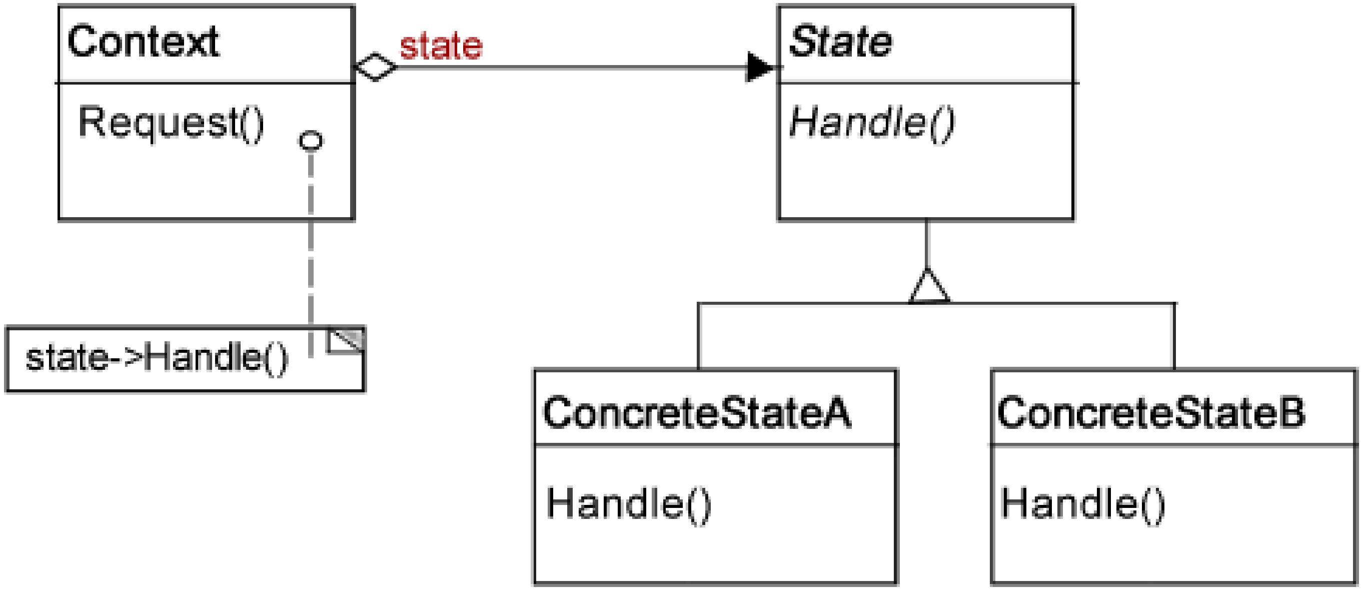
State Pattern

- The main idea is that, at any given moment, there’s a *finite* number of *states* which a program can be in.
- Within any unique state, the program behaves differently, and the program can be switched from one state to another instantaneously.
- However, depending on a current state, the program may or may not switch to certain other states.
- These switching rules, called *transitions*, are also finite and predetermined

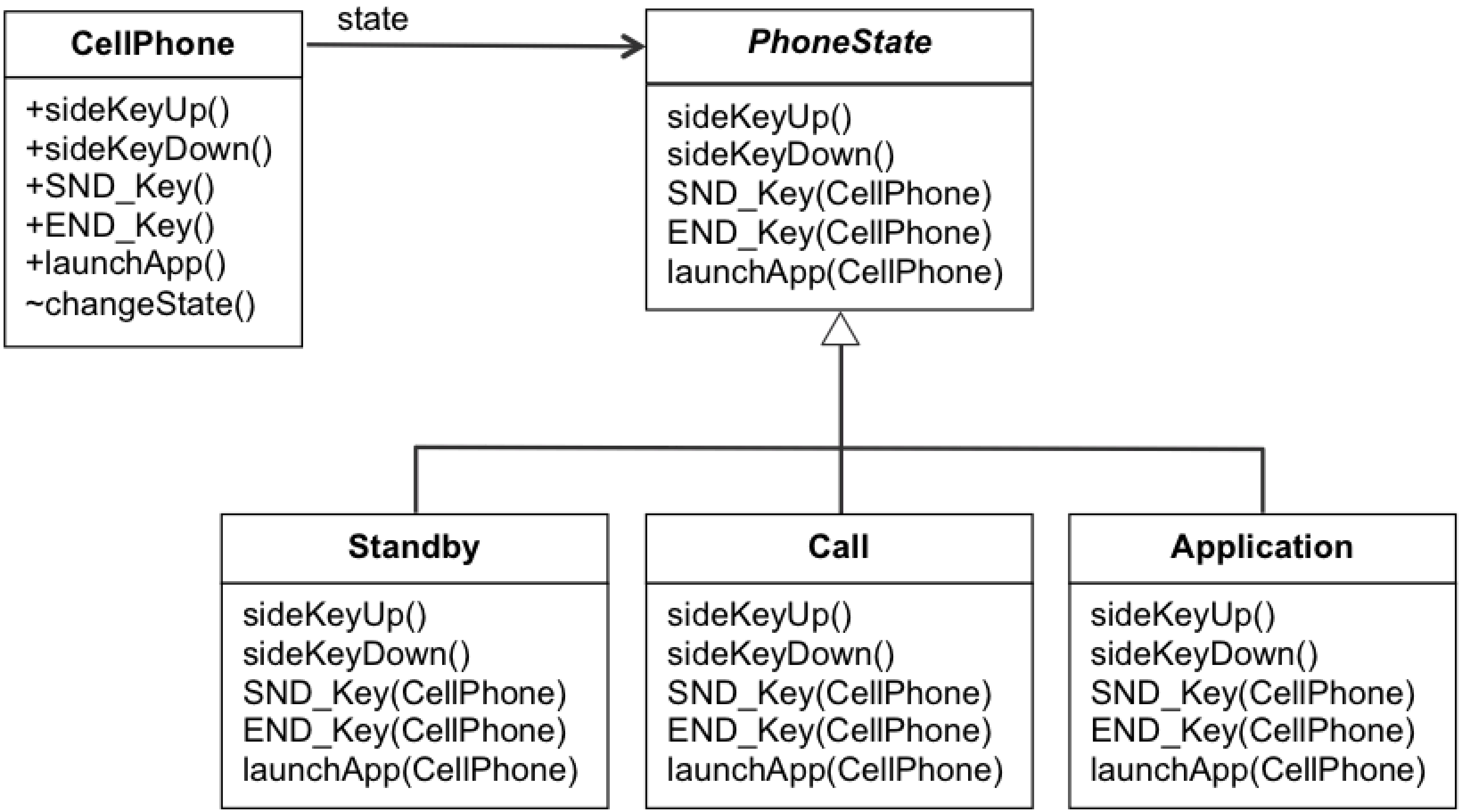
State

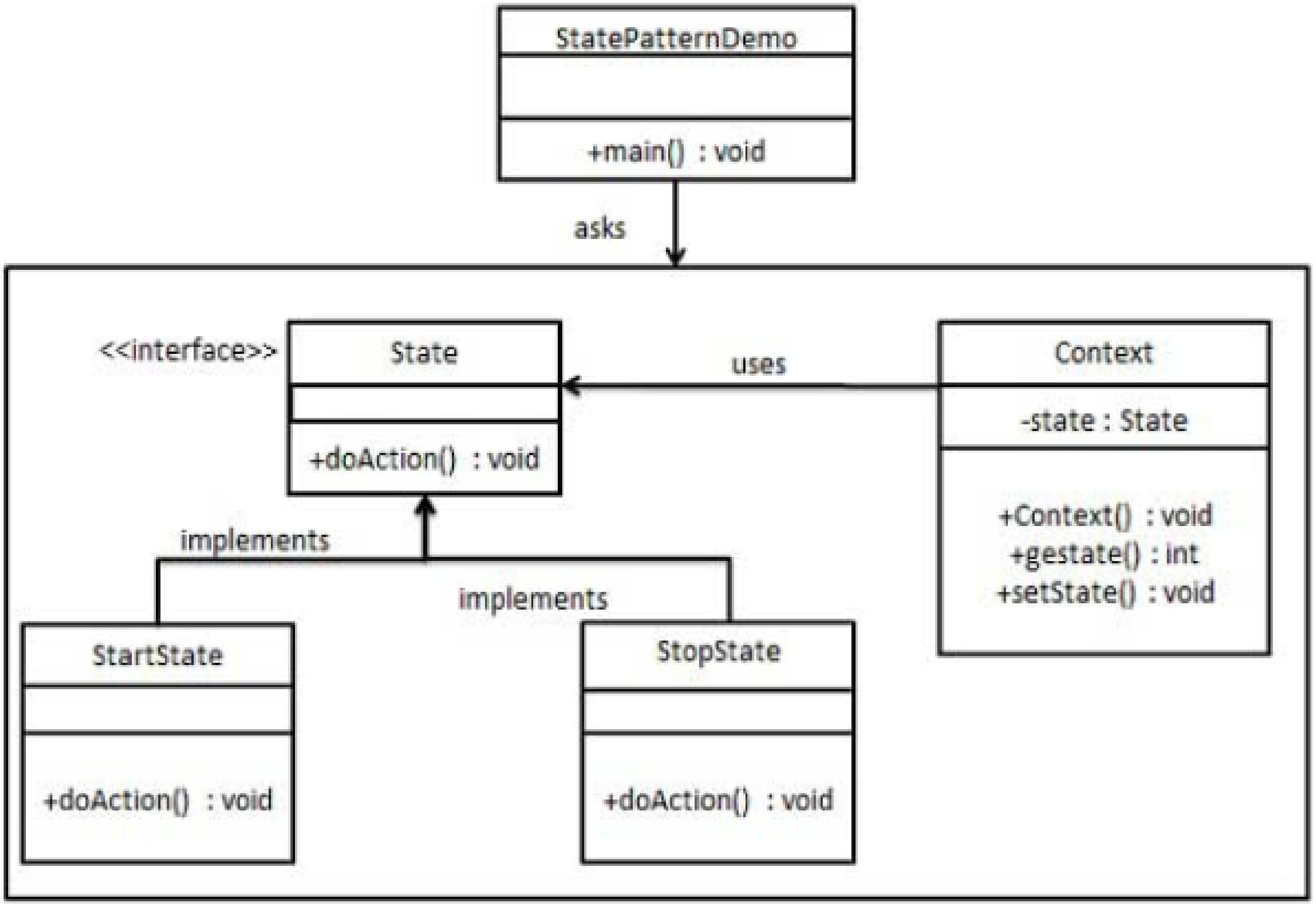
Intent	• Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Problem	• A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state.

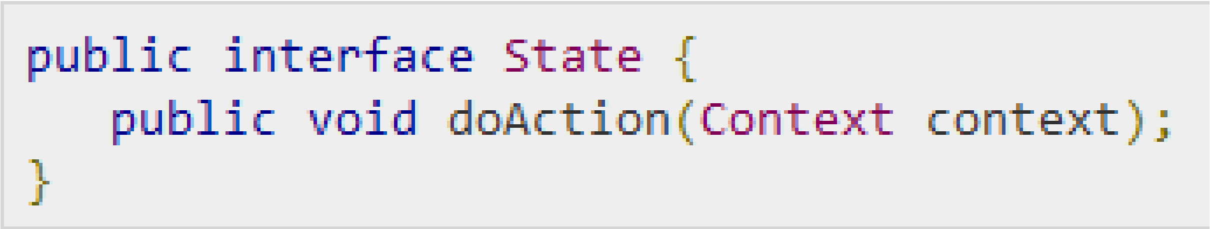
Solution


Context
Request()
state
State
Handle()
state->Handle()
ConcreteStateA
Handle()
ConcreteStateB
Handle()

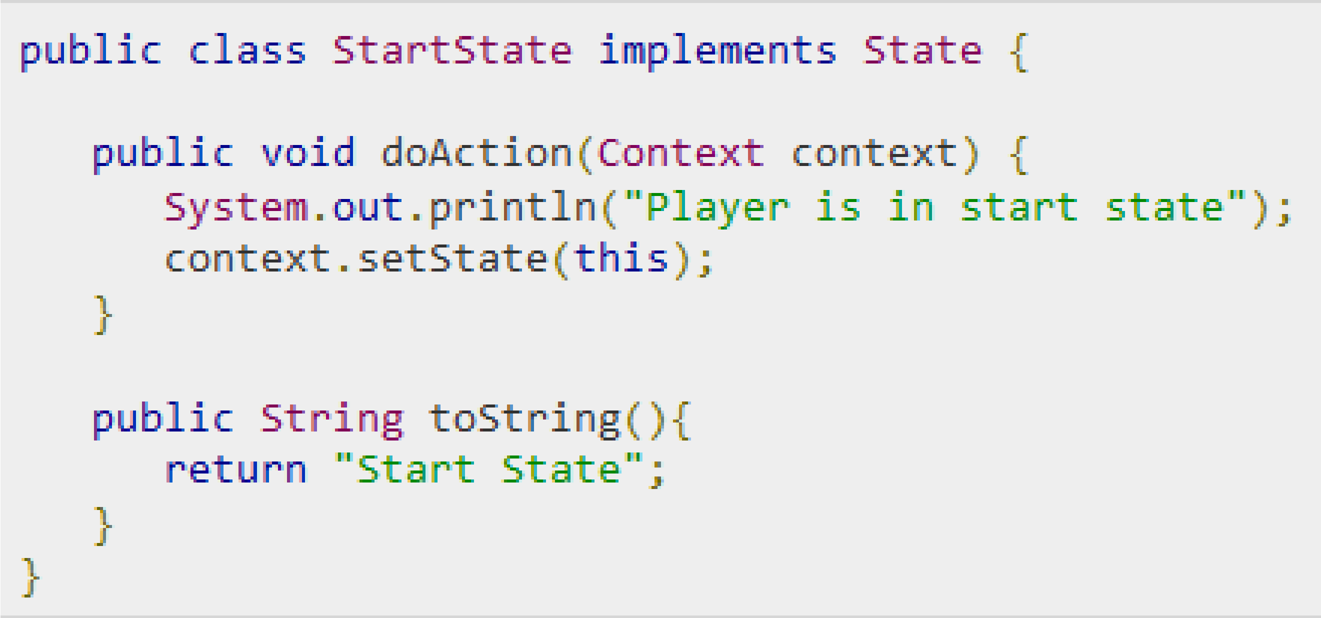
Example


CellPhone
+sideKeyUp()
+sideKeyDown()
+SND_Key()
+END_Key()
+launchApp()
~changeState()
state
PhoneState
sideKeyUp()
sideKeyDown()
SND_Key(CellPhone)
END_Key(CellPhone)
launchApp(CellPhone)
Standby
sideKeyUp()
sideKeyDown()
SND_Key(CellPhone)
END_Key(CellPhone)
launchApp(CellPhone)
Call
sideKeyUp()
sideKeyDown()
SND_Key(CellPhone)
END_Key(CellPhone)
launchApp(CellPhone)
Application
sideKeyUp()
sideKeyDown()
SND_Key(CellPhone)
END_Key(CellPhone)
launchApp(CellPhone)

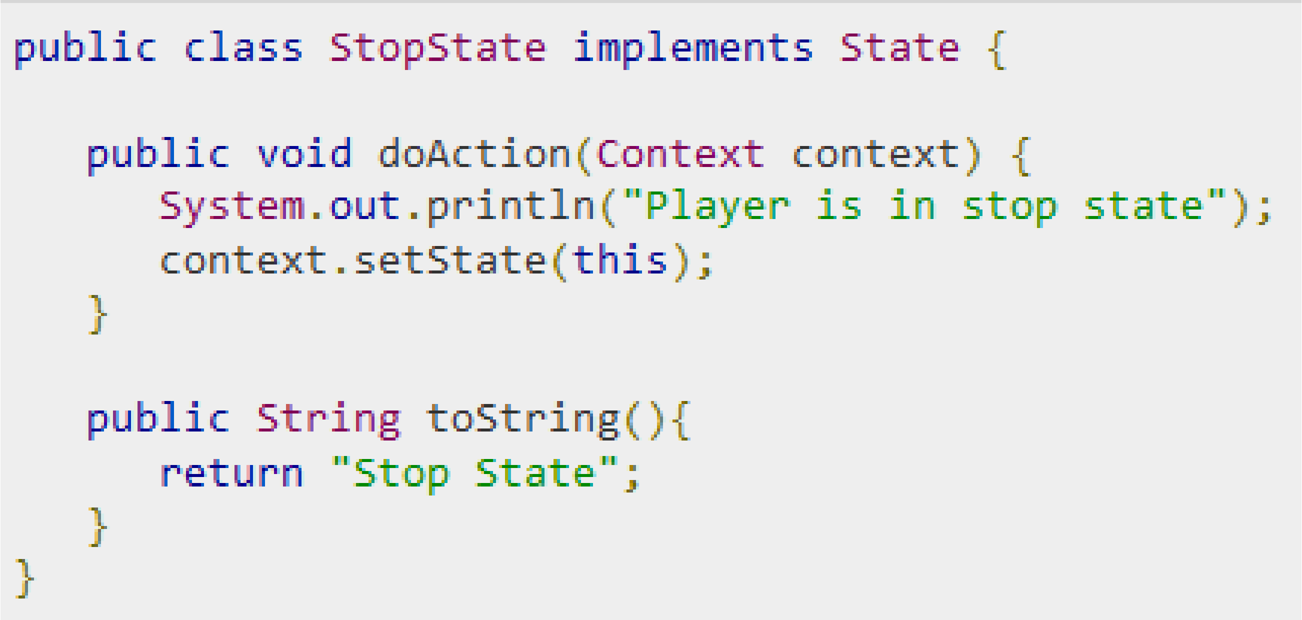




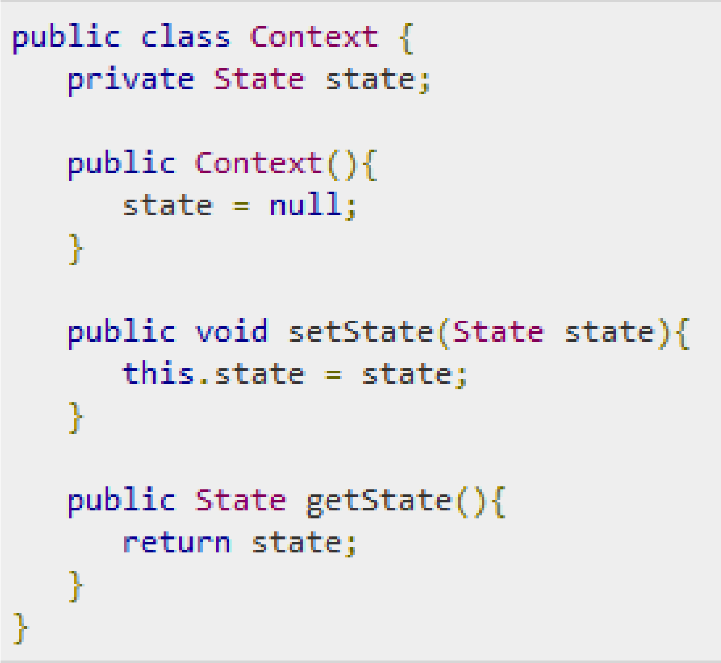
```
public interface State {
   public void doAction(Context context);
}
```



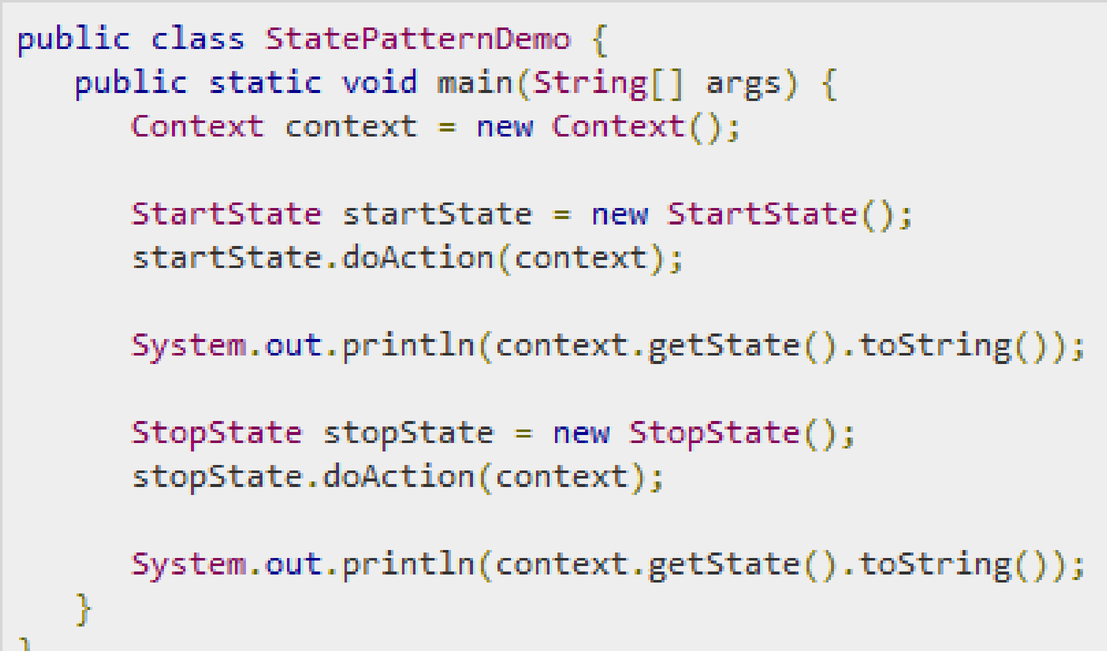
```java
public class StartState implements State {

   public void doAction(Context context) {
      System.out.println("Player is in start state");
      context.setState(this);
   }

   public String toString(){
      return "Start State";
   }
}
```



```java
public class StopState implements State {

   public void doAction(Context context) {
      System.out.println("Player is in stop state");
      context.setState(this);
   }

   public String toString(){
      return "Stop State";
   }
}
```



```
public class Context {
   private State state;

   public Context(){
      state = null;
   }

   public void setState(State state){
      this.state = state;
   }

   public State getState(){
      return state;
   }
}
```

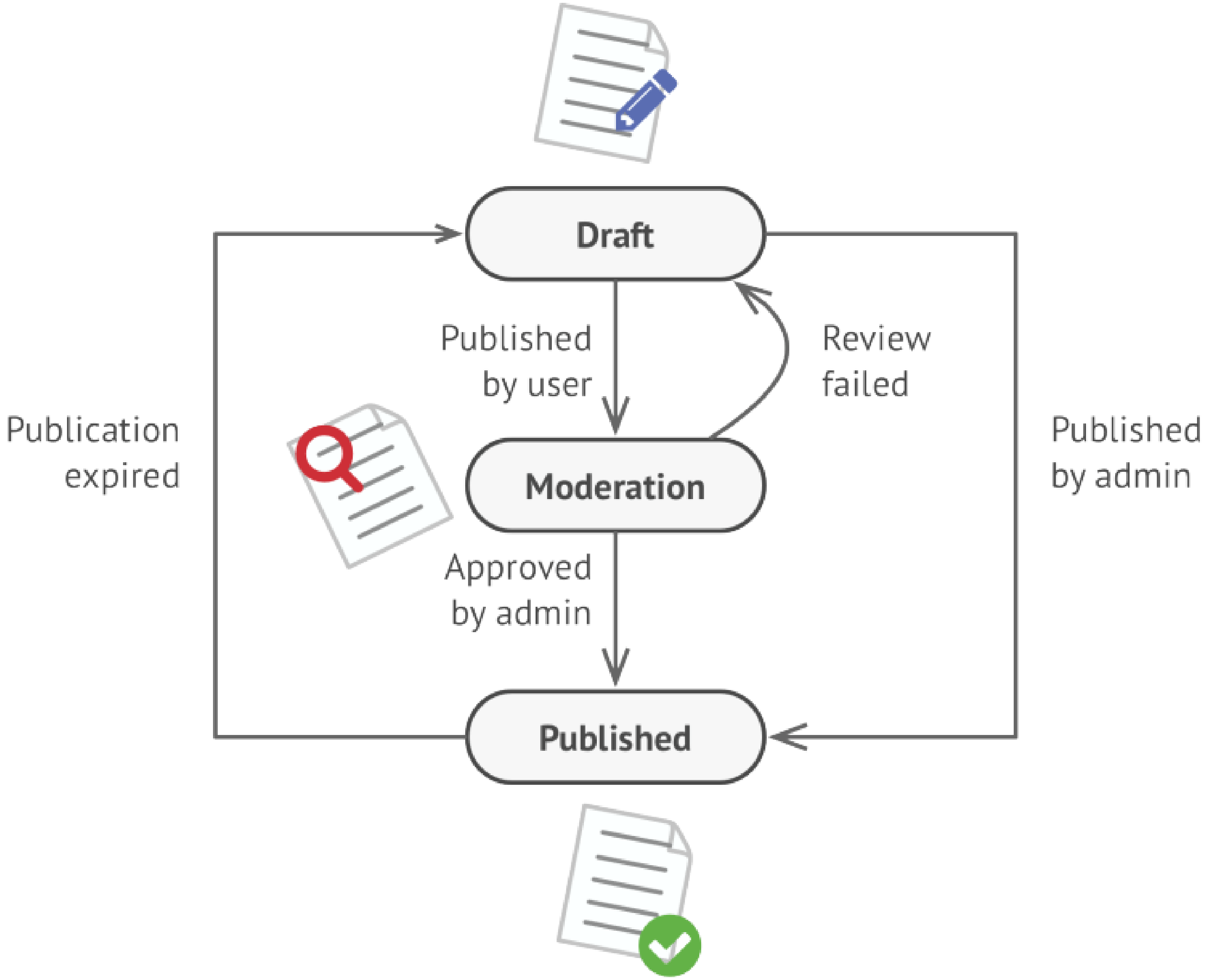


```
public class StatePatternDemo {
   public static void main(String[] args) {
      Context context = new Context();

      StartState startState = new StartState();
      startState.doAction(context);

      System.out.println(context.getState().toString());

      StopState stopState = new StopState();
      stopState.doAction(context);

      System.out.println(context.getState().toString());
   }
}
```

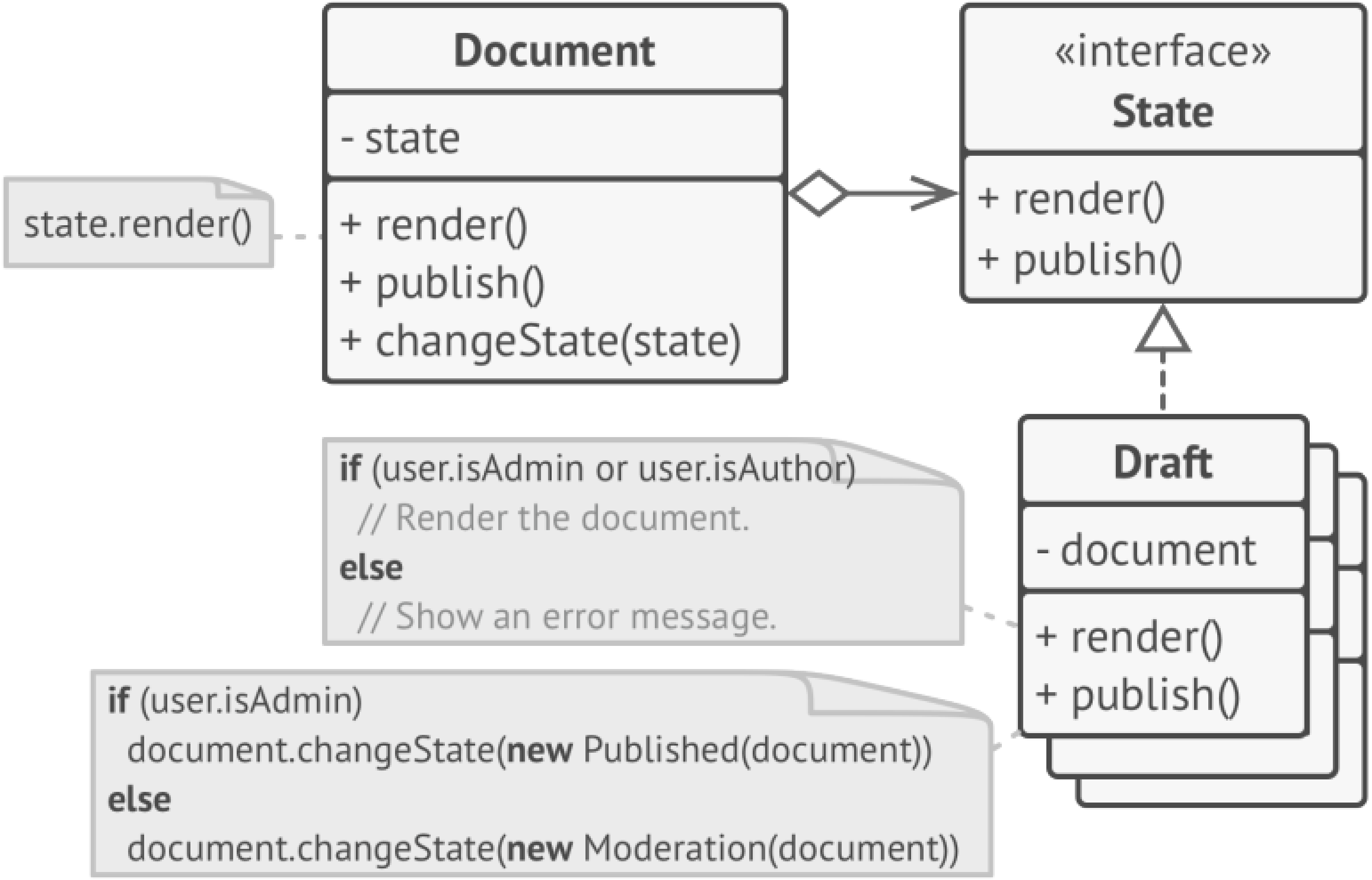
Solution

- The State pattern suggests that you create new classes for all possible states of an object and extract all state-specific behaviors into these classes.
- Instead of implementing all behaviors on its own, the original object, called *context*, stores a reference to one of the state objects that represents its current state, and delegates all the state-related work to that object.
- To transition the context into another state, replace the active state object with another object that represents that new state.

Example



Draft
Published by user
Review failed
Publication expired
Published by admin
Moderation
Approved by admin
Published



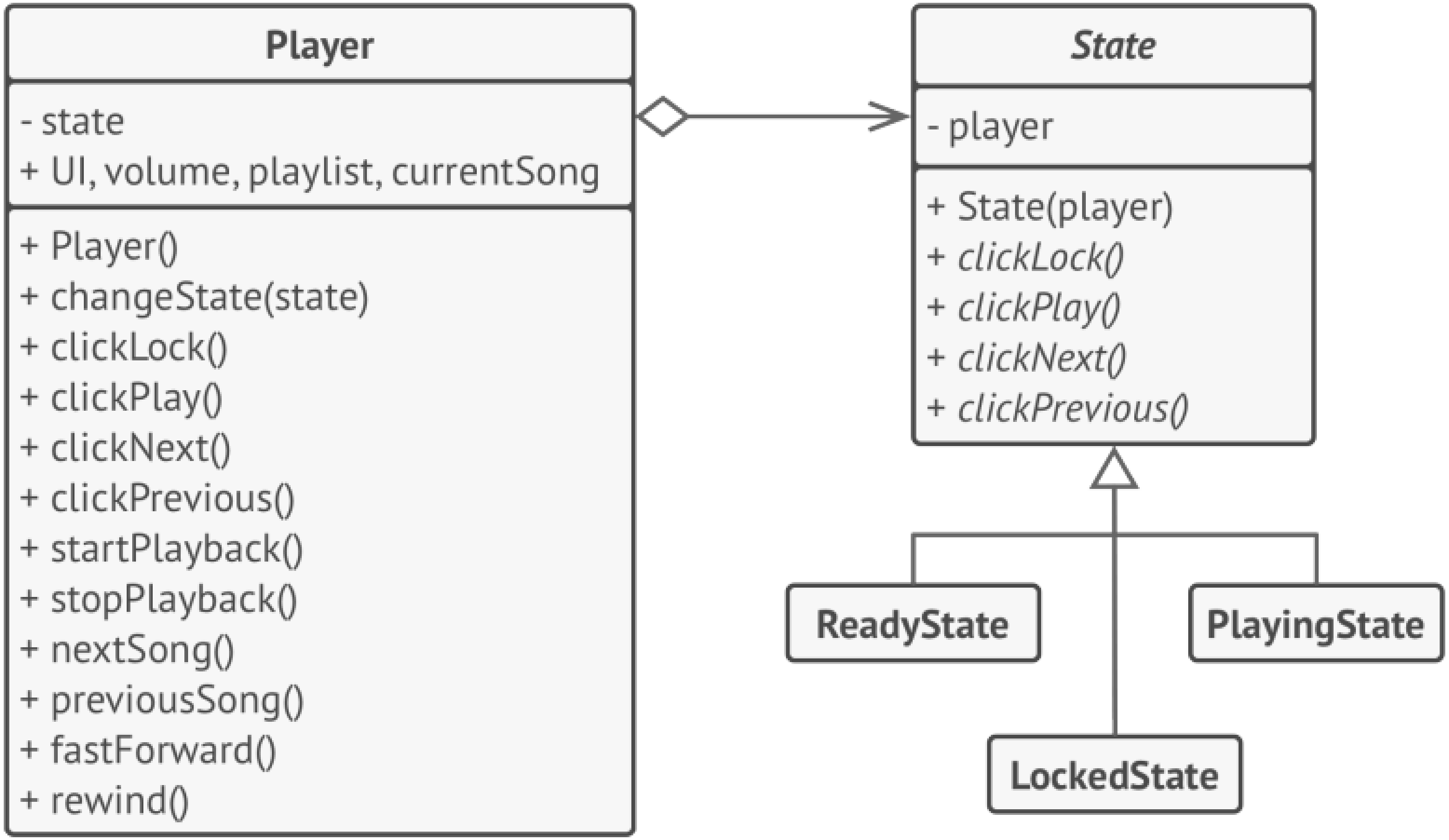
```
state.render()
```

```
if (user.isAdmin or user.isAuthor)
  // Render the document.
else
  // Show an error message.
```

```
if (user.isAdmin)
  document.changeState(new Published(document))
else
  document.changeState(new Moderation(document))
```

Document delegates the work to a state object.

Another Example



Player

- state
+ UI, volume, playlist, currentSong

+ Player()
+ changeState(state)
+ clickLock()
+ clickPlay()
+ clickNext()
+ clickPrevious()
+ startPlayback()
+ stopPlayback()
+ nextSong()
+ previousSong()
+ fastForward()
+ rewind()

State

- player

+ State(player)
+ *clickLock()*
+ *clickPlay()*
+ *clickNext()*
+ *clickPrevious()*

ReadyState

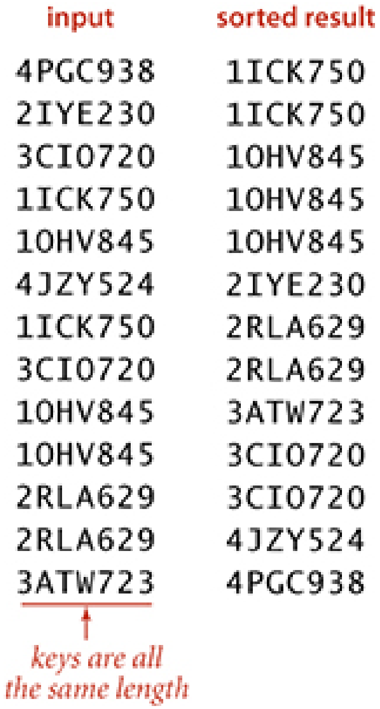
PlayingState

LockedState

Consequences

- Localizes the state specific behavior
- Makes state transitions explicit

Motivation: Strategy



input	sorted result
4PGC938	1ICK750
2IYE230	1ICK750
3CIO720	1OHV845
1ICK750	1OHV845
1OHV845	1OHV845
4JZY524	2IYE230
1ICK750	2RLA629
3CIO720	2RLA629
1OHV845	3ATW723
1OHV845	3CIO720
2RLA629	3CIO720
2RLA629	4JZY524
3ATW723	4PGC938

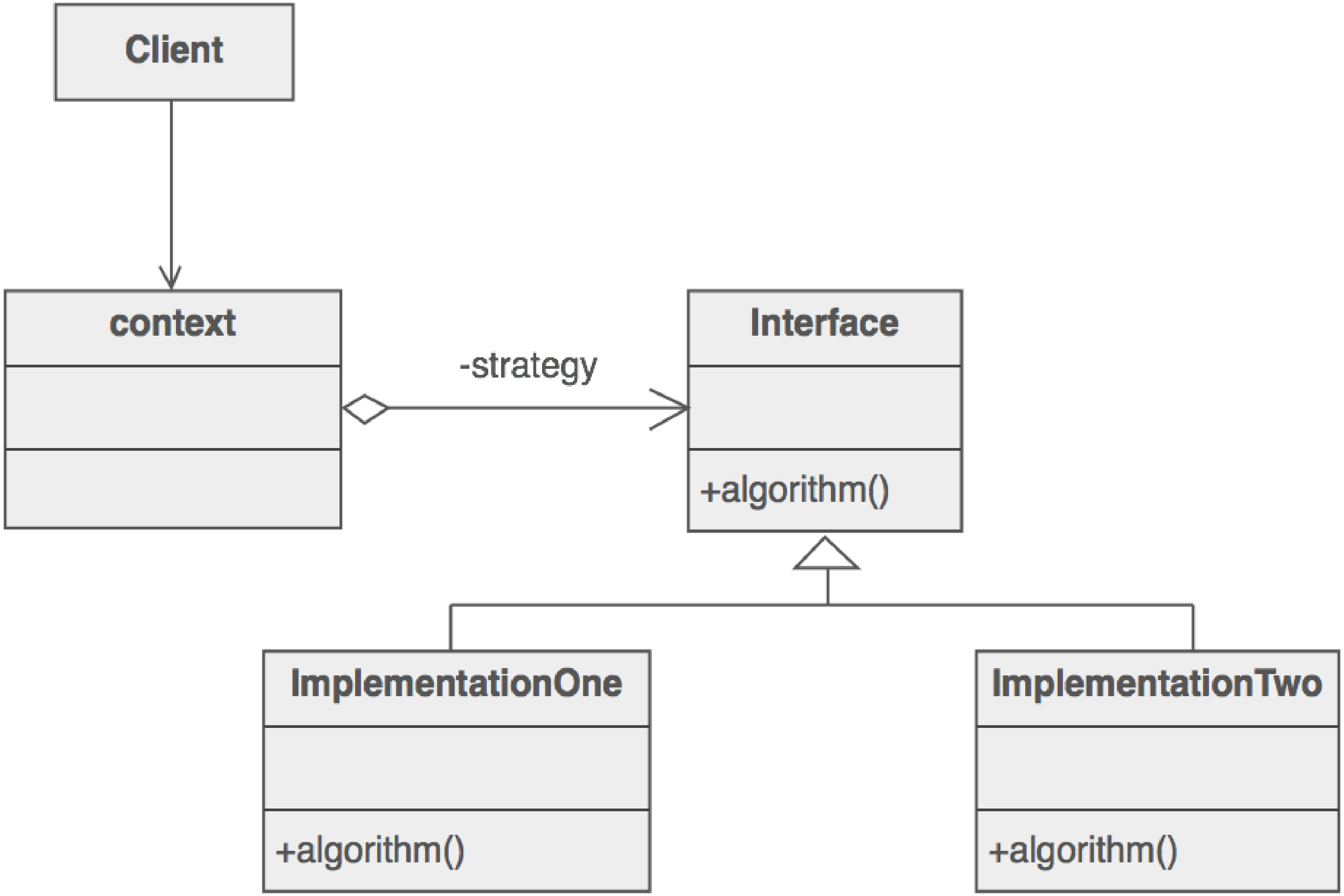
keys are all the same length

- Quick sort
- Merge sort
- Insertion sort
- Bubble sort
- Radix sort
- Heap sort
- Bucket sort
- ..

Strategy

Intent	• Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
Problem	• Capture the abstraction in an interface, bury implementation details in derived classes.

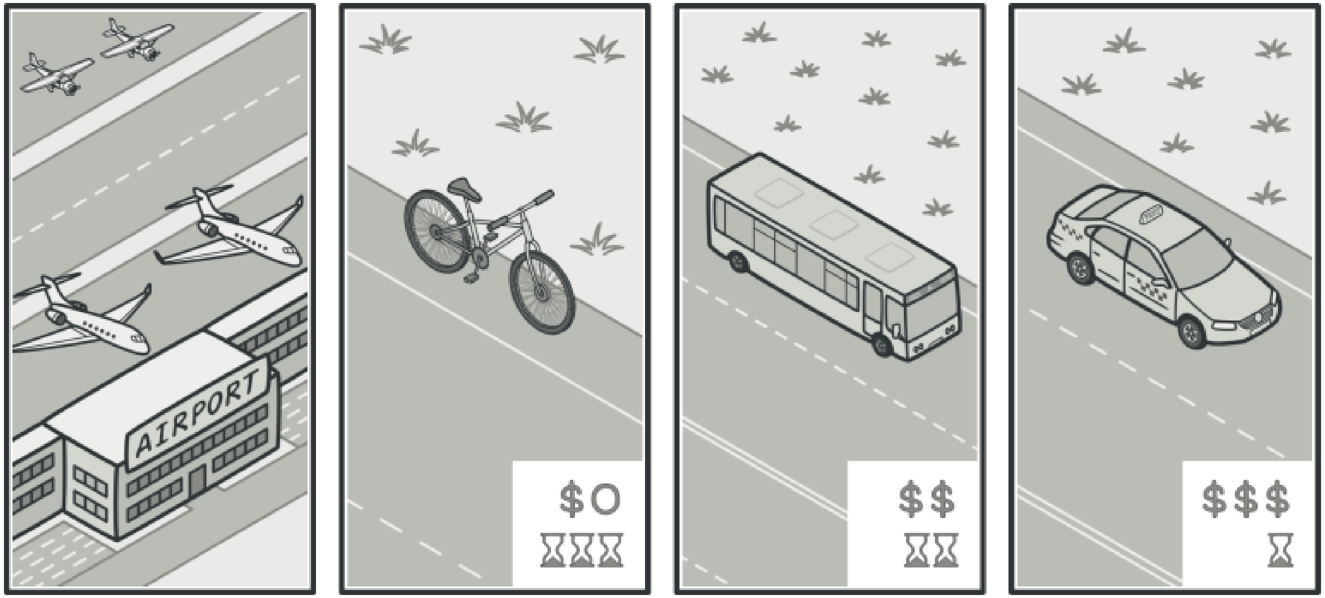
Solution


Client
context
-strategy
Interface
+algorithm()
ImplementationOne
+algorithm()
ImplementationTwo
+algorithm()

How It Works

- 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*.
- The original class, called *context*, must have a field for storing a reference to one of the strategies.
- The context delegates the work to a linked strategy object instead of executing it on its own.
- The context isn't responsible for selecting an appropriate algorithm for the job. Instead, the client passes the desired strategy to the context.
- Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.

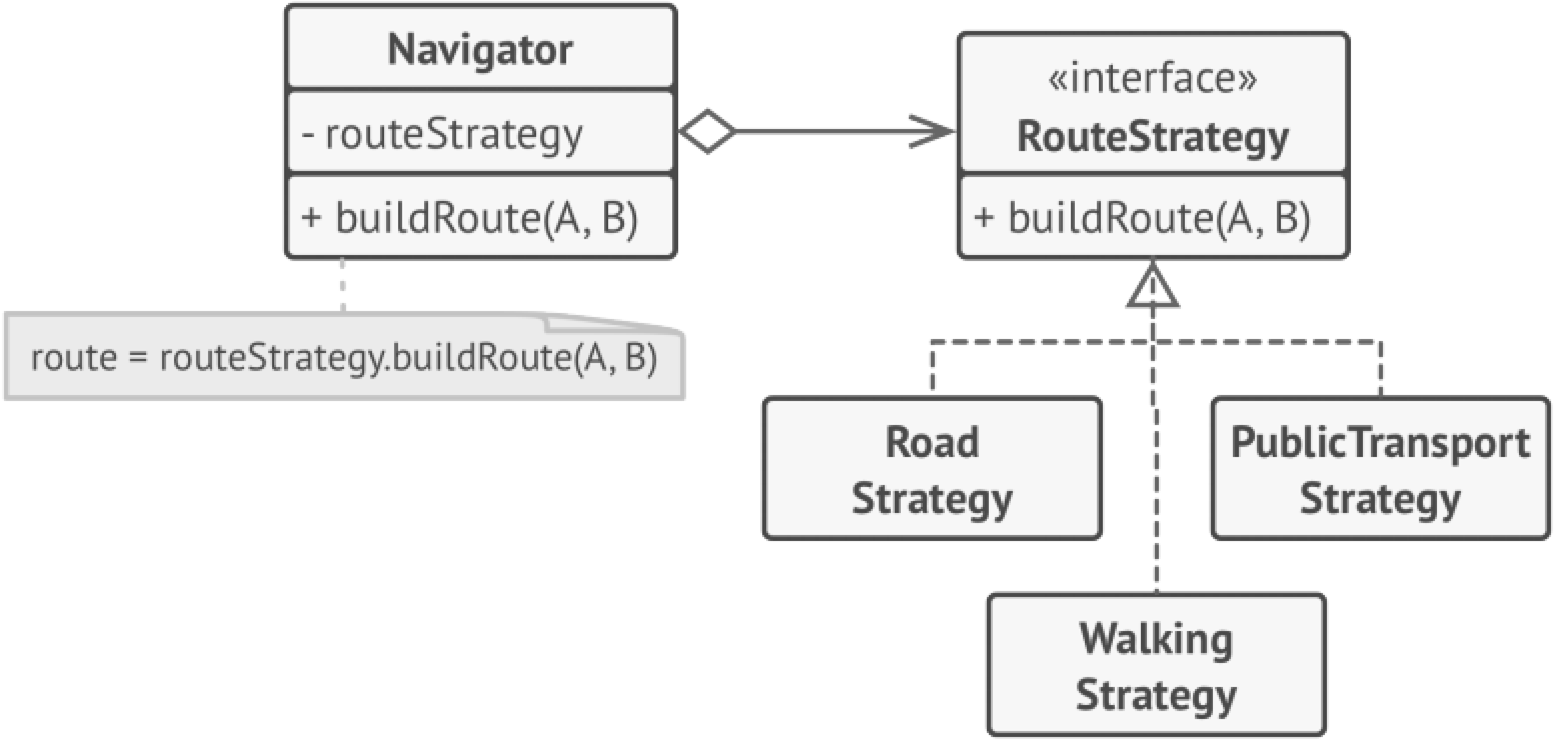
Another Example



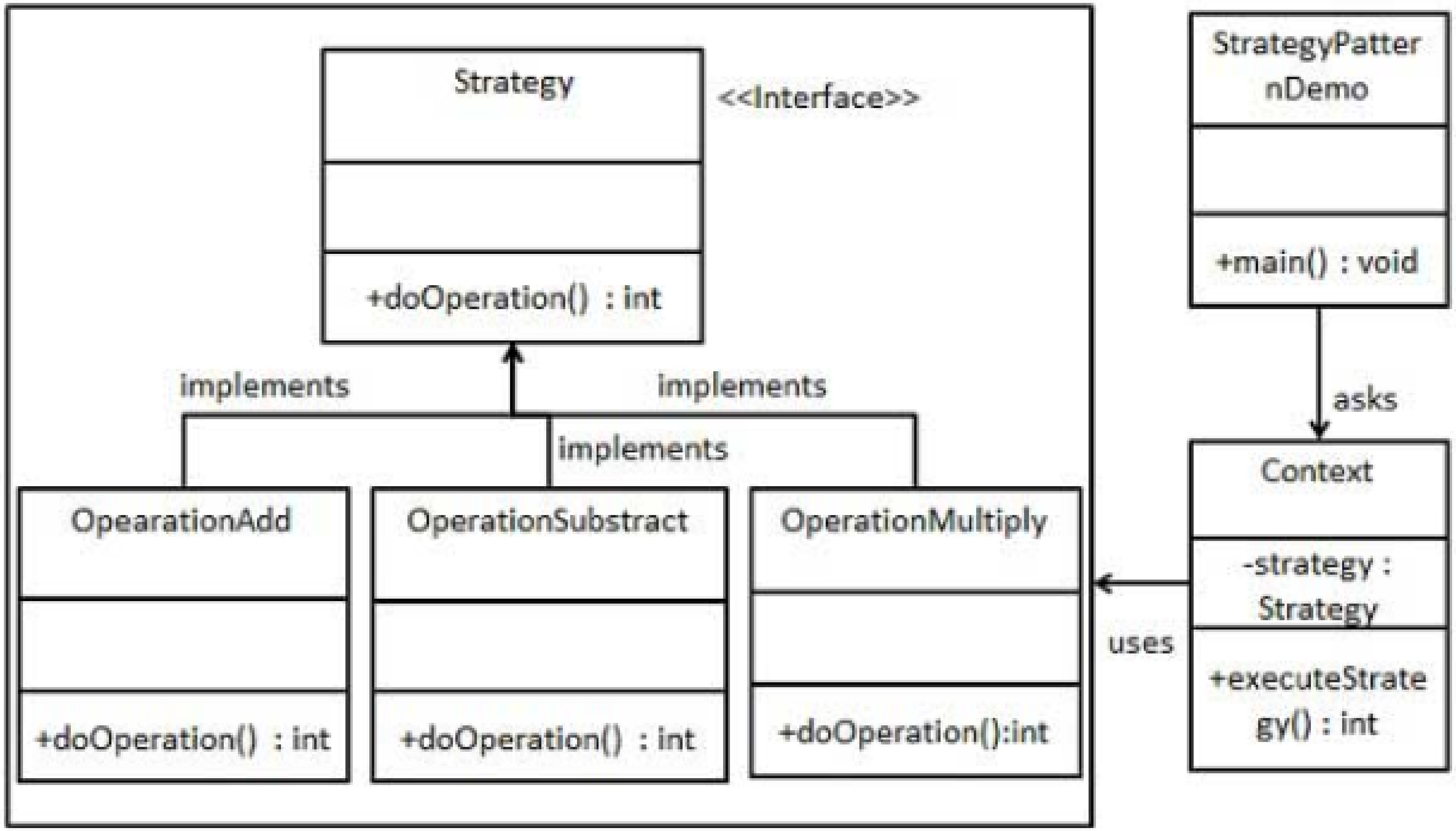
Various strategies for getting to the airport.

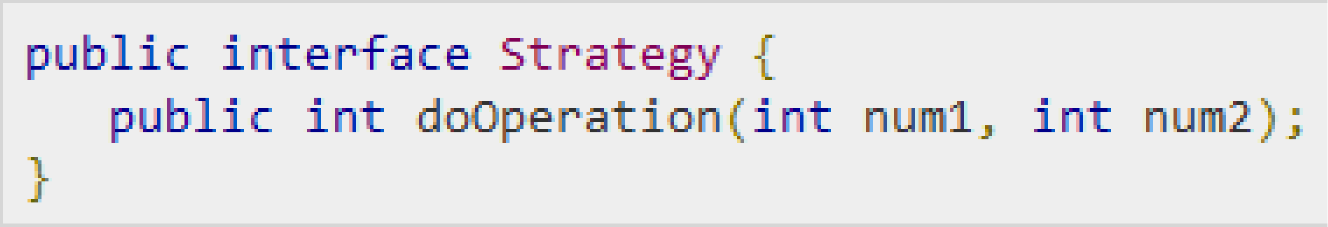
Imagine that you have to get to the airport. You can catch a bus, order a cab, or get on your bicycle. These are your transportation strategies. You can pick one of the strategies depending on factors such as budget or time constraints.

Another Example



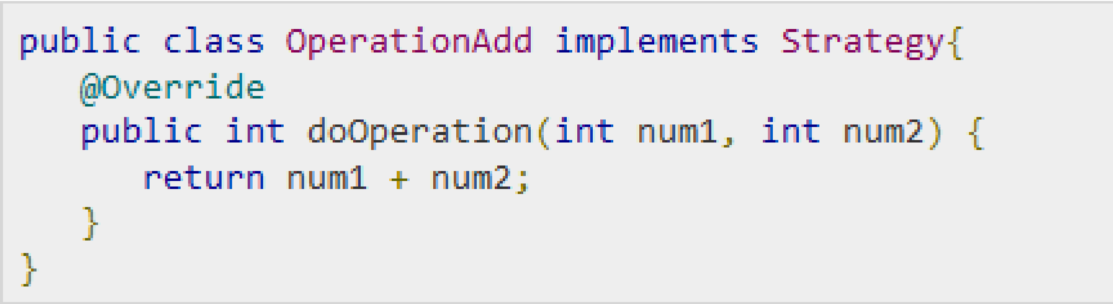
- In the navigation app, each routing algorithm can be extracted to its own class with a single buildRoute method. The method accepts an origin and destination and returns a collection of the route's checkpoints.





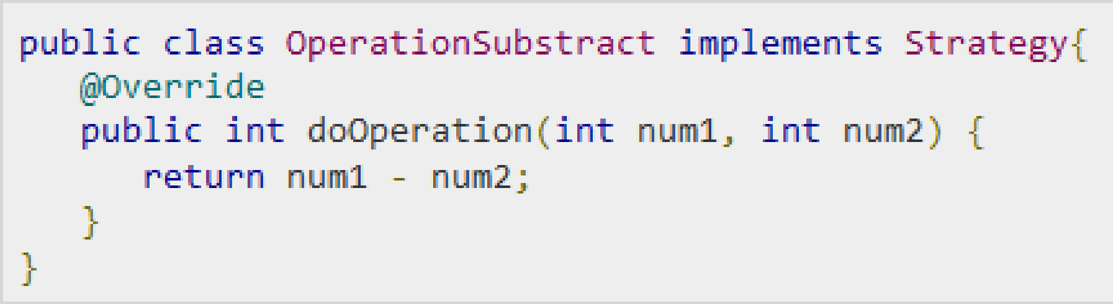
```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

OperationAdd.java



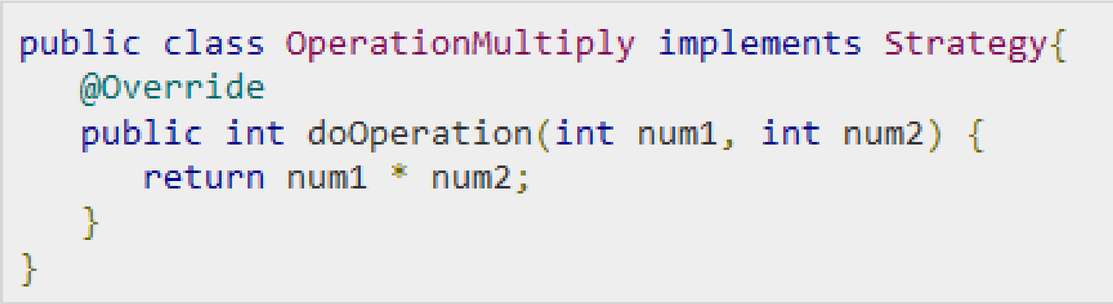
```java
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

OperationSubstract.java

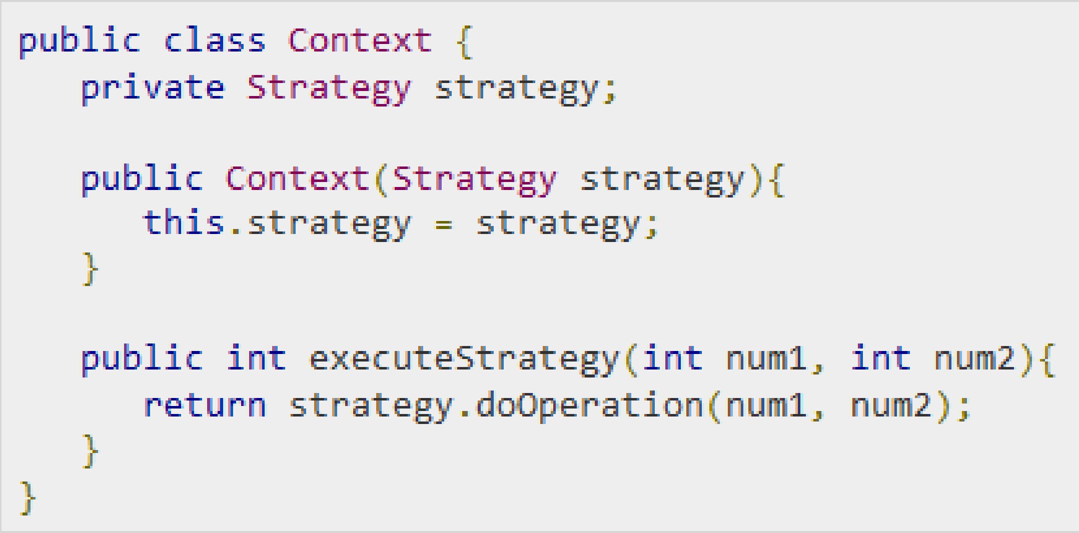


```java
public class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
```

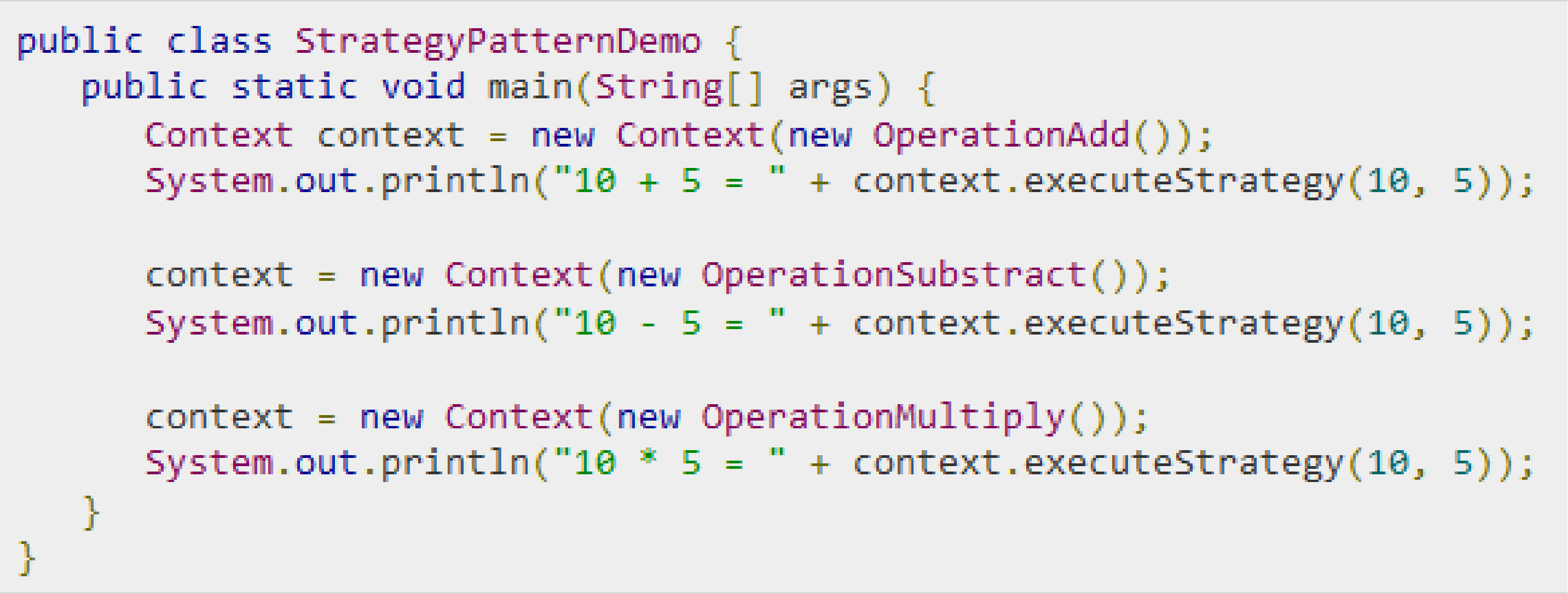
OperationMultiply.java



```java
public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```



```
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

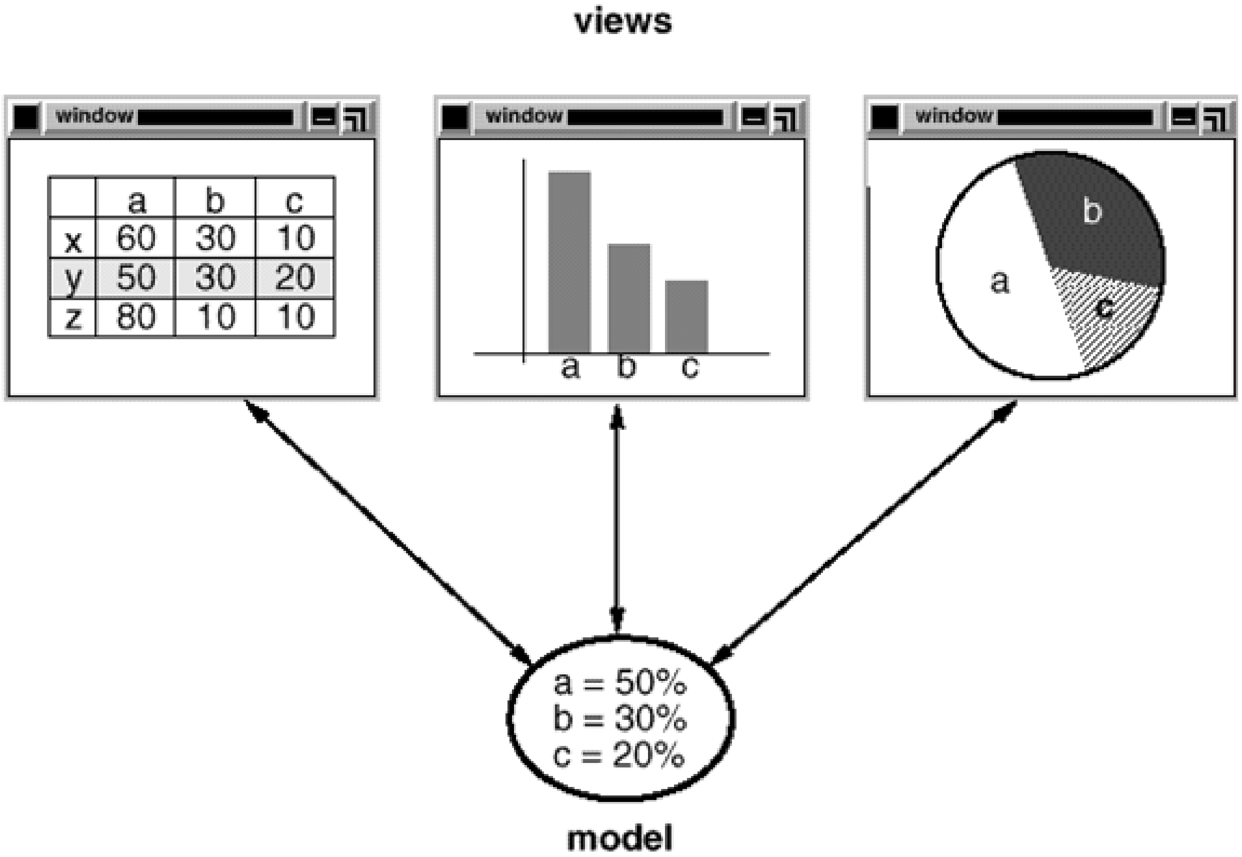


```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```

Consequences

- Families of related algorithms
- Eliminate conditional statements
- Client must be aware of different strategies

Motivation: Observer

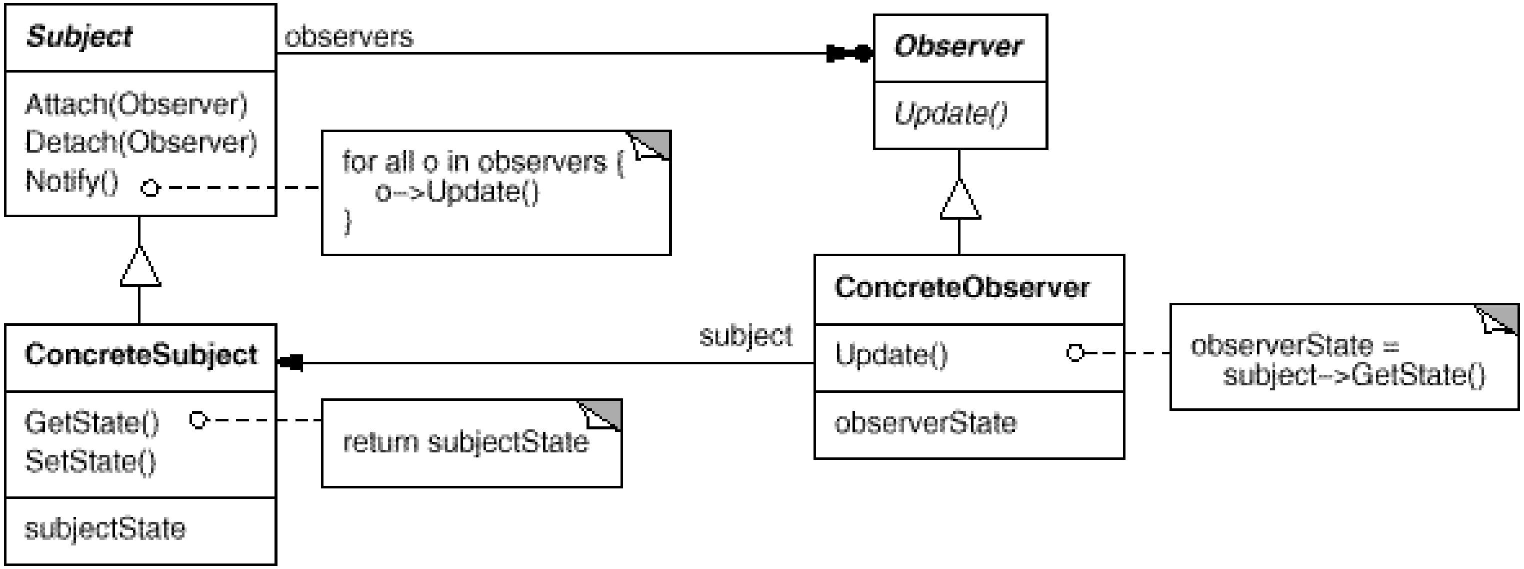


	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

Observer

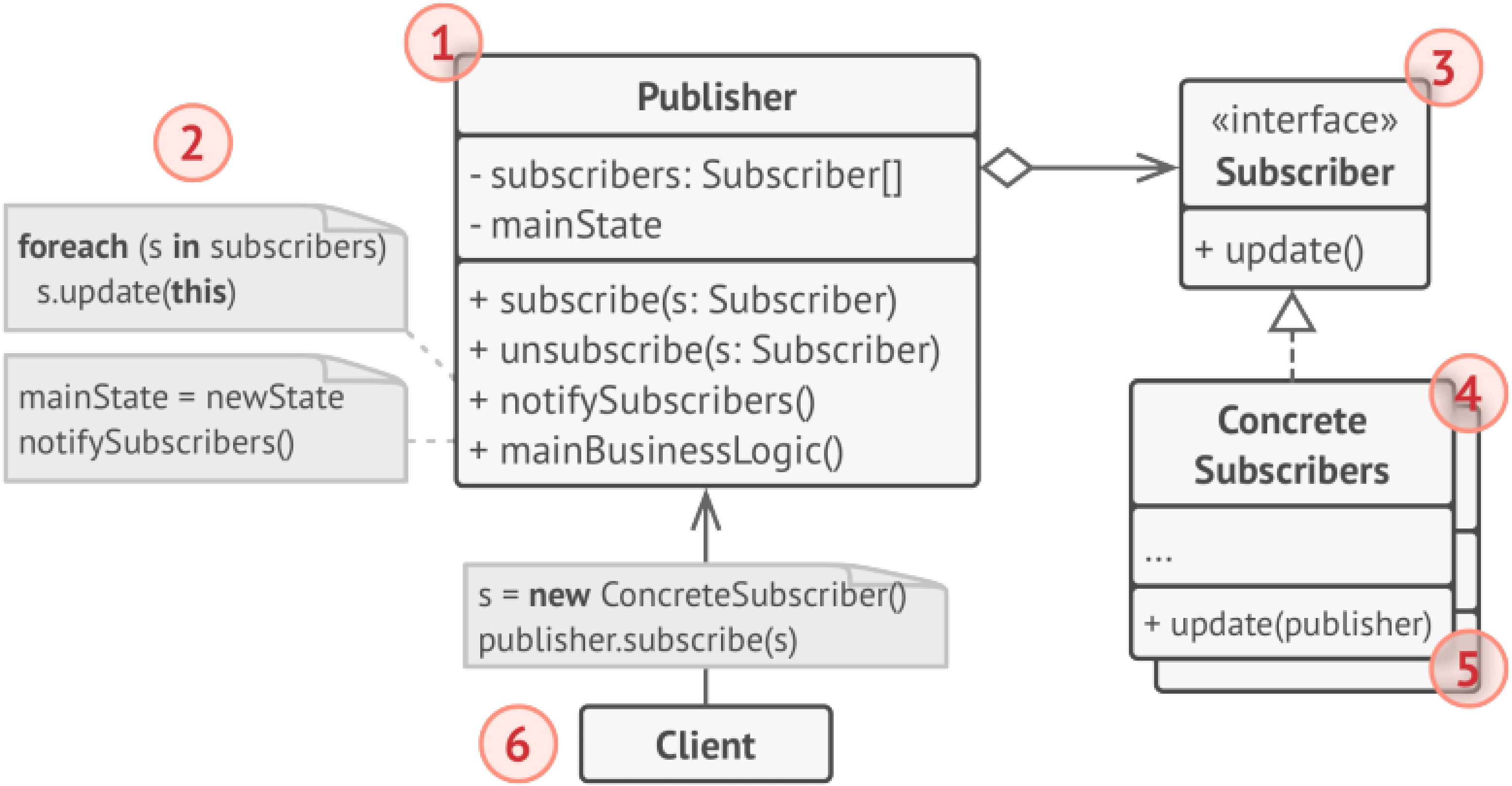
Intent	• Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Problem	• Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.

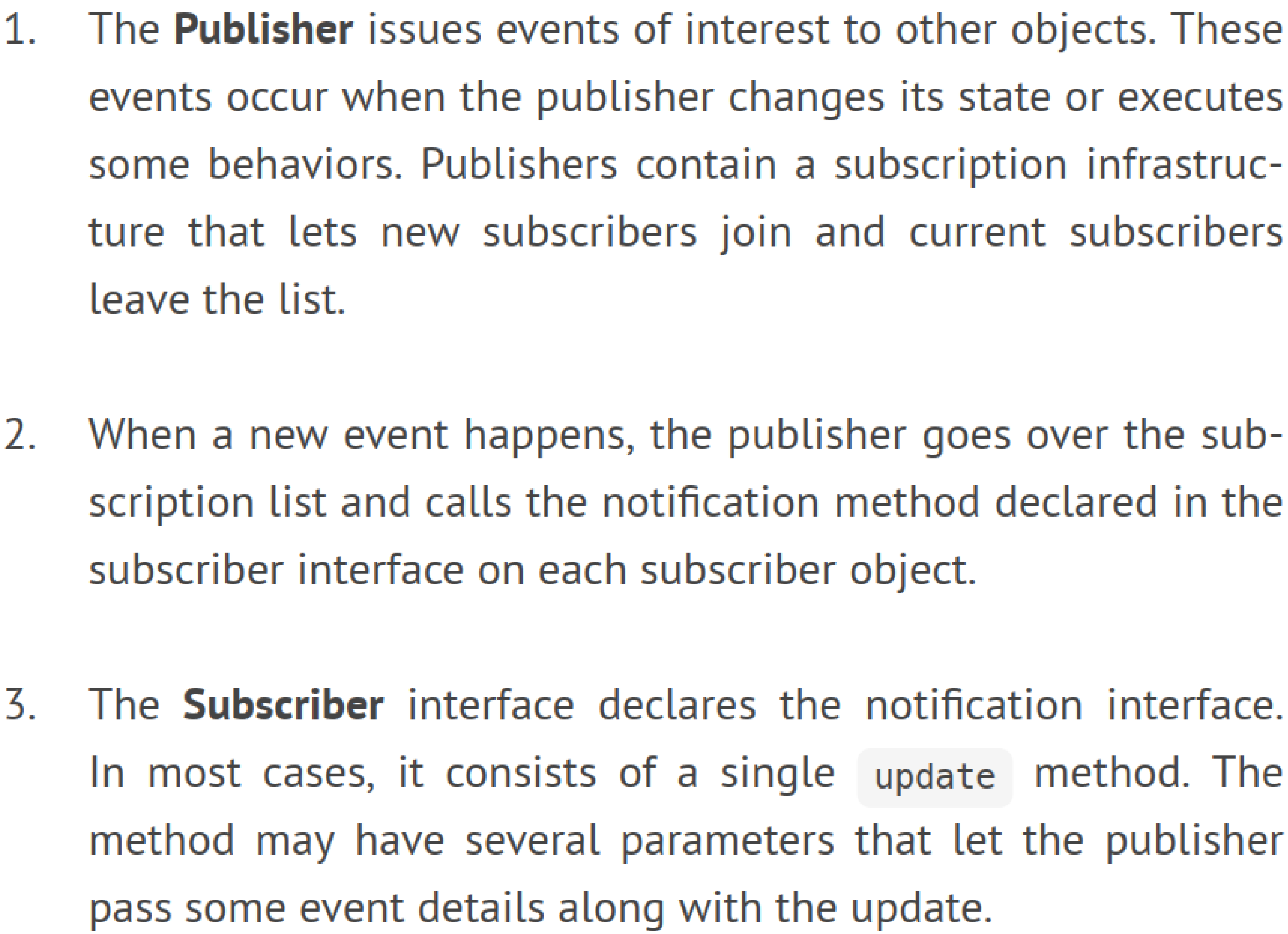
Solution



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

Example

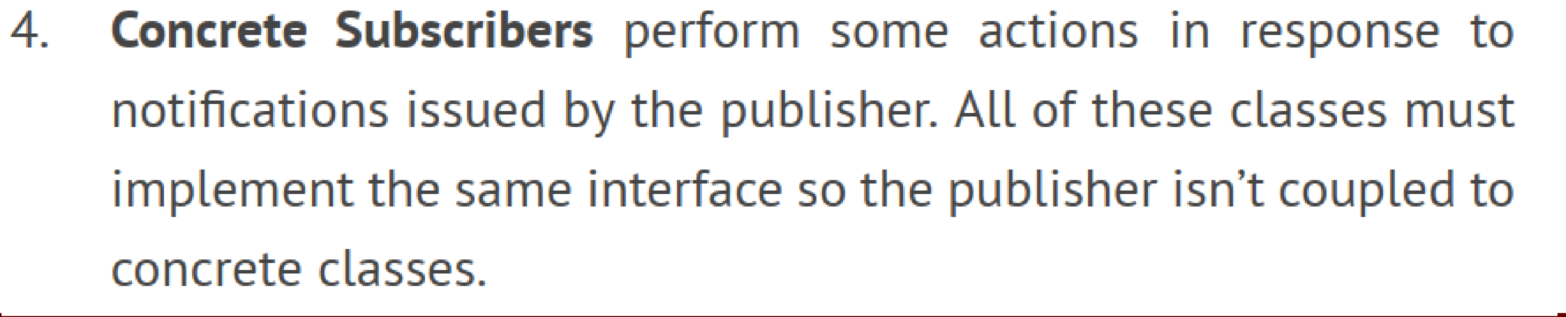

1
Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
2
foreach (s in subscribers)
s.update(this)
mainState = newState
notifySubscribers()
3
«interface»
Subscriber
+ update()
4
Concrete
Subscribers
...
+ update(publisher)
5
s = new ConcreteSubscriber()
publisher.subscribe(s)
6
Client



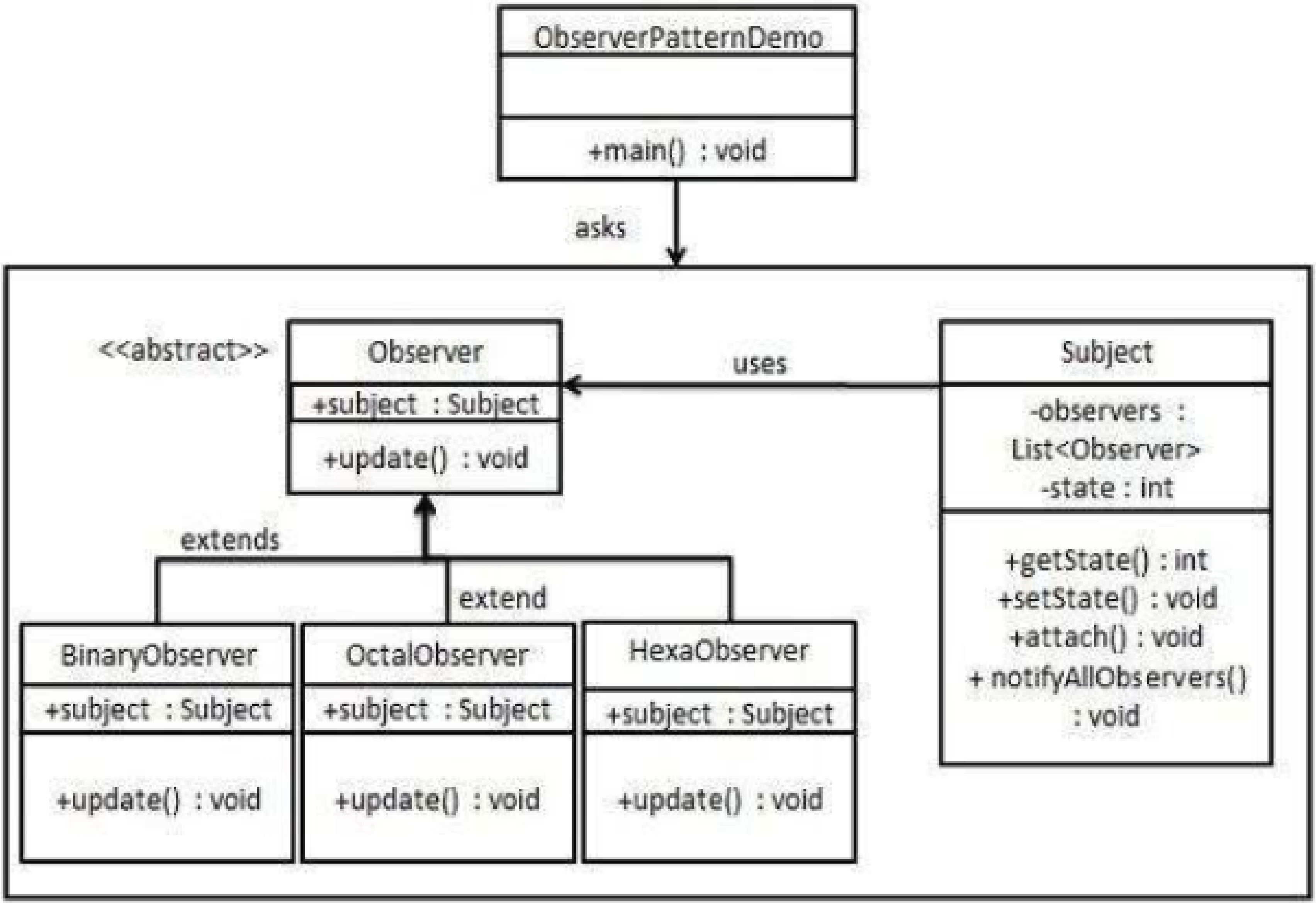
1. The **Publisher** issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list.

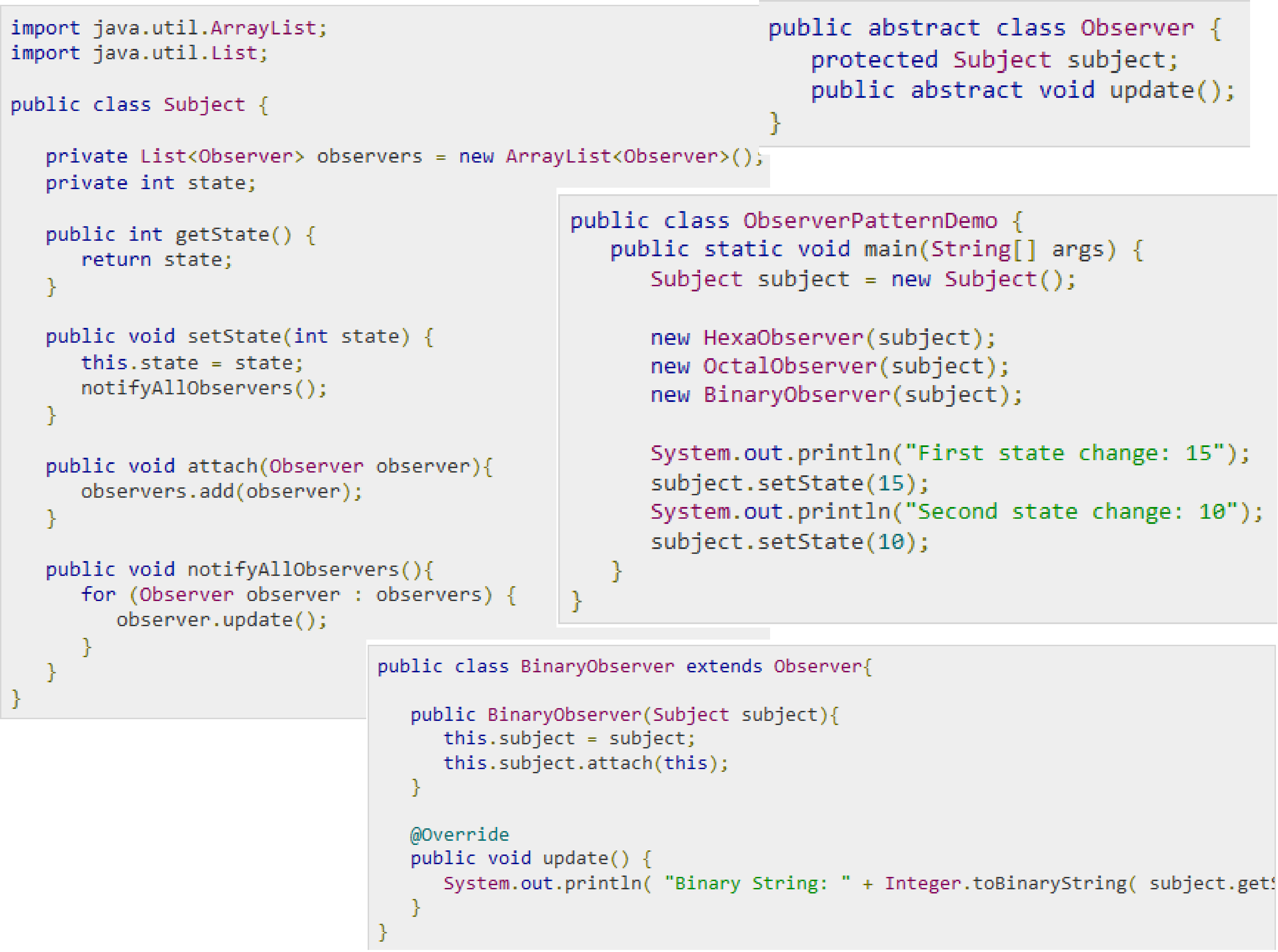
2. When a new event happens, the publisher goes over the subscription list and calls the notification method declared in the subscriber interface on each subscriber object.

3. The **Subscriber** interface declares the notification interface. In most cases, it consists of a single `update` method. The method may have several parameters that let the publisher pass some event details along with the update.



4. **Concrete Subscribers** perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes.


ObserverPatternDemo
+main() : void
asks
<<abstract>>
Observer
+subject : Subject
+update() : void
uses
Subject
-observers : List<Observer>
-state : int
+getState() : int
+setState() : void
+attach() : void
+ notifyAllObservers() : void
extends
extend
BinaryObserver
+subject : Subject
+update() : void
OctalObserver
+subject : Subject
+update() : void
HexaObserver
+subject : Subject
+update() : void



```java
import java.util.ArrayList;
import java.util.List;

public class Subject {

   private List<Observer> observers = new ArrayList<Observer>();
   private int state;

   public int getState() {
      return state;
   }

   public void setState(int state) {
      this.state = state;
      notifyAllObservers();
   }

   public void attach(Observer observer){
      observers.add(observer);
   }

   public void notifyAllObservers(){
      for (Observer observer : observers) {
         observer.update();
      }
   }
}
```

```java
public abstract class Observer {
   protected Subject subject;
   public abstract void update();
}
```

```java
public class ObserverPatternDemo {
   public static void main(String[] args) {
      Subject subject = new Subject();

      new HexaObserver(subject);
      new OctalObserver(subject);
      new BinaryObserver(subject);

      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}
```

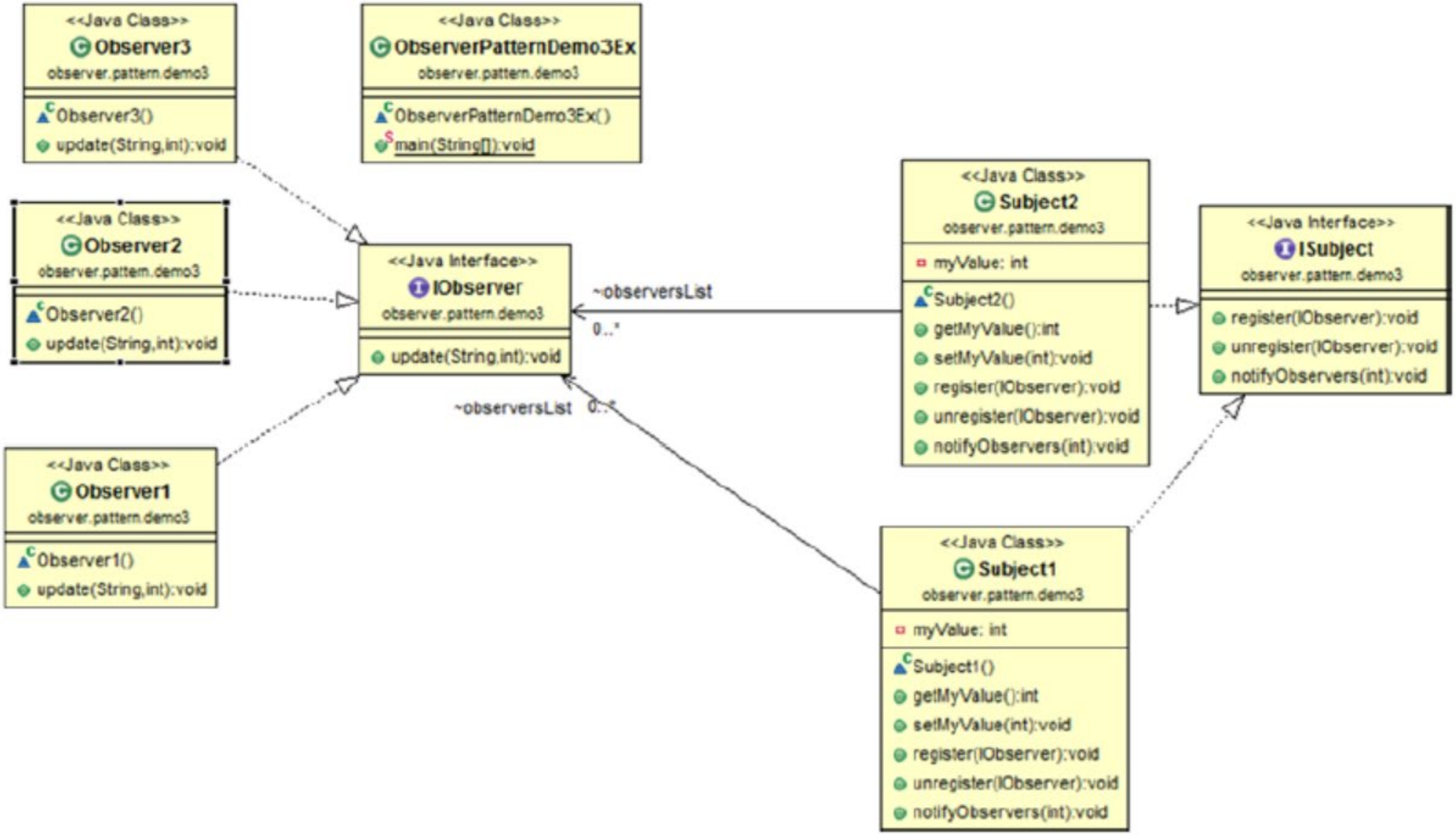
```java
public class BinaryObserver extends Observer{

   public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Binary String: " + Integer.toBinaryString( subject.get
   }
}
```

Consequences

- Decoupling subject and observer

- Support broadcast communication

Many observers, many subjects



<<Java Class>>
Observer3
observer.pattern.demo3
Observer3()
update(String,int):void

<<Java Class>>
ObserverPatternDemo3Ex
observer.pattern.demo3
ObserverPatternDemo3Ex()
main(String[]):void

<<Java Class>>
Observer2
observer.pattern.demo3
Observer2()
update(String,int):void

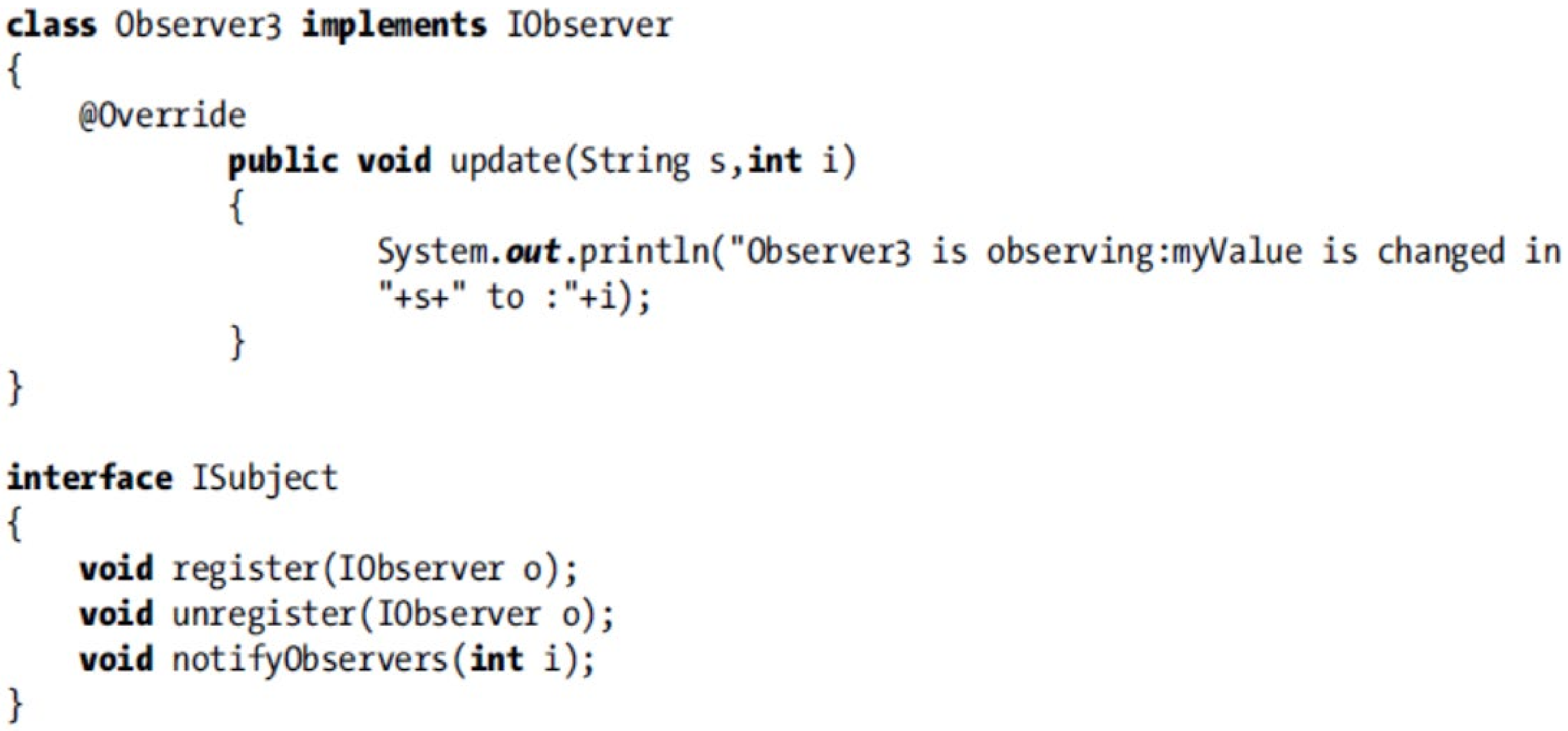
<<Java Interface>>
IObserver
observer.pattern.demo3
update(String,int):void

<<Java Class>>
Observer1
observer.pattern.demo3
Observer1()
update(String,int):void

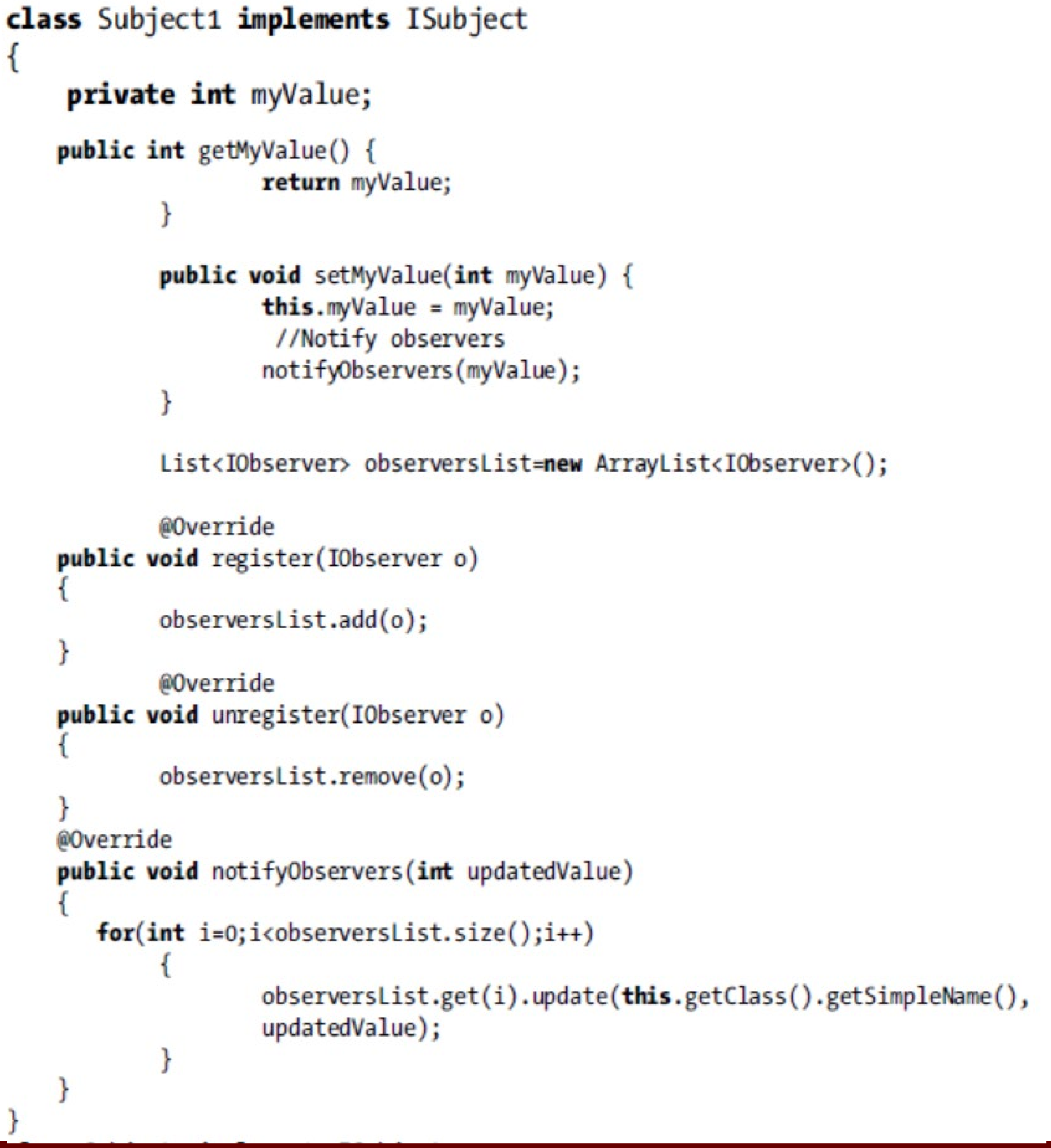
<<Java Class>>
Subject2
observer.pattern.demo3
myValue: int
Subject2()
getMyValue():int
setMyValue(int):void
register(IObserver):void
unregister(IObserver):void
notifyObservers(int):void

<<Java Interface>>
ISubject
observer.pattern.demo3
register(IObserver):void
unregister(IObserver):void
notifyObservers(int):void

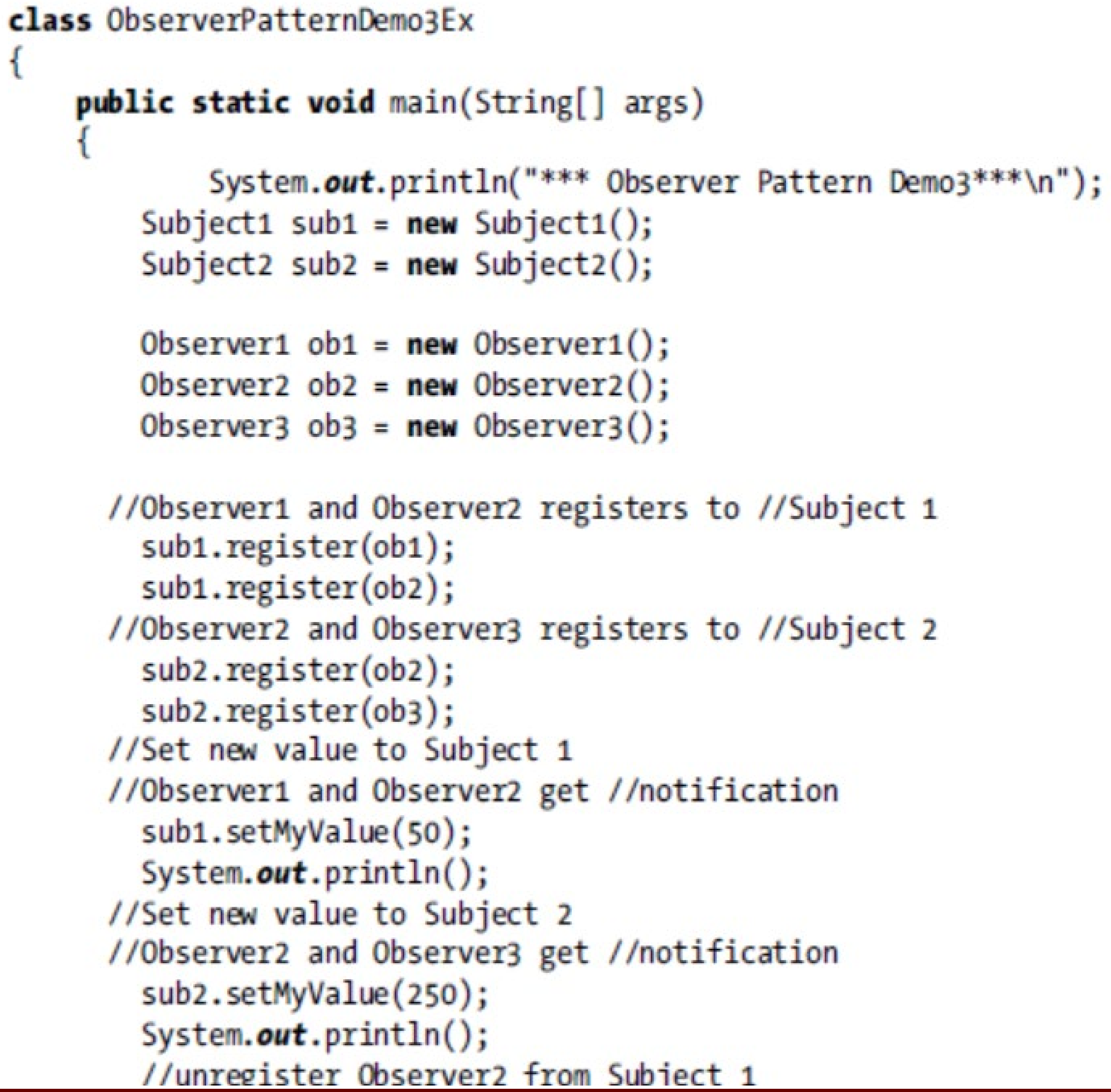
<<Java Class>>
Subject1
observer.pattern.demo3
myValue: int
Subject1()
getMyValue():int
setMyValue(int):void
register(IObserver):void
unregister(IObserver):void
notifyObservers(int):void



```
class Observer3 implements IObserver
{
    @Override
        public void update(String s,int i)
        {
            System.out.println("Observer3 is observing:myValue is changed in
            "+s+" to :"+i);
        }
}

interface ISubject
{
    void register(IObserver o);
    void unregister(IObserver o);
    void notifyObservers(int i);
}
```

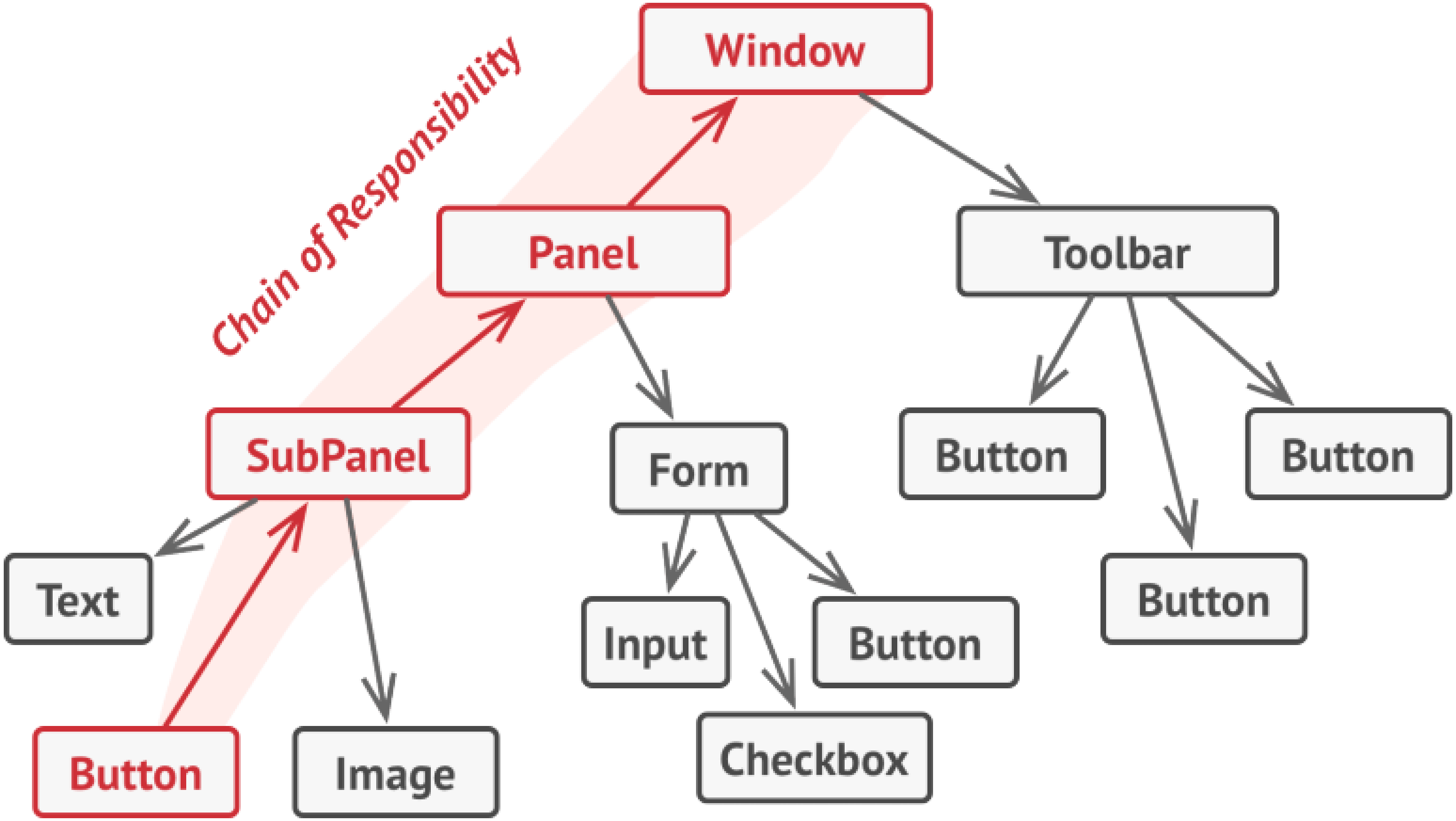


```
class Subject1 implements ISubject
{
    private int myValue;
   public int getMyValue() {
                 return myValue;
          }

          public void setMyValue(int myValue) {
                 this.myValue = myValue;
                  //Notify observers
                 notifyObservers(myValue);
          }

          List<IObserver> observersList=new ArrayList<IObserver>();

          @Override
   public void register(IObserver o)
   {
          observersList.add(o);
   }
          @Override
   public void unregister(IObserver o)
   {
          observersList.remove(o);
   }
   @Override
   public void notifyObservers(int updatedValue)
   {
      for(int i=0;i<observersList.size();i++)
          {
                 observersList.get(i).update(this.getClass().getSimpleName(),
                 updatedValue);
          }
   }
}
```



```
class ObserverPatternDemo3Ex
{
    public static void main(String[] args)
    {
            System.out.println("*** Observer Pattern Demo3***\n");
        Subject1 sub1 = new Subject1();
        Subject2 sub2 = new Subject2();

        Observer1 ob1 = new Observer1();
        Observer2 ob2 = new Observer2();
        Observer3 ob3 = new Observer3();

      //Observer1 and Observer2 registers to //Subject 1
        sub1.register(ob1);
        sub1.register(ob2);
      //Observer2 and Observer3 registers to //Subject 2
        sub2.register(ob2);
        sub2.register(ob3);
      //Set new value to Subject 1
      //Observer1 and Observer2 get //notification
        sub1.setMyValue(50);
        System.out.println();
      //Set new value to Subject 2
      //Observer2 and Observer3 get //notification
        sub2.setMyValue(250);
        System.out.println();
        //unregister Observer2 from Subject 1
```

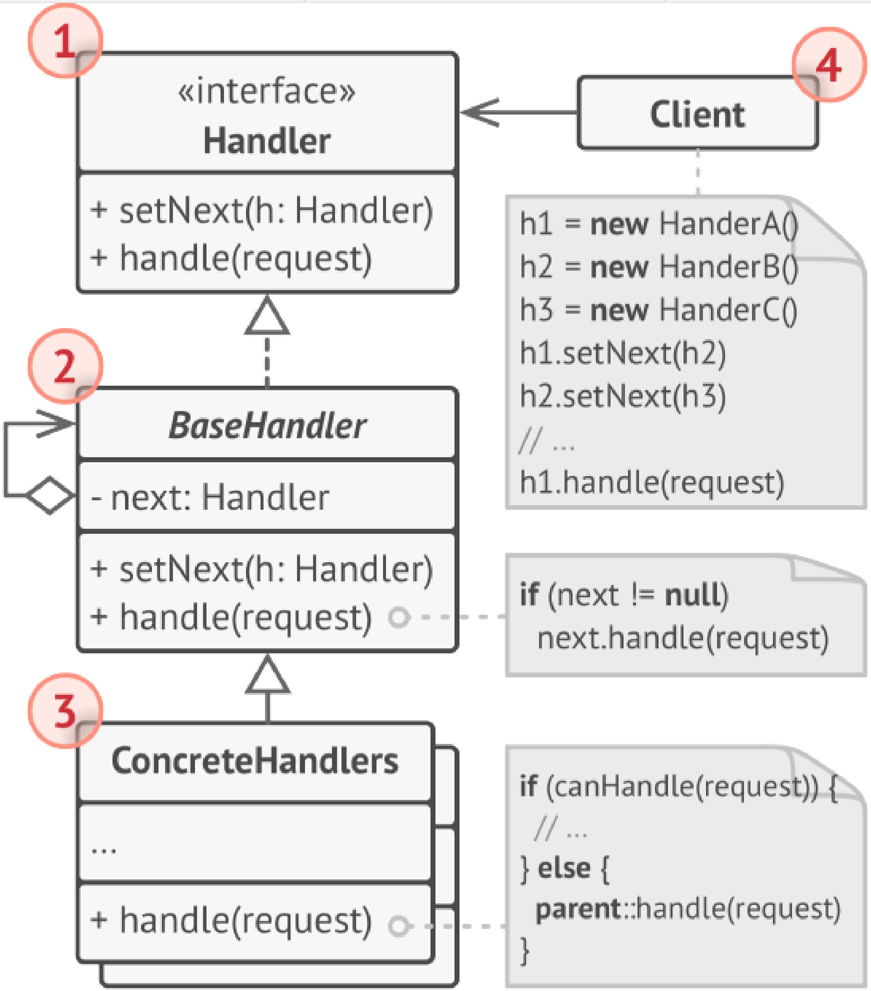
Chain of Responsibility



A chain can be formed from a branch of an object tree.

Chain of Responsibility

- When a user clicks a button, the event propagates through the chain of GUI elements that starts with the button, goes along its containers (like forms or panels), and ends up with the main application window.
- The event is processed by the first element in the chain that’s capable of handling it.
- It’s crucial that all handler classes implement the same interface. Each concrete handler should only care about the following one having the execute method.
- This way you can compose chains at runtime, using various handlers without coupling your code to their concrete classes.

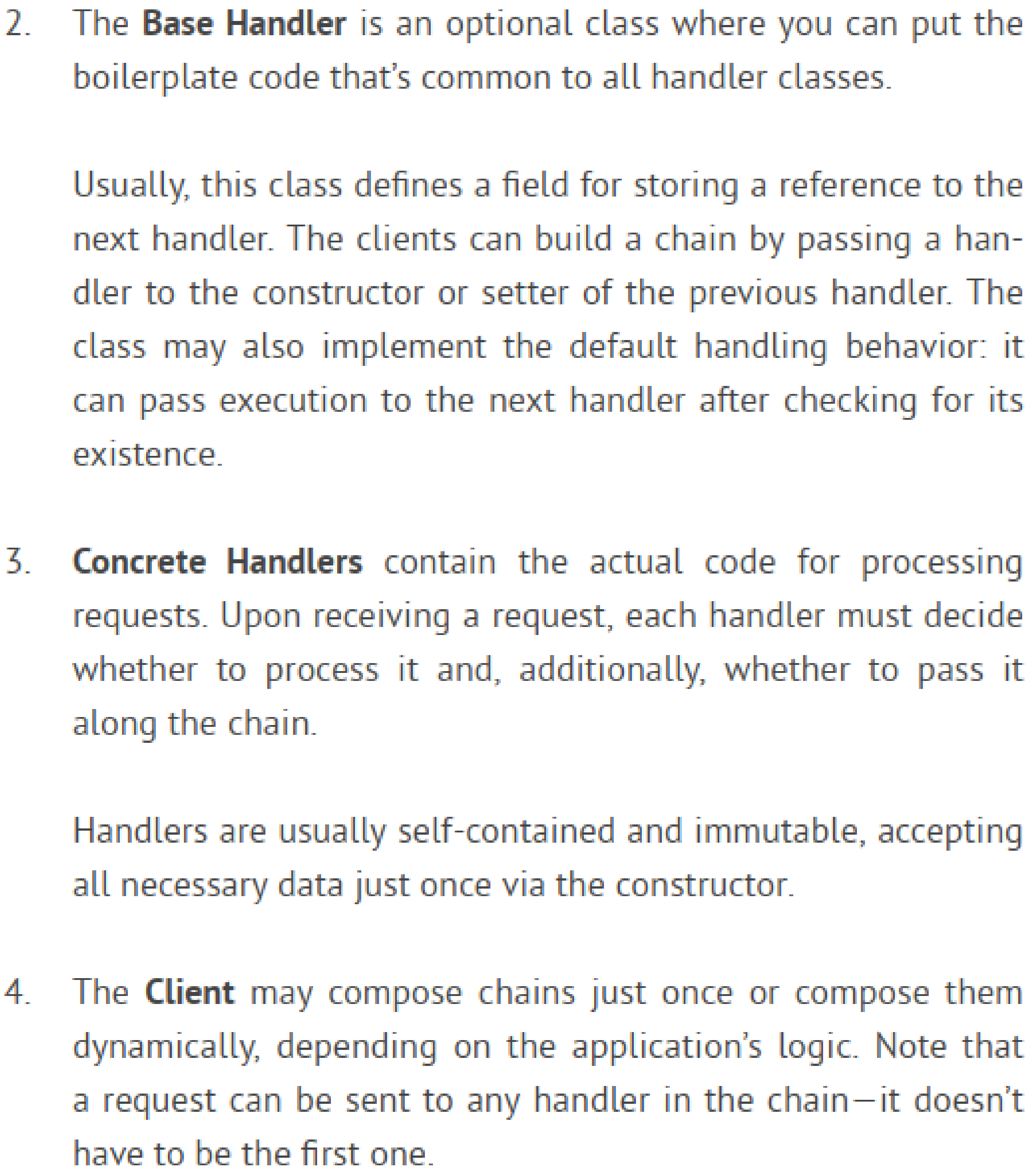


```
h1 = new HanderA()
h2 = new HanderB()
h3 = new HanderC()
h1.setNext(h2)
h2.setNext(h3)
// ...
h1.handle(request)
```

```
if (next != null)
  next.handle(request)
```

```
if (canHandle(request)) {
  // ...
} else {
  parent::handle(request)
}
```

1. The **Handler** declares the interface, common for all concrete handlers. It usually contains just a single method for handling requests, but sometimes it may also have another method for setting the next handler on the chain.



2. The **Base Handler** is an optional class where you can put the boilerplate code that's common to all handler classes.

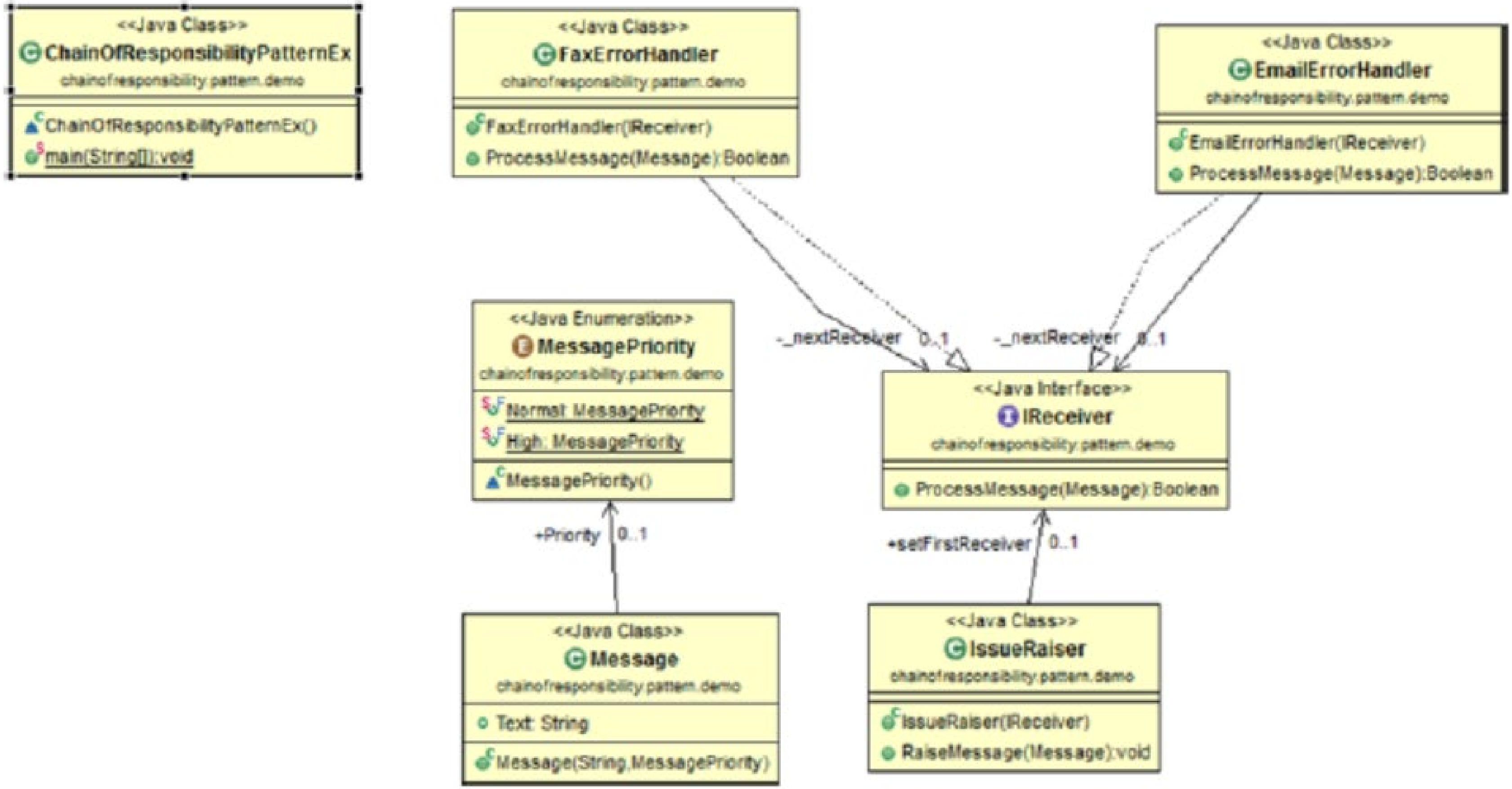
   Usually, this class defines a field for storing a reference to the next handler. The clients can build a chain by passing a handler to the constructor or setter of the previous handler. The class may also implement the default handling behavior: it can pass execution to the next handler after checking for its existence.

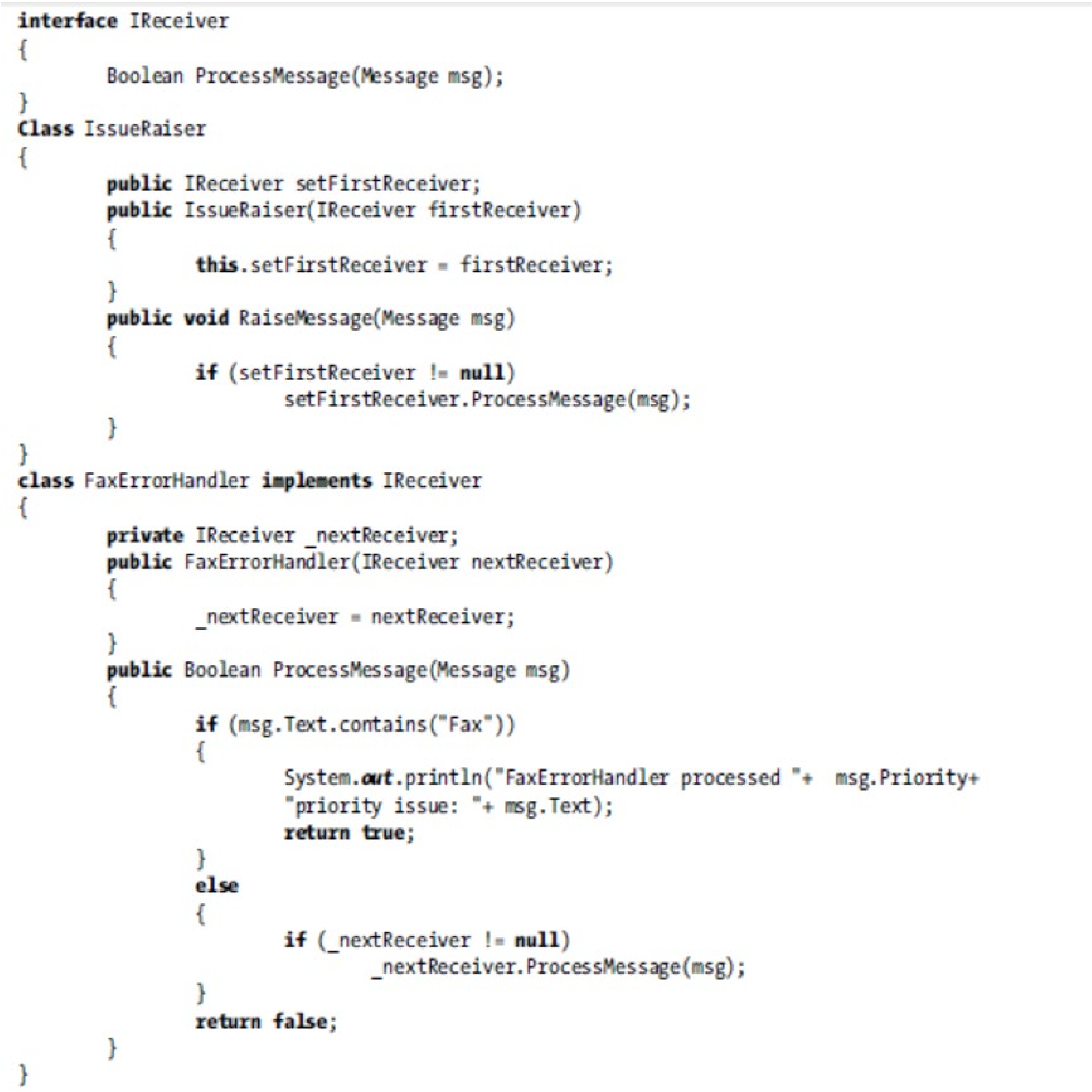
3. **Concrete Handlers** contain the actual code for processing requests. Upon receiving a request, each handler must decide whether to process it and, additionally, whether to pass it along the chain.

   Handlers are usually self-contained and immutable, accepting all necessary data just once via the constructor.

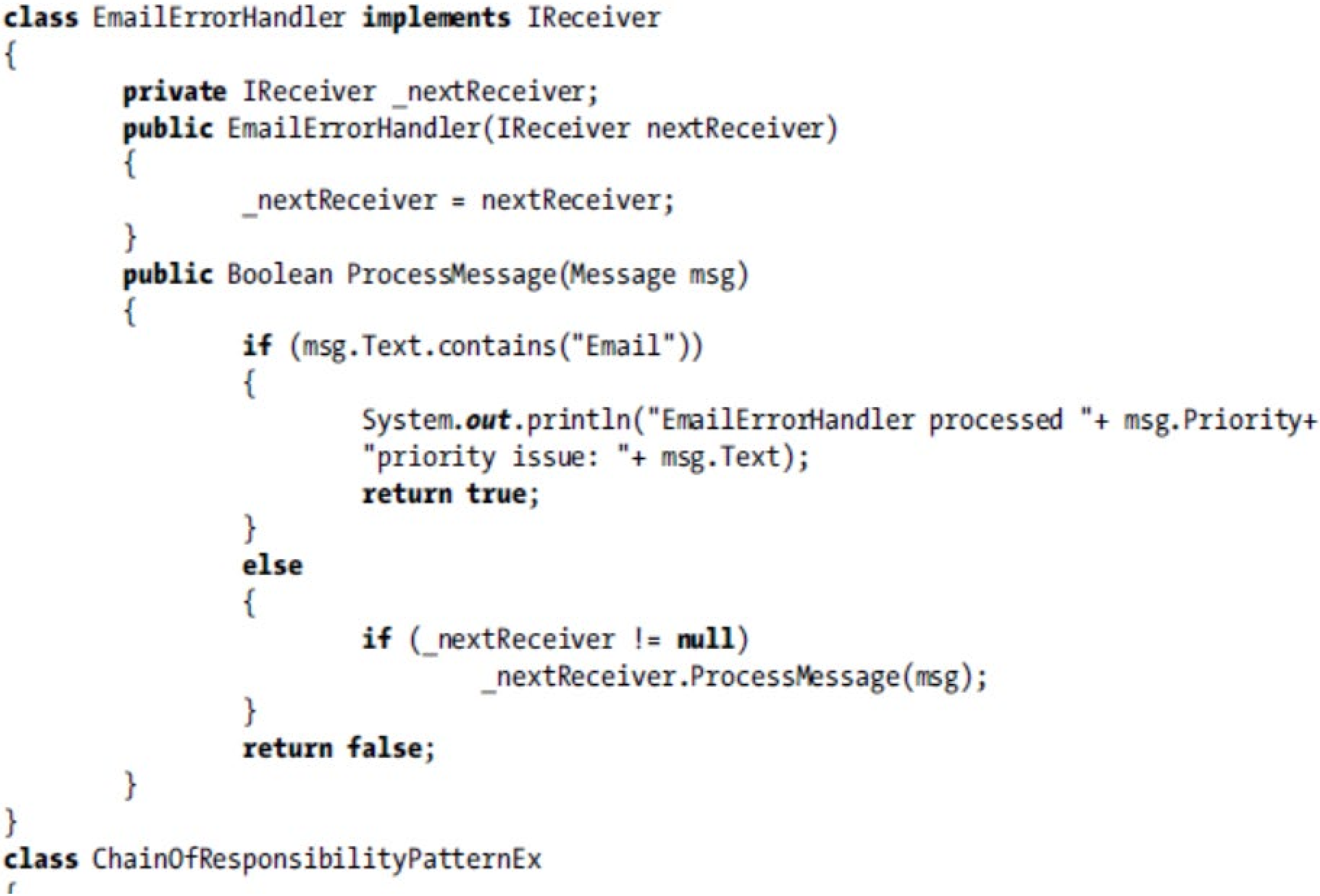
4. The **Client** may compose chains just once or compose them dynamically, depending on the application's logic. Note that a request can be sent to any handler in the chain—it doesn't have to be the first one.

Chain of Responsibility

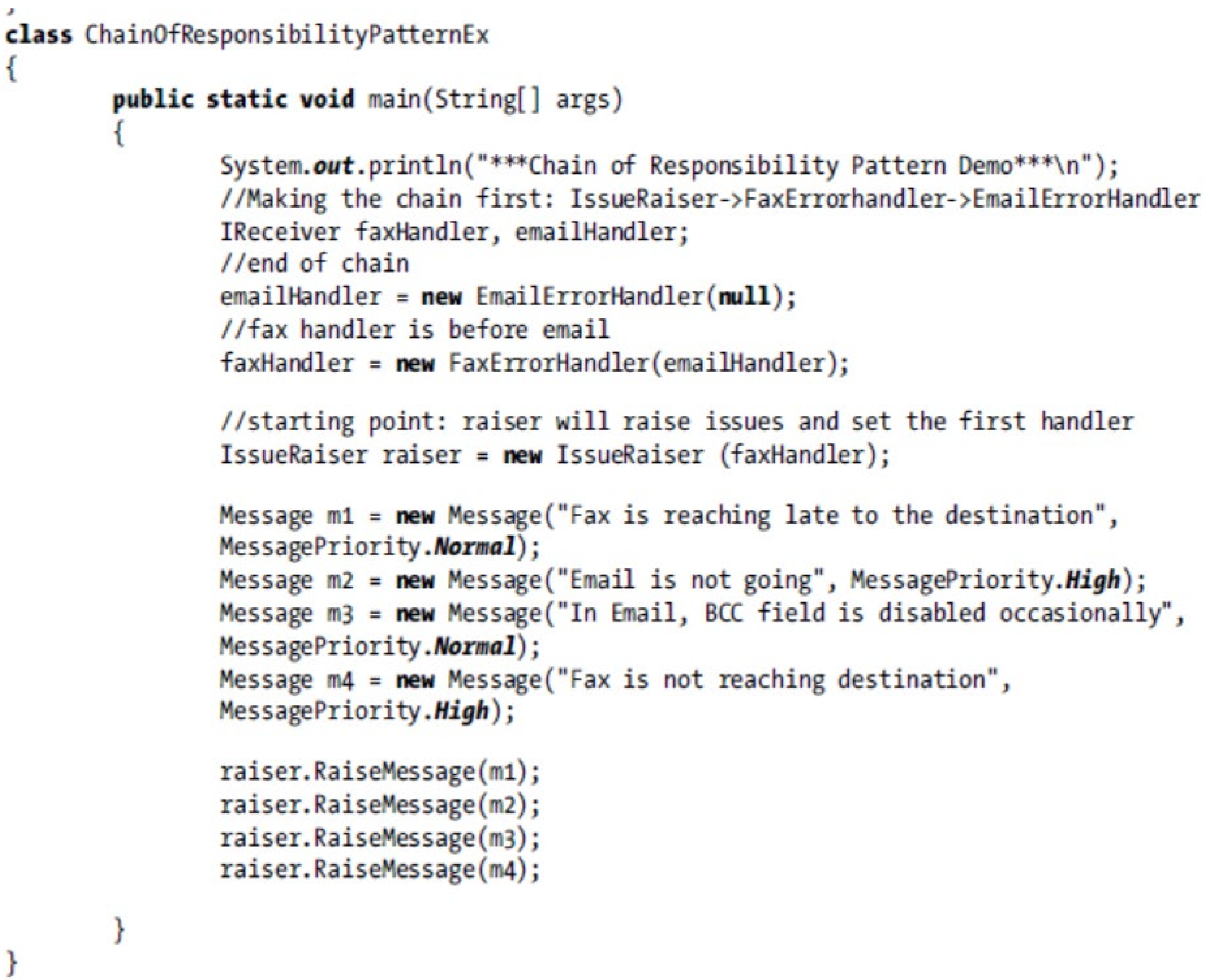

<<Java Class>>
ChainOfResponsibilityPatternEx
chainofresponsibility.pattern.demo
ChainOfResponsibilityPatternEx()
main(String[]):void
<<Java Class>>
FaxErrorHandler
chainofresponsibility.pattern.demo
FaxErrorHandler(IReceiver)
ProcessMessage(Message):Boolean
<<Java Class>>
EmailErrorHandler
chainofresponsibility.pattern.demo
EmailErrorHandler(IReceiver)
ProcessMessage(Message):Boolean
<<Java Enumeration>>
MessagePriority
chainofresponsibility.pattern.demo
Normal: MessagePriority
High: MessagePriority
MessagePriority()
-_nextReceiver
0..1
-_nextReceiver
0..1
<<Java Interface>>
IReceiver
chainofresponsibility.pattern.demo
ProcessMessage(Message):Boolean
+Priority
0..1
+setFirstReceiver
0..1
<<Java Class>>
Message
chainofresponsibility.pattern.demo
Text: String
Message(String,MessagePriority)
<<Java Class>>
IssueRaiser
chainofresponsibility.pattern.demo
IssueRaiser(IReceiver)
RaiseMessage(Message):void



```
interface IReceiver
{
        Boolean ProcessMessage(Message msg);
}
Class IssueRaiser
{
        public IReceiver setFirstReceiver;
        public IssueRaiser(IReceiver firstReceiver)
        {
                this.setFirstReceiver = firstReceiver;
        }
        public void RaiseMessage(Message msg)
        {
                if (setFirstReceiver != null)
                        setFirstReceiver.ProcessMessage(msg);
        }
}
class FaxErrorHandler implements IReceiver
{
        private IReceiver _nextReceiver;
        public FaxErrorHandler(IReceiver nextReceiver)
        {
                _nextReceiver = nextReceiver;
        }
        public Boolean ProcessMessage(Message msg)
        {
                if (msg.Text.contains("Fax"))
                {
                        System.out.println("FaxErrorHandler processed "+  msg.Priority+
                        "priority issue: "+ msg.Text);
                        return true;
                }
                else
                {
                        if (_nextReceiver != null)
                                _nextReceiver.ProcessMessage(msg);
                }
                return false;
        }
}
```

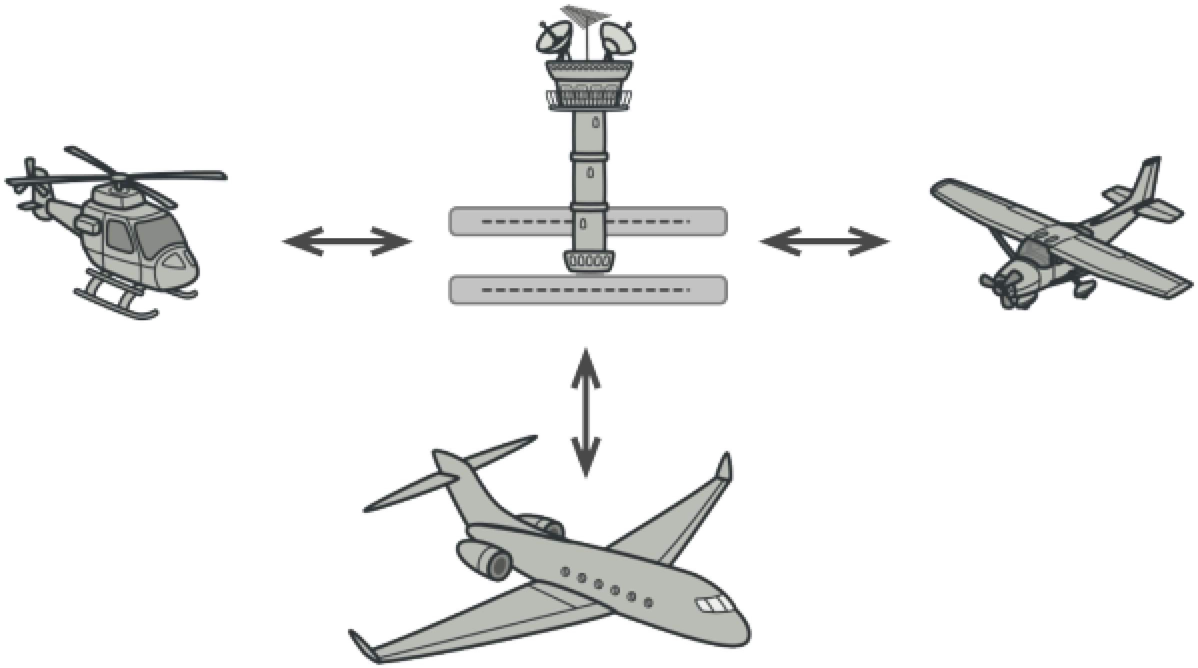


```
class EmailErrorHandler implements IReceiver
{
        private IReceiver _nextReceiver;
        public EmailErrorHandler(IReceiver nextReceiver)
        {
                _nextReceiver = nextReceiver;
        }
        public Boolean ProcessMessage(Message msg)
        {
                if (msg.Text.contains("Email"))
                {
                        System.out.println("EmailErrorHandler processed "+ msg.Priority+
                        "priority issue: "+ msg.Text);
                        return true;
                }
                else
                {
                        if (_nextReceiver != null)
                                _nextReceiver.ProcessMessage(msg);
                }
                return false;
        }
}
class ChainOfResponsibilityPatternEx
{
```



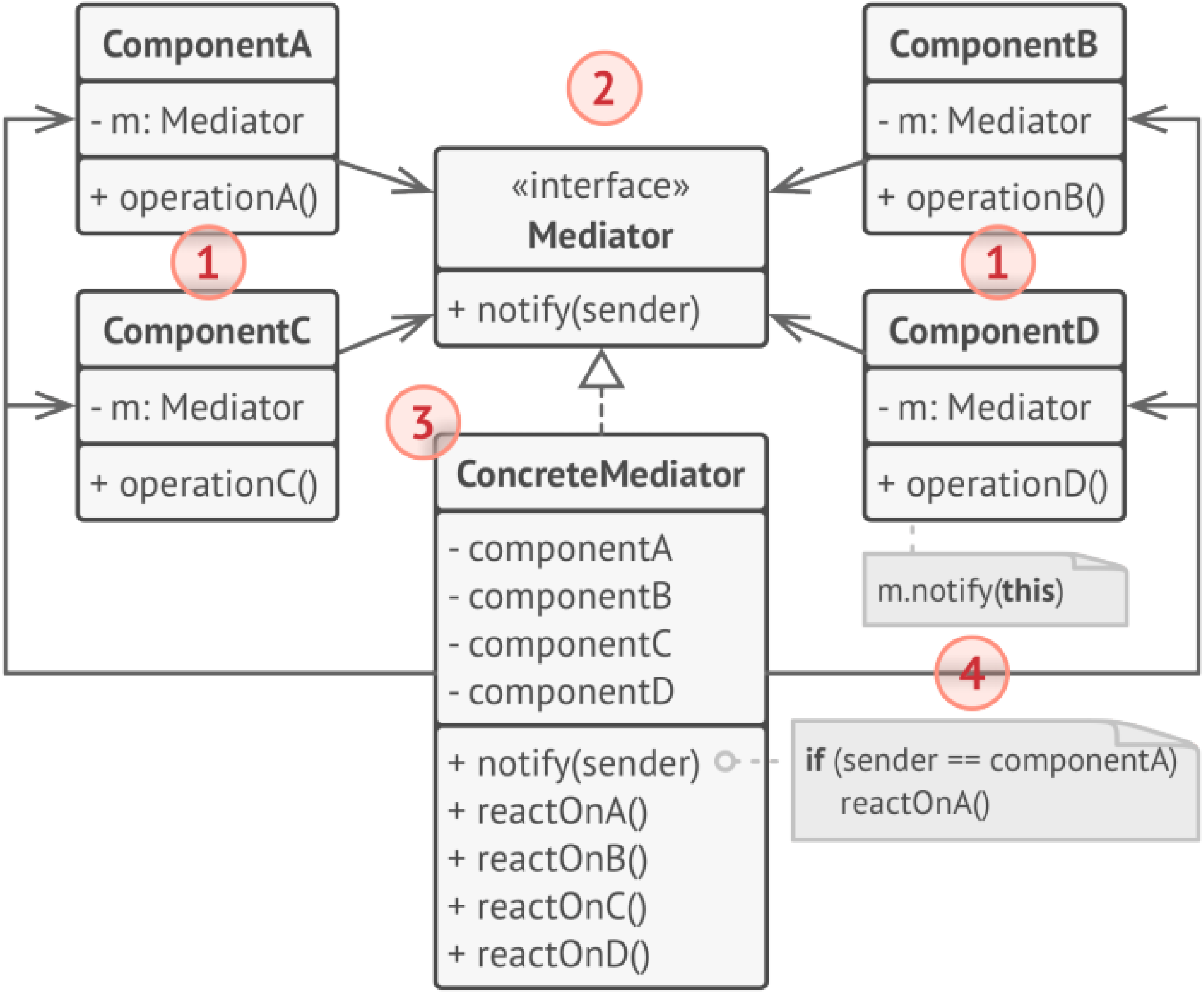
```
,
class ChainOfResponsibilityPatternEx
{
        public static void main(String[] args)
        {
                System.out.println("***Chain of Responsibility Pattern Demo***\n");
                //Making the chain first: IssueRaiser->FaxErrorhandler->EmailErrorHandler
                IReceiver faxHandler, emailHandler;
                //end of chain
                emailHandler = new EmailErrorHandler(null);
                //fax handler is before email
                faxHandler = new FaxErrorHandler(emailHandler);

                //starting point: raiser will raise issues and set the first handler
                IssueRaiser raiser = new IssueRaiser (faxHandler);

                Message m1 = new Message("Fax is reaching late to the destination",
                MessagePriority.Normal);
                Message m2 = new Message("Email is not going", MessagePriority.High);
                Message m3 = new Message("In Email, BCC field is disabled occasionally",
                MessagePriority.Normal);
                Message m4 = new Message("Fax is not reaching destination",
                MessagePriority.High);

                raiser.RaiseMessage(m1);
                raiser.RaiseMessage(m2);
                raiser.RaiseMessage(m3);
                raiser.RaiseMessage(m4);

        }
}
```

Mediator

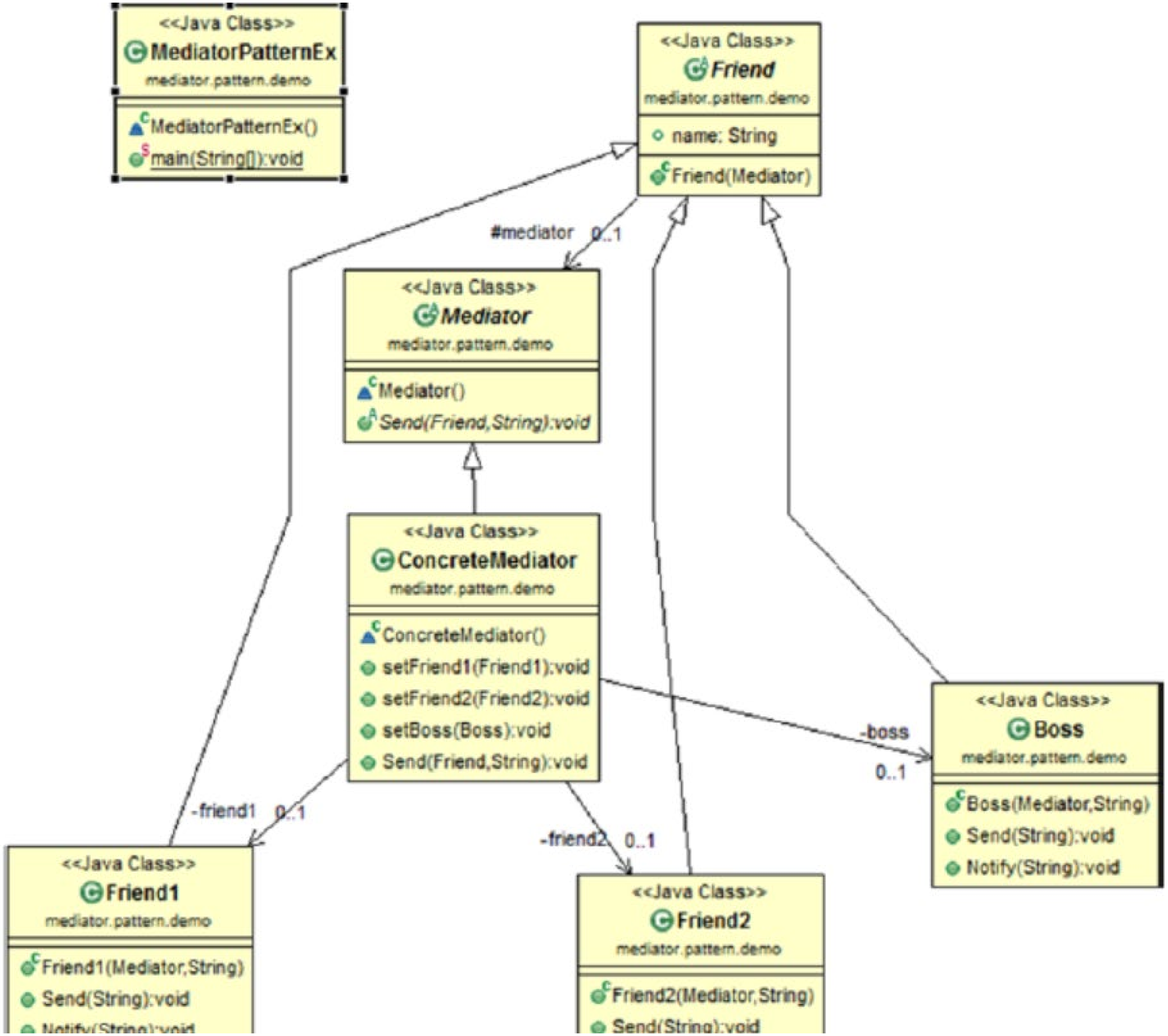


Aircraft pilots don’t talk to each other directly when deciding who gets to land their plane next. All communication goes through the control tower.

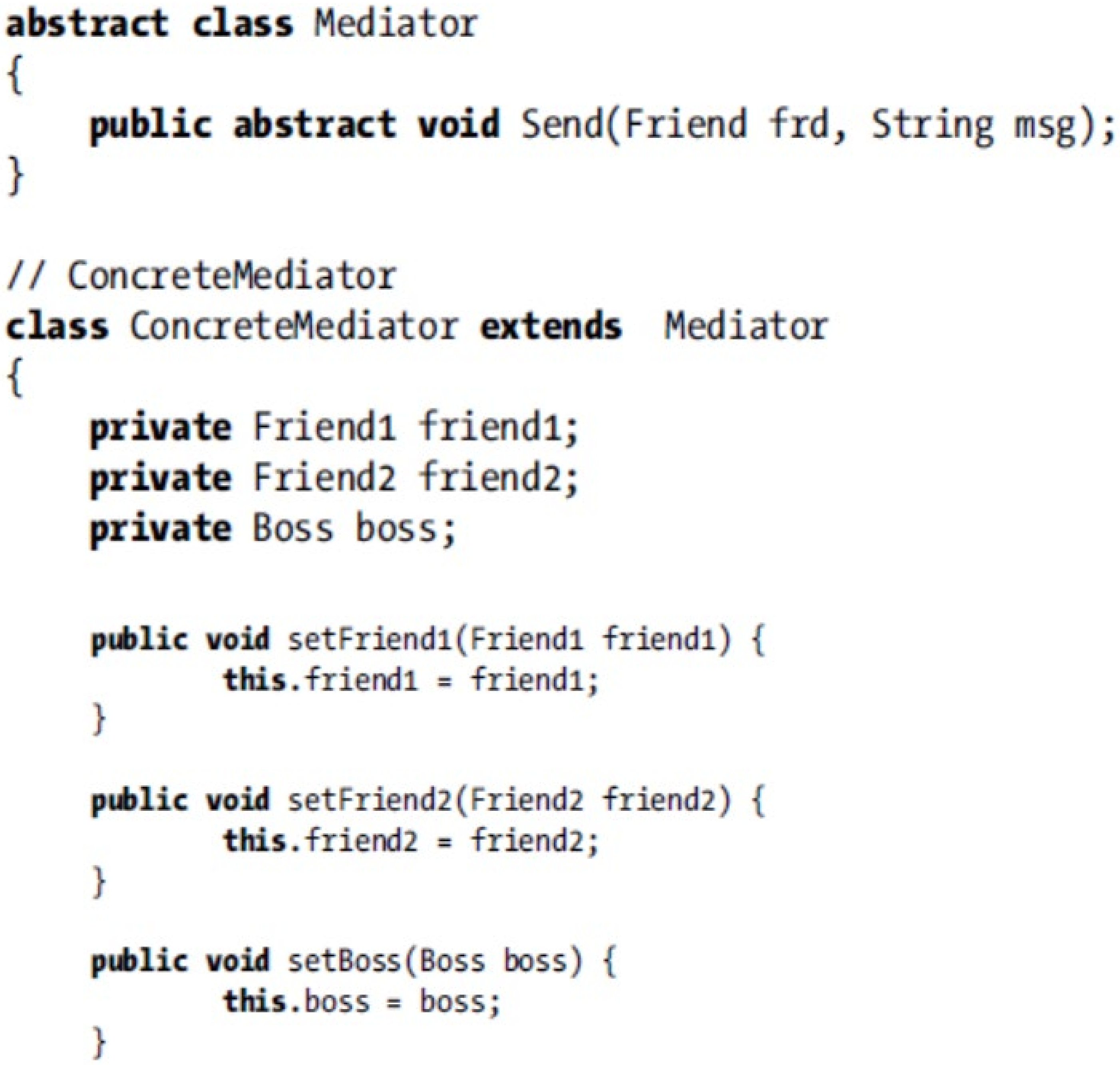
Solution


ComponentA
- m: Mediator
+ operationA()
ComponentB
- m: Mediator
+ operationB()
2
«interface»
Mediator
+ notify(sender)
1
1
ComponentC
- m: Mediator
+ operationC()
ComponentD
- m: Mediator
+ operationD()
3
ConcreteMediator
- componentA
- componentB
- componentC
- componentD
+ notify(sender)
+ reactOnA()
+ reactOnB()
+ reactOnC()
+ reactOnD()
m.notify(this)
4
if (sender == componentA)
reactOnA()

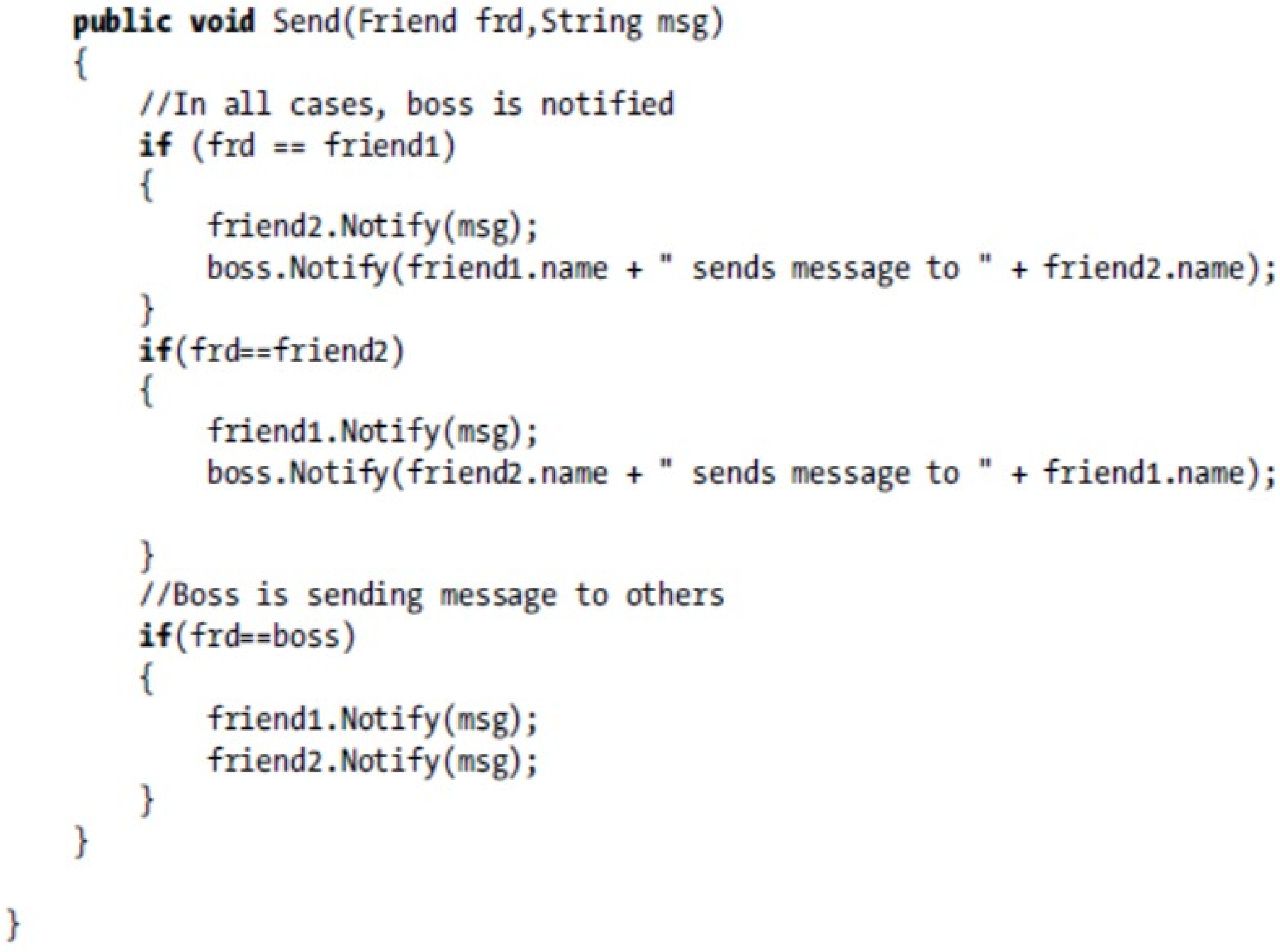
Mediator



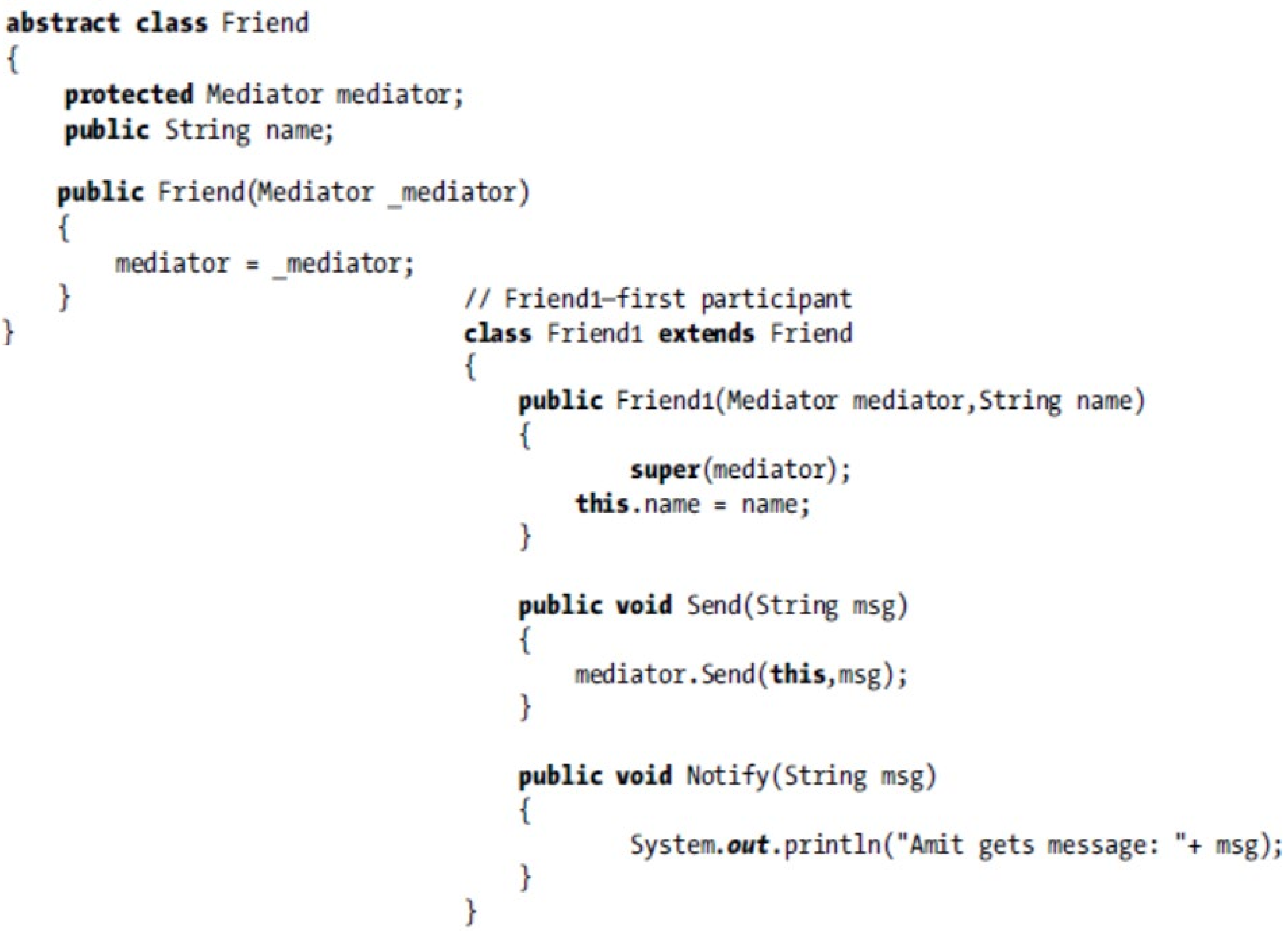
<<Java Class>>
MediatorPatternEx
mediator.pattern.demo
MediatorPatternEx()
main(String[]):void
<<Java Class>>
Friend
mediator.pattern.demo
name: String
Friend(Mediator)
#mediator 0..1
<<Java Class>>
Mediator
mediator.pattern.demo
Mediator()
Send(Friend,String):void
<<Java Class>>
ConcreteMediator
mediator.pattern.demo
ConcreteMediator()
setFriend1(Friend1):void
setFriend2(Friend2):void
setBoss(Boss):void
Send(Friend,String):void
-boss
0..1
<<Java Class>>
Boss
mediator.pattern.demo
Boss(Mediator,String)
Send(String):void
Notify(String):void
-friend1 0..1
-friend2 0..1
<<Java Class>>
Friend1
mediator.pattern.demo
Friend1(Mediator,String)
Send(String):void
Notify(String):void
<<Java Class>>
Friend2
mediator.pattern.demo
Friend2(Mediator,String)
Send(String):void



```
abstract class Mediator
{
    public abstract void Send(Friend frd, String msg);
}

// ConcreteMediator
class ConcreteMediator extends  Mediator
{
    private Friend1 friend1;
    private Friend2 friend2;
    private Boss boss;


    public void setFriend1(Friend1 friend1) {
            this.friend1 = friend1;
    }

    public void setFriend2(Friend2 friend2) {
            this.friend2 = friend2;
    }

    public void setBoss(Boss boss) {
            this.boss = boss;
    }
```

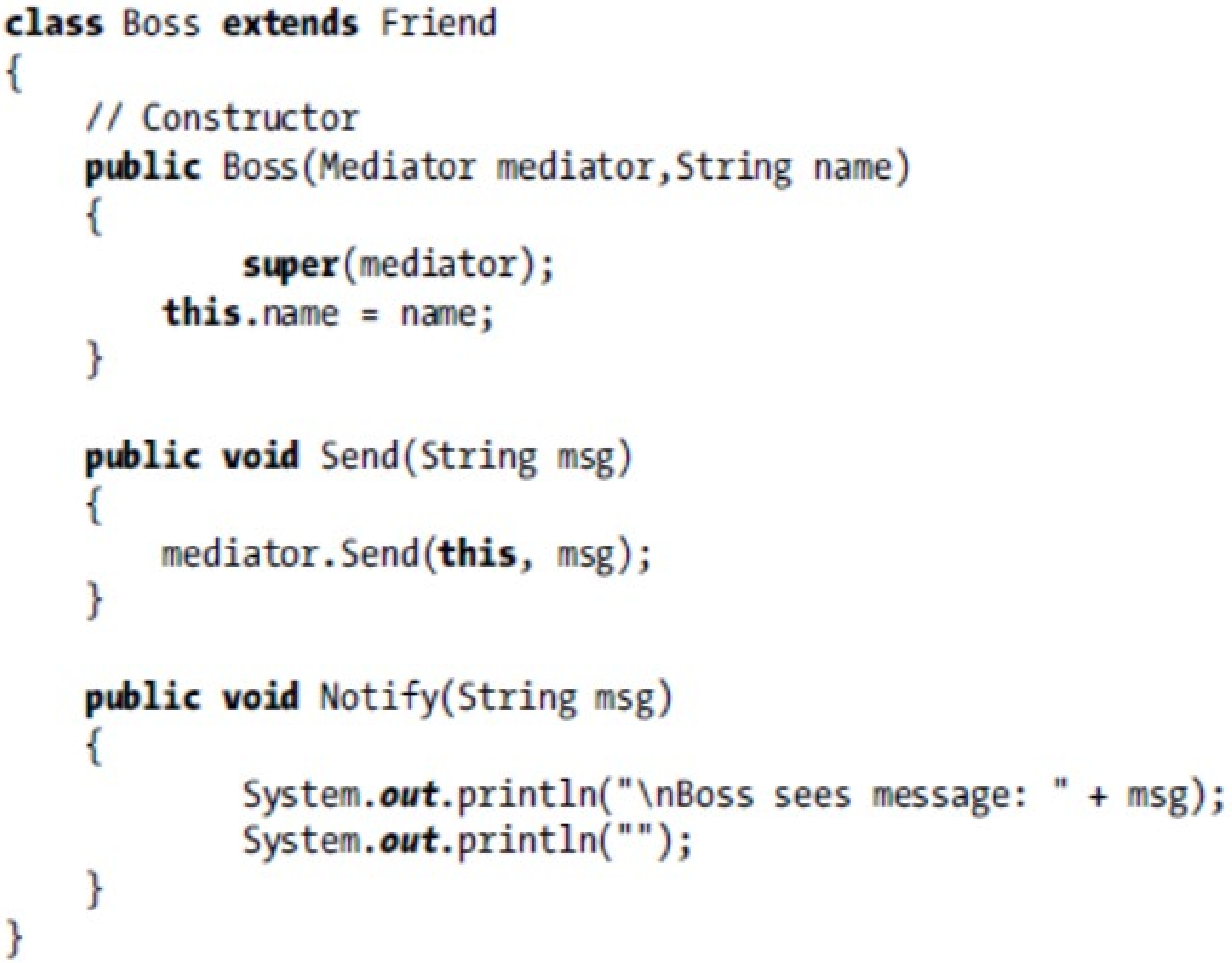


```
    public void Send(Friend frd,String msg)
    {
        //In all cases, boss is notified
        if (frd == friend1)
        {
            friend2.Notify(msg);
            boss.Notify(friend1.name + " sends message to " + friend2.name);
        }
        if(frd==friend2)
        {
            friend1.Notify(msg);
            boss.Notify(friend2.name + " sends message to " + friend1.name);

        }
        //Boss is sending message to others
        if(frd==boss)
        {
            friend1.Notify(msg);
            friend2.Notify(msg);
        }
    }

}
```

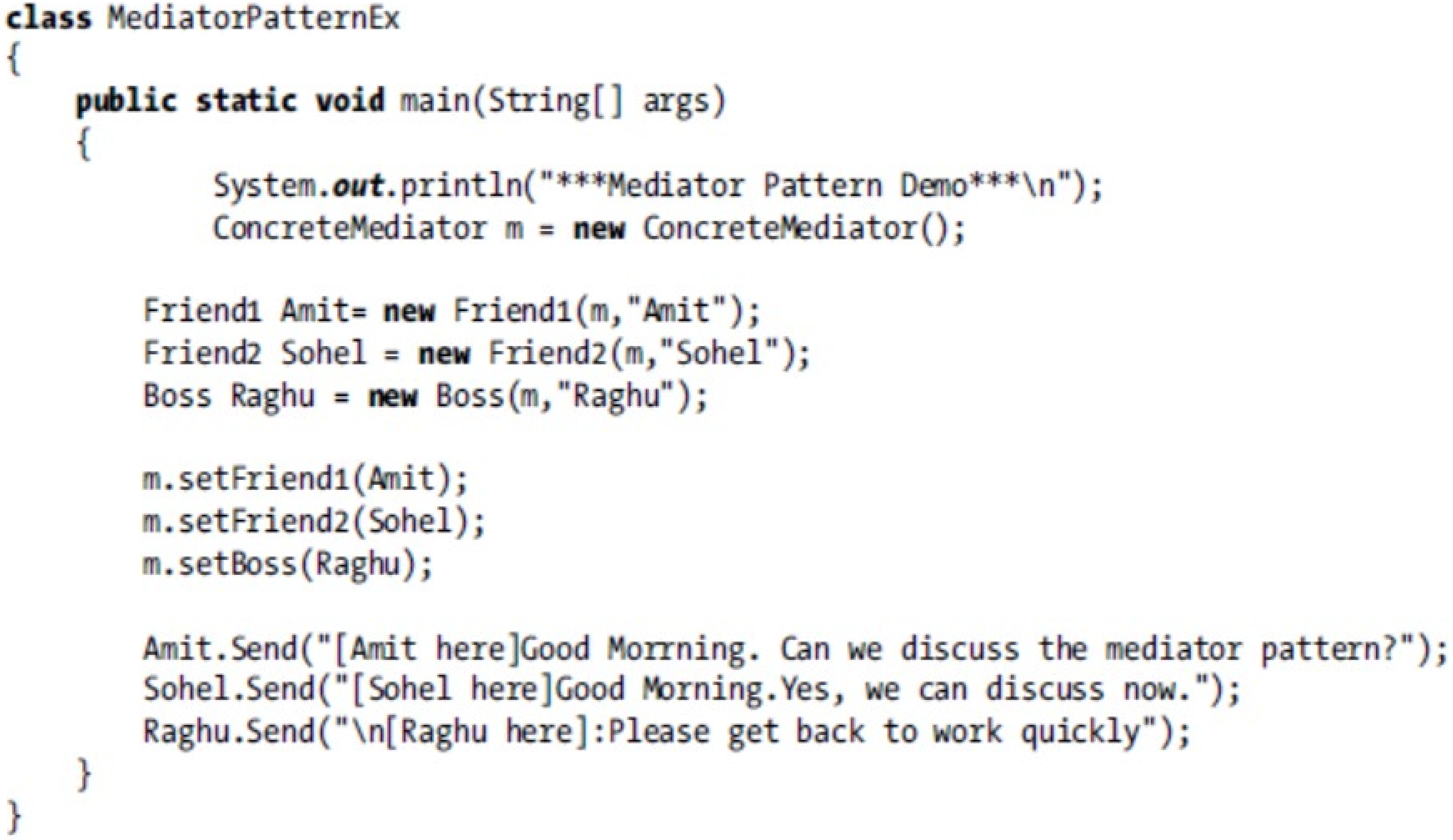


```
abstract class Friend
{
     protected Mediator mediator;
     public String name;

    public Friend(Mediator _mediator)
    {
        mediator = _mediator;
    }
}
```

```
// Friend1-first participant
class Friend1 extends Friend
{
    public Friend1(Mediator mediator,String name)
    {
          super(mediator);
        this.name = name;
    }

    public void Send(String msg)
    {
        mediator.Send(this,msg);
    }

    public void Notify(String msg)
    {
          System.out.println("Amit gets message: "+ msg);
    }
}
```



```
class Boss extends Friend
{
    // Constructor
    public Boss(Mediator mediator,String name)
    {
            super(mediator);
        this.name = name;
    }

    public void Send(String msg)
    {
        mediator.Send(this, msg);
    }

    public void Notify(String msg)
    {
            System.out.println("\nBoss sees message: " + msg);
            System.out.println("");
    }
}
```



```
class MediatorPatternEx
{
    public static void main(String[] args)
    {
            System.out.println("***Mediator Pattern Demo***\n");
            ConcreteMediator m = new ConcreteMediator();

        Friend1 Amit= new Friend1(m,"Amit");
        Friend2 Sohel = new Friend2(m,"Sohel");
        Boss Raghu = new Boss(m,"Raghu");

        m.setFriend1(Amit);
        m.setFriend2(Sohel);
        m.setBoss(Raghu);

        Amit.Send("[Amit here]Good Morrning. Can we discuss the mediator pattern?");
        Sohel.Send("[Sohel here]Good Morning.Yes, we can discuss now.");
        Raghu.Send("\n[Raghu here]:Please get back to work quickly");
    }
}
```