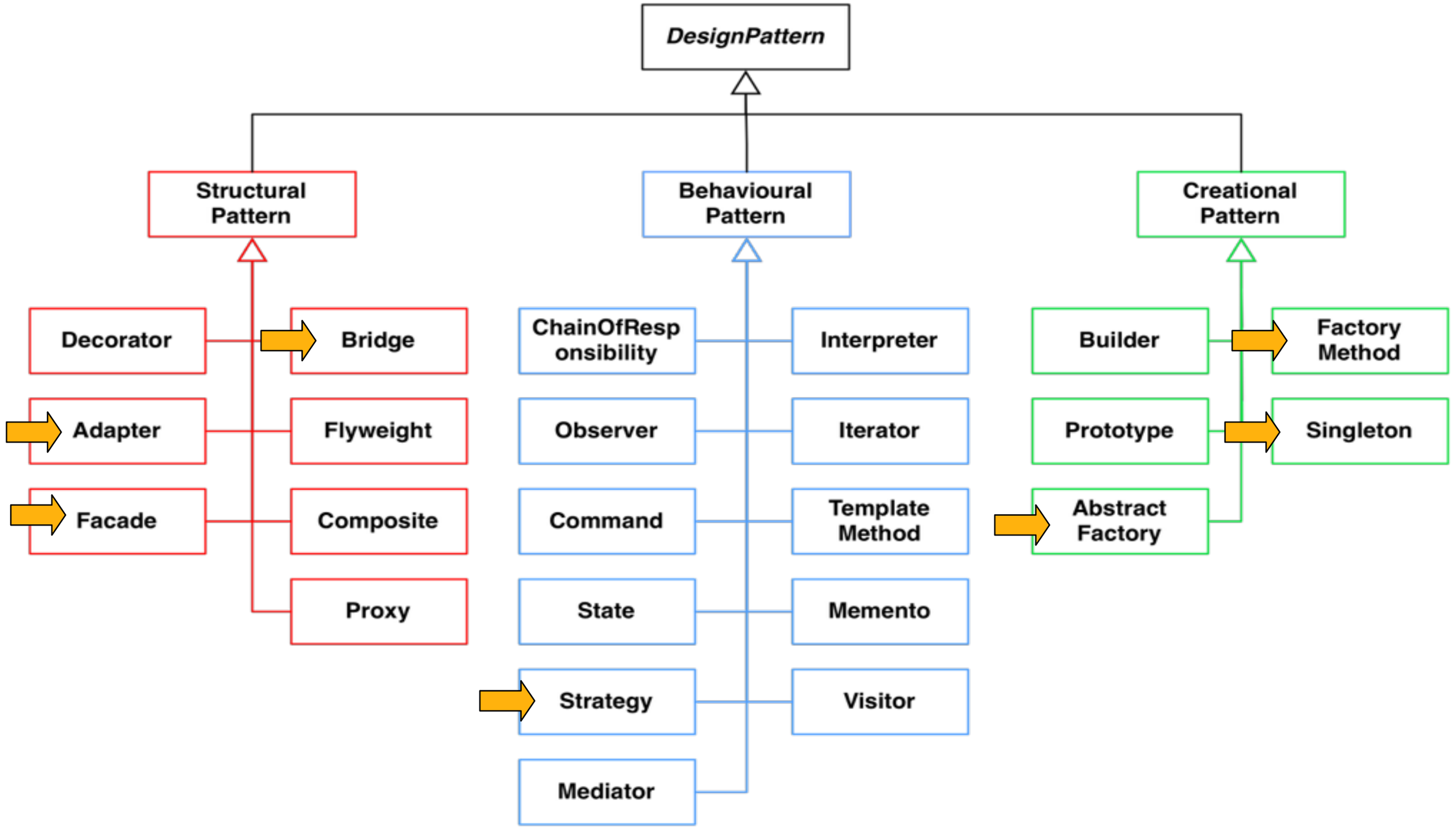
COIS3040 Lecture 6

So far ..


DesignPattern
Structural Pattern
Behavioural Pattern
Creational Pattern
Decorator
Bridge
Adapter
Flyweight
Facade
Composite
Proxy
ChainOfResponsibility
Interpreter
Observer
Iterator
Command
Template Method
State
Memento
Strategy
Visitor
Mediator
Builder
Factory Method
Prototype
Singleton
Abstract Factory

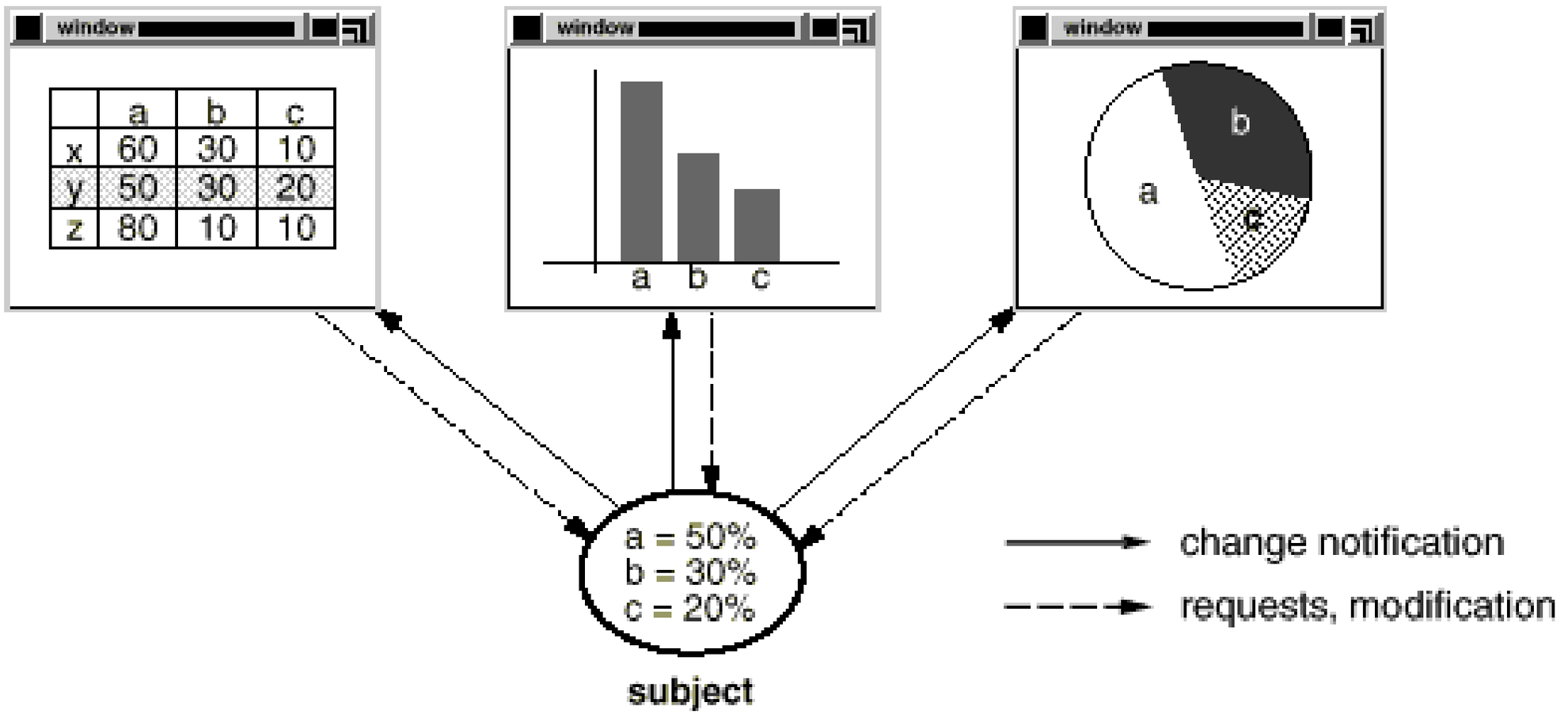
Adapter VS Bridge

- "Adapter makes things work after they're designed; Bridge makes them work before they are. [GoF, p219]”
- Adapter pattern is useful when you have existing code, be it third party, or in-house, but out of your control, or otherwise not changeable to quite meet the interface you need it to.
- The Bridge pattern is something you implement up front - if you know you have two orthogonal hierarchies, it provides a way to decouple the interface and the implementation in such a way that you don't get an insane number of classes.

Observer

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

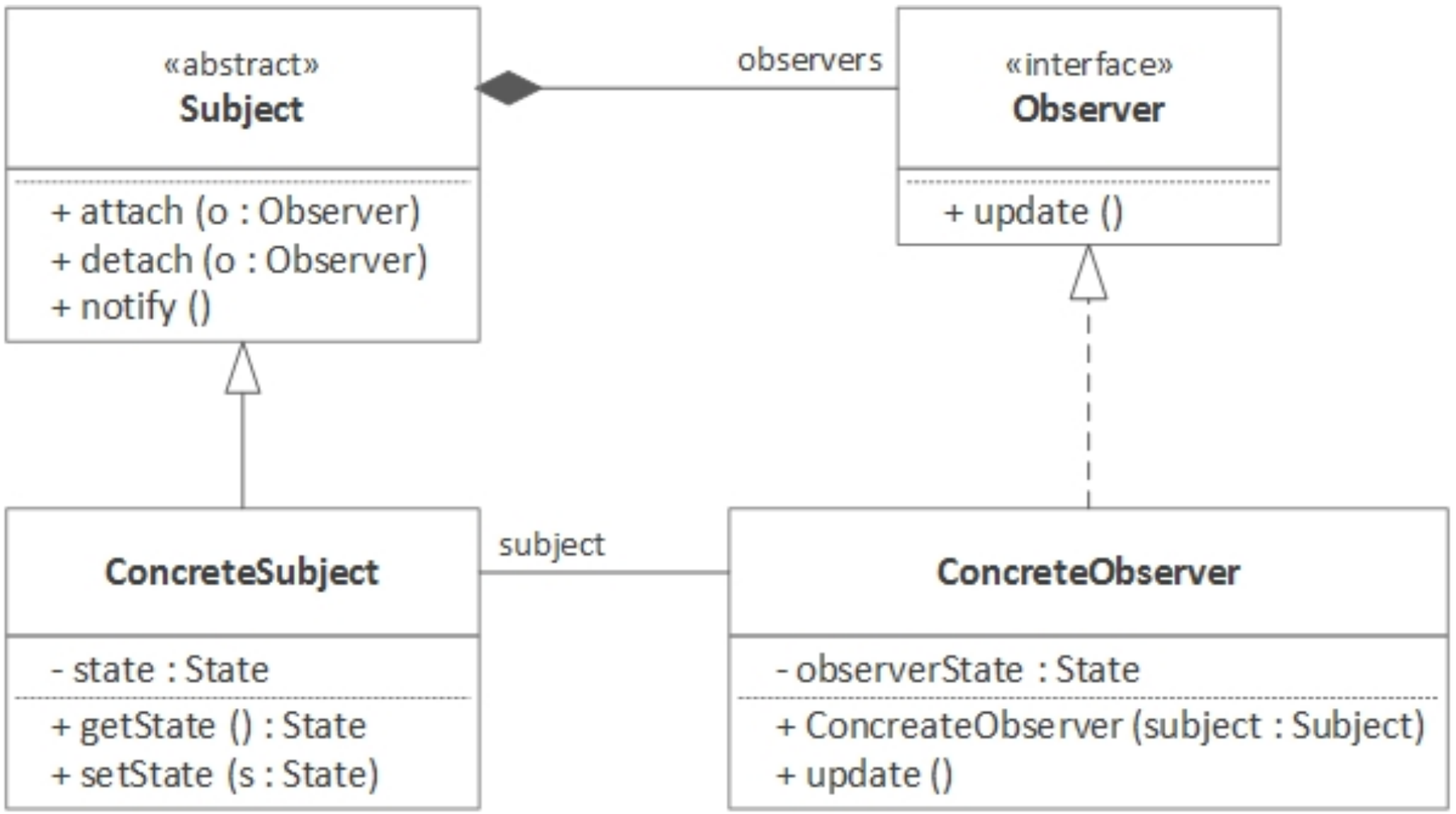
Observer


window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
window
a b c
window
b
a
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modification

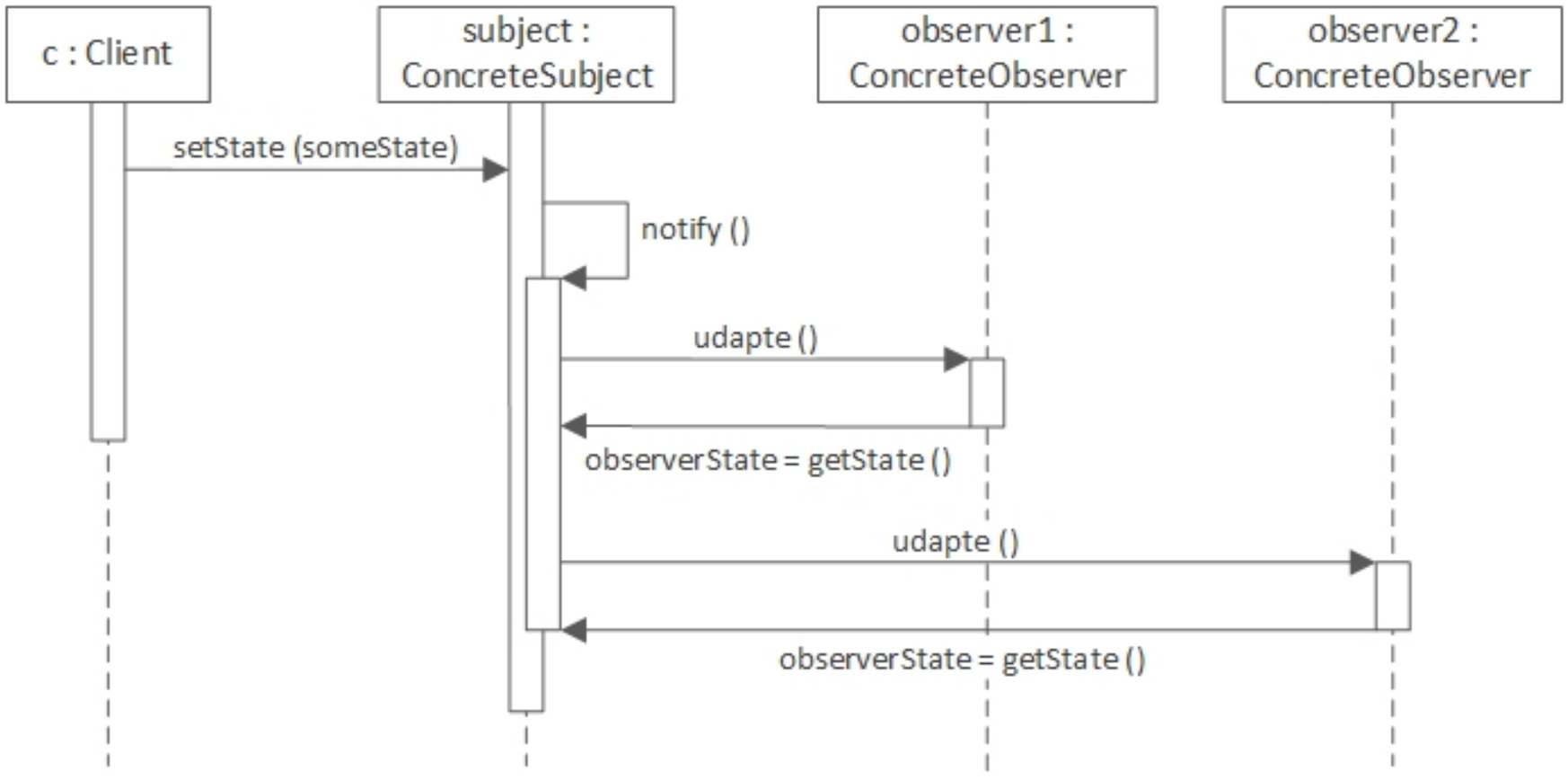
Observer

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are.

Observer


«abstract»
Subject
+ attach (o : Observer)
+ detach (o : Observer)
+ notify ()
observers
«interface»
Observer
+ update ()
ConcreteSubject
- state : State
+ getState () : State
+ setState (s : State)
subject
ConcreteObserver
- observerState : State
+ ConcreateObserver (subject : Subject)
+ update ()

Observer


c : Client
subject : ConcreteSubject
observer1 : ConcreteObserver
observer2 : ConcreteObserver
setState (someState)
notify ()
udapte ()
observerState = getState ()
udapte ()
observerState = getState ()

Observer

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

class HexObserver extends Observer {
  public HexObserver( Subject s ) {
    subj = s;
    subj.attach( this );
  }
  public void update() {
  System.out.print( " " +
Integer.toHexString( subj.getState() ) );
  }
}
```

Observer

```
class BinObserver extends Observer {
  public BinObserver( Subject s ) {
    subj = s;
    subj.attach( this ); } // Observers register
themselves
    public void update() {
    System.out.print( " " +
Integer.toBinaryString( subj.getState() ) );
  }
}
```

Observer

```
class Subject {
  private Observer[] observers = new Observer[9];
  private int totalObs = 0;
  private int state;
  public void attach( Observer o ) {
    observers[totalObs++] = o;
  }

  public int getState() {
    return state;
  }

  public void setState( int in ) {
    state = in;
    notify();
  }

  private void notify() {
    for (int i=0; i < totalObs; i++) {
      observers[i].update();
    }
  }
}
```

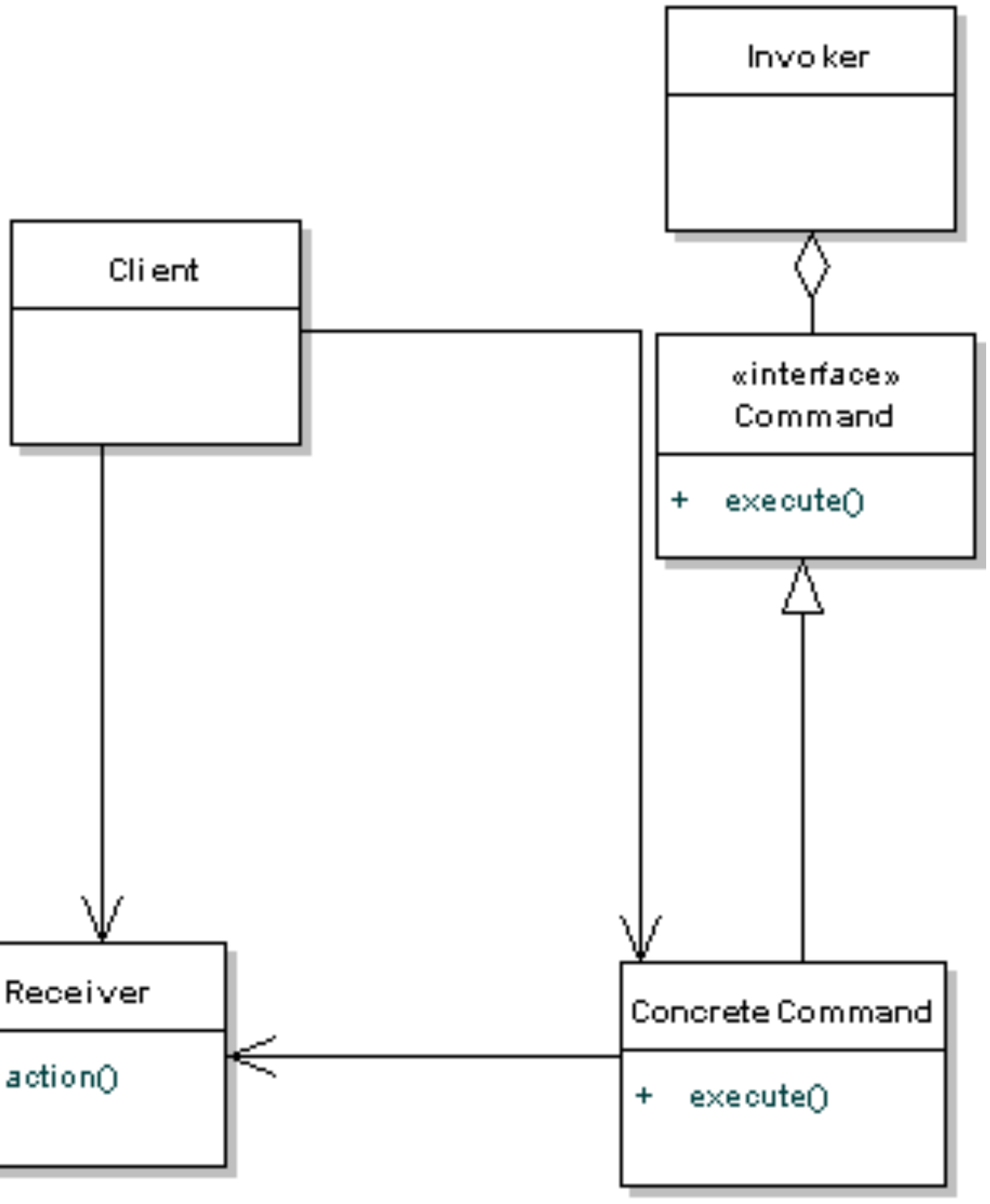
Observer

```
public class ObserverDemo {
  public static void main( String[] args ) {
    Subject sub = new Subject();
    // Client configures the number and type of Observers
    new HexObserver( sub );
    new BinObserver( sub );
    Scanner scan = new Scanner();
    while (true) {
      System.out.print( "\nEnter a number: " );
      sub.setState( scan.nextInt() );
    }
  }
}
```

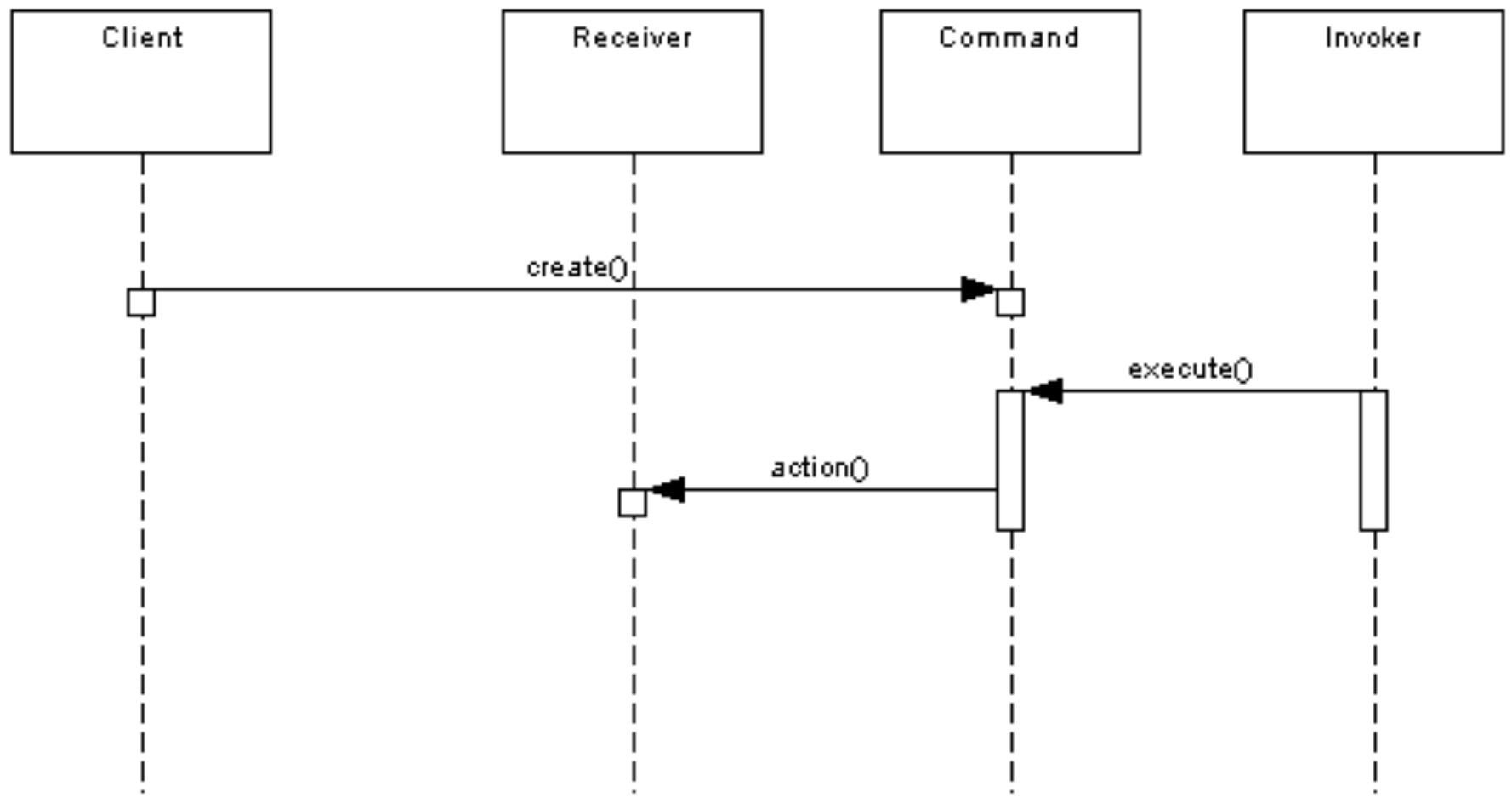
Command

- Command is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.
- Support undo with Unexecute operation. Executed commands are stored in a history list.

Command


Invoker
Client
«interface»
Command
+ execute()
Receiver
+ action()
Concrete Command
+ execute()

Command



Command

- **Command:**
  - Declares an interface for executing an operation.
- **ConcreteCommand:**
  - Defines a binding between a Receiver object and an action.
  - Implements Execute by invoking the corresponding operation(s) on Receiver.
- **Client**
  - Creates a ConcreteCommand object and sets its receiver.
- **Invoker**
  - Asks the command to carry out the request.
- **Receiver**
  - Knows how to perform the operation associated with the request.

Command

```
public interface Command{
  public void execute();
}
public class LightOnCommand implements Command{
  Light light;
  public LightOnCommand(Light light){
    this.light = light;
  }
  public void execute(){
    light.switchOn();
  }
}
```

Command

```
//Receiver
public class Light{
  private boolean on;
  public void switchOn(){
    on = true;
  }
  public void switchOff(){
    on = false;
  }
}
```

Command

```
//Invoker
public class RemoteControl{
  private Command command;
  public void setCommand(Command command){
    this.command = command;
  }
  public void pressButton(){
    command.execute();
  }
}
```

Command

```
//Client
public class Client{
  public static void main(String[] args)    {
    RemoteControl control = new RemoteControl();
    Light light = new Light();
    Command lightsOn = new LightsOnCommand(light);
   //switch on
    control.setCommand(lightsOn);
    control.pressButton();
  }
}
```

Mediator

- Helpful in an enterprise application where multiple objects are interacting with each other.
- If the objects interact with each other directly, the system components are tightly-coupled with each other that makes maintainability cost higher and not flexible to extend easily.
- Mediator pattern focuses on provide a mediator between objects for communication and help in implementing lose-coupling between objects.

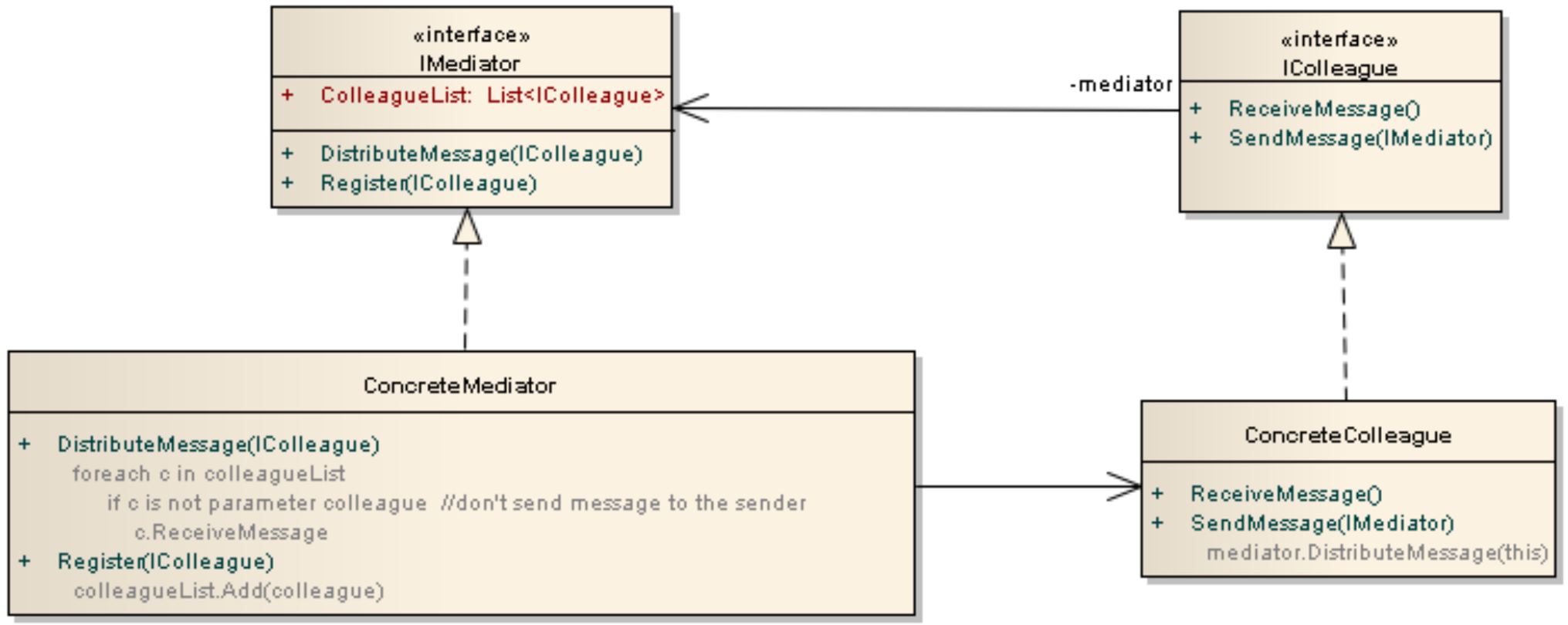
Mediator

- Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights. Mediator works as a router between objects and it can have it’s own logic to provide way of communication.

Mediator

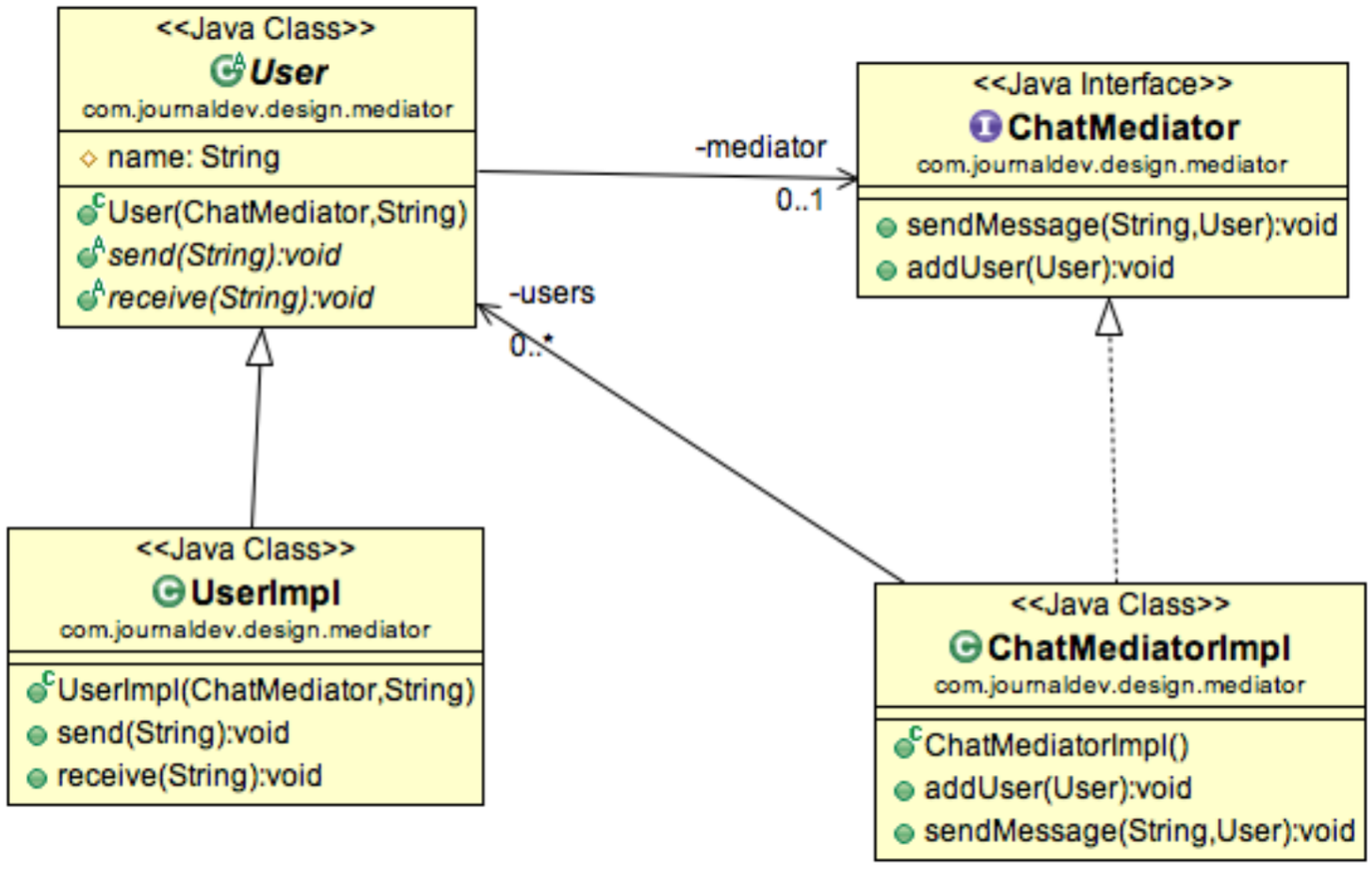
- The system objects that communicate each other are called Colleagues. Usually we have an interface or abstract class that provides the contract for communication and then we have concrete implementation of mediators.

Mediator


«interface»
IMediator
+ ColleagueList: List<IColleague>
+ DistributeMessage(IColleague)
+ Register(IColleague)
-mediator
«interface»
IColleague
+ ReceiveMessage()
+ SendMessage(IMediator)
ConcreteMediator
+ DistributeMessage(IColleague)
foreach c in colleagueList
if c is not parameter colleague //don't send message to the sender
c.ReceiveMessage
+ Register(IColleague)
colleagueList.Add(colleague)
ConcreteColleague
+ ReceiveMessage()
+ SendMessage(IMediator)
mediator.DistributeMessage(this)

Mediator

- For our example, we will try to implement a chat application where users can do group chat. Every user will be identified by it’s name and they can send and receive messages. The message sent by any user should be received by all the other users in the group.



Mediator

```
public interface ChatMediator {

public void sendMessage(String msg, User user);

        void addUser(User user);
}
```

Mediator

```
public abstract class User {
        protected ChatMediator mediator;
        protected String name;

        public User(ChatMediator med, String name){
                this.mediator=med;
                this.name=name;
        }

        public abstract void send(String msg);

        public abstract void receive(String msg);
}
```

Mediator

```
public class ChatMediatorImpl implements ChatMediator {

        private List<User> users;

        public ChatMediatorImpl(){
                this.users=new ArrayList<>();
        }

        @Override
        public void addUser(User user){
                this.users.add(user);
        }

        @Override
        public void sendMessage(String msg, User user) {
                for(User u : this.users){
                        //message should not be received by the user sending it
                        if(u != user){
                                u.receive(msg);
                        }
                }
        }

}
```

Mediator

```
public class UserImpl extends User {

        public UserImpl(ChatMediator med, String name) {
                super(med, name);
        }

        @Override
        public void send(String msg){
                System.out.println(this.name+": Sending Message="+msg);
                mediator.sendMessage(msg, this);
        }
        @Override
        public void receive(String msg) {
                System.out.println(this.name+": Received Message:"+msg);
        }

}
```

Mediator

```
public class ChatClient {

        public static void main(String[] args) {
                ChatMediator mediator = new ChatMediatorImpl();
                User user1 = new UserImpl(mediator, "Pankaj");
                User user2 = new UserImpl(mediator, "Lisa");
                User user3 = new UserImpl(mediator, "Saurabh");
                User user4 = new UserImpl(mediator, "David");
                mediator.addUser(user1);
                mediator.addUser(user2);
                mediator.addUser(user3);
                mediator.addUser(user4);

                user1.send("Hi All");

        }

}
```