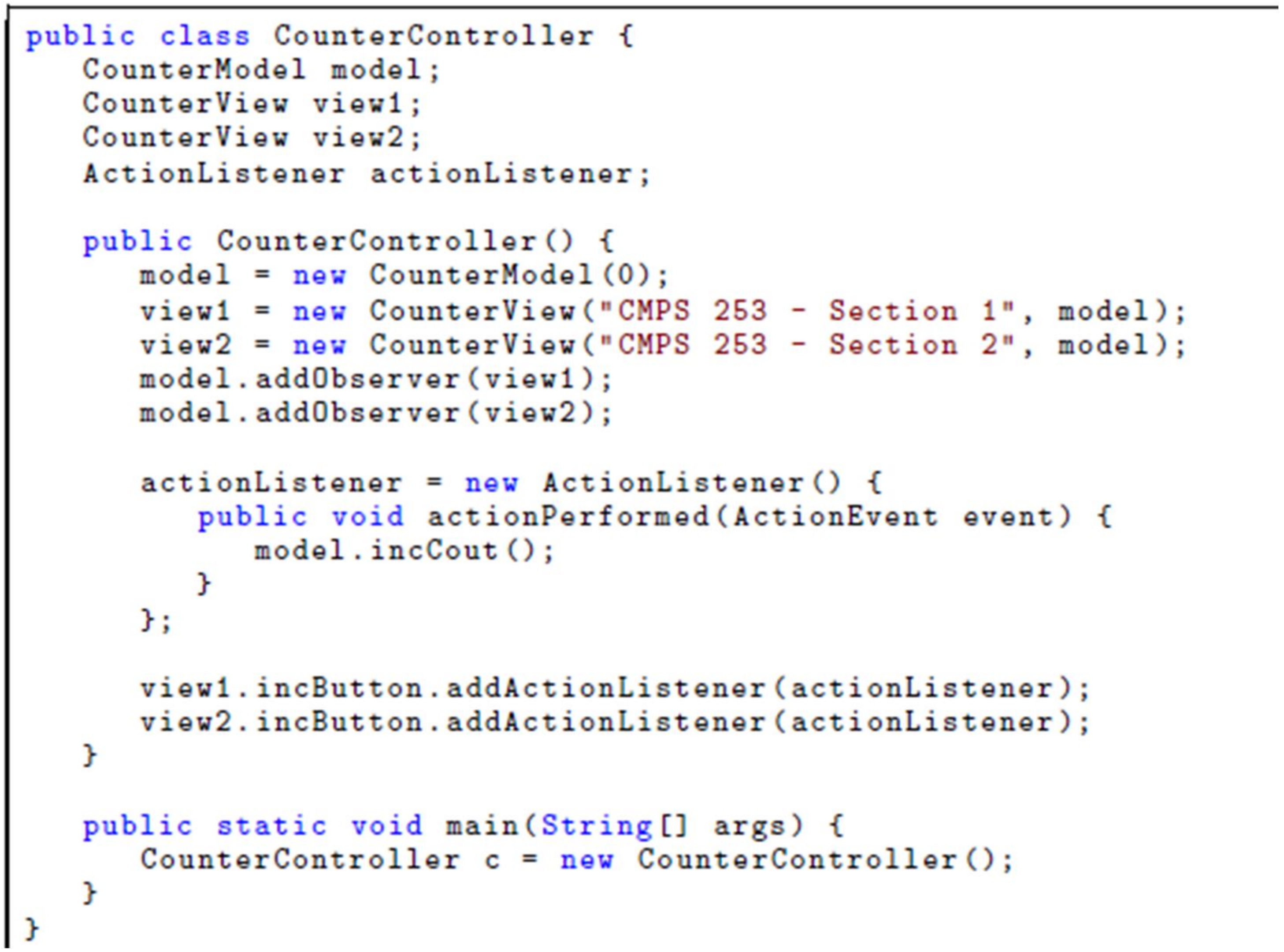


```
import java.util.Observable;

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

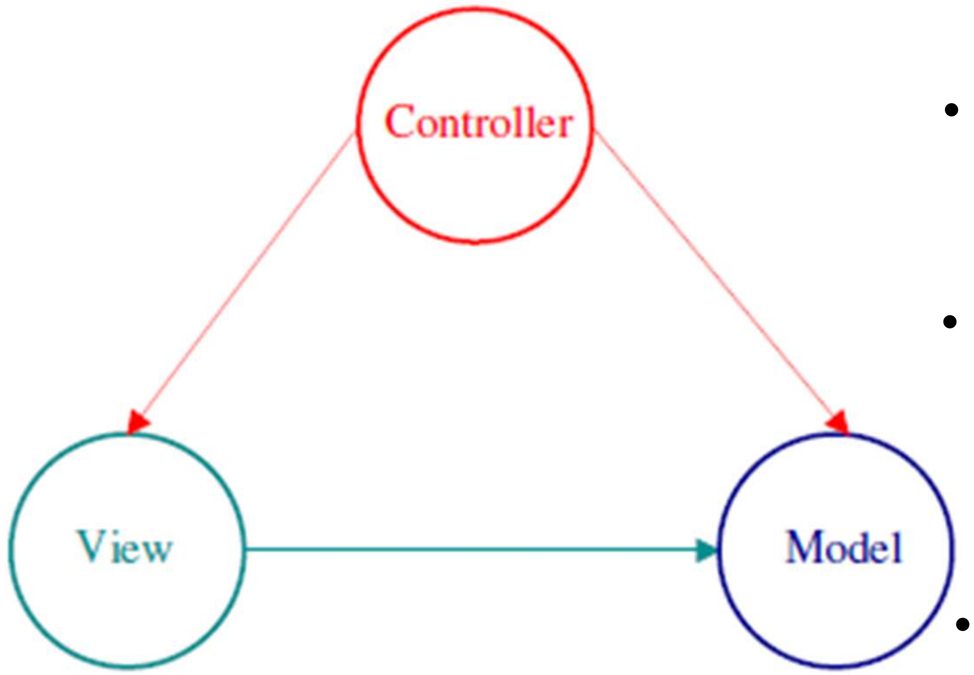
```java
import java.util.Observable;
import java.util.Observer;
...
public class CounterView extends JFrame implements Observer {

   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



```
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.