PREGRADO

UNIDAD 2 | SOFTWARE FEATURE LEVEL DESIGN & PATTERNS

GOG BEHAVIORAL DESIGN PATTERNS

SI720 | Diseño y patrones de software

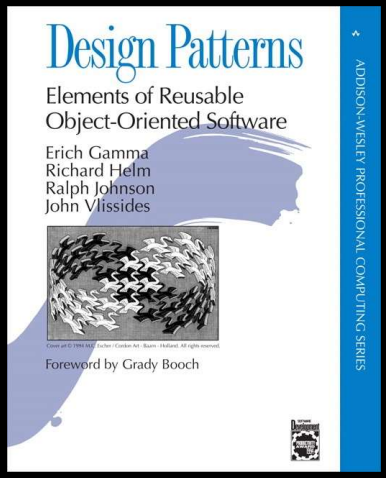
Al finalizar la unidad, el estudiante elabora y comunica artefactos de diseño de software aplicando principios básicos y
patrones de diseño para un dominio y contexto determinados

AGENDA

INTRO
CHAIN OF RESPONSIBILITY
COMMAND
ITERATOR
OBSERVER
STATES
OTHER PATTERNS

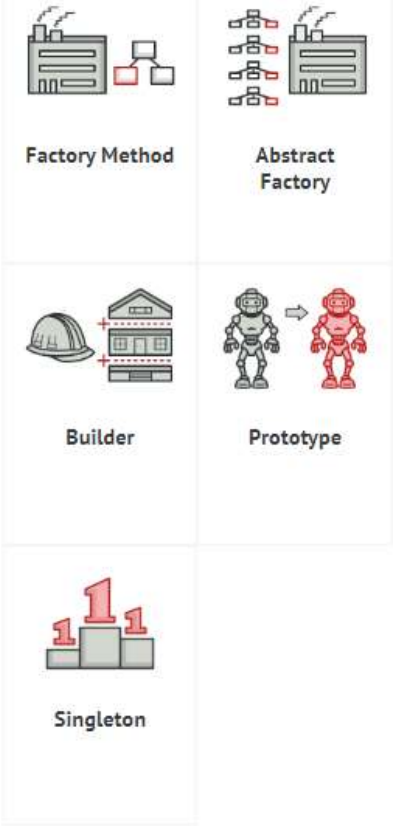
GoF design patterns

Creational	Structural	Behavioral
Builder	Adapter	Chain of responsibility
Factory Method	Bridge	Command
Prototype	Composite	Interpreter
Singleton	Decorator	Iterator
	Façade	Mediator
	Flyweight	Memento
	Proxy	Observer
		States
		Strategy
		Template Method
		Visitor



Creational

Mecanismos de creación de objetos, incrementan flexibilidad y reutilización de código

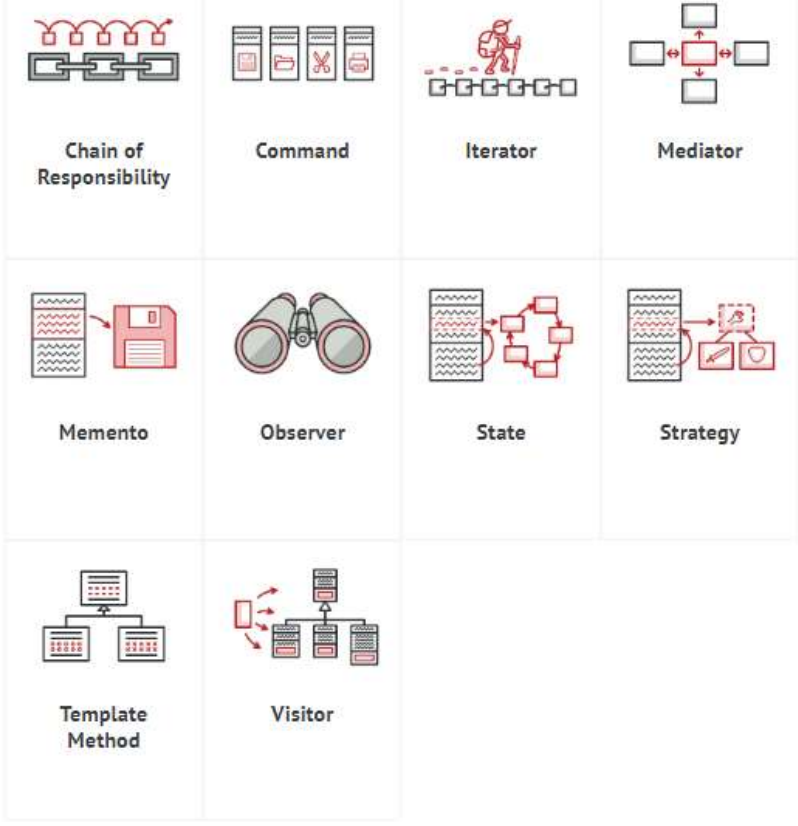


Structural

Facilitan la organización de objetos y clases en estructura mas grandes, manteniendo la estructura flexible y eficiente

Behavioral

Facilitan el manejo de algoritmos y asignación de responsabilidades entre objetos



Behavioral Design Patterns **Chain of Responsibility**

Permite que un conjunto de clases atienda un request sin conocimiento mutuo.

Se pone a todos los receptores en una cadena la cual permite que el request pase de un receptor al siguiente en la caestdena, hasta que uno lo atienda o éste llegue al final de la cadena.

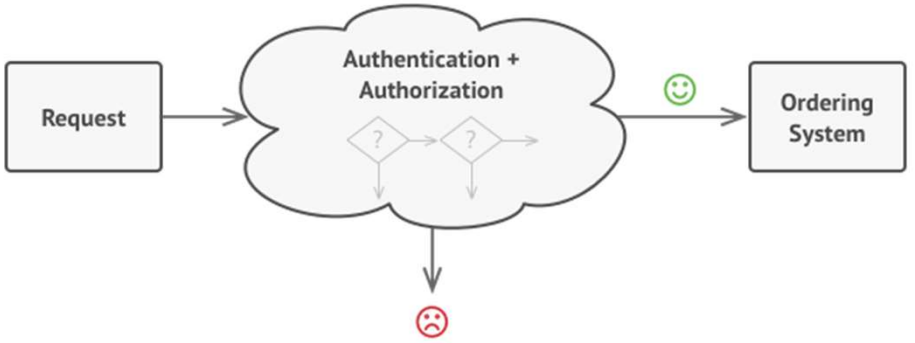
Chain of Responsibility

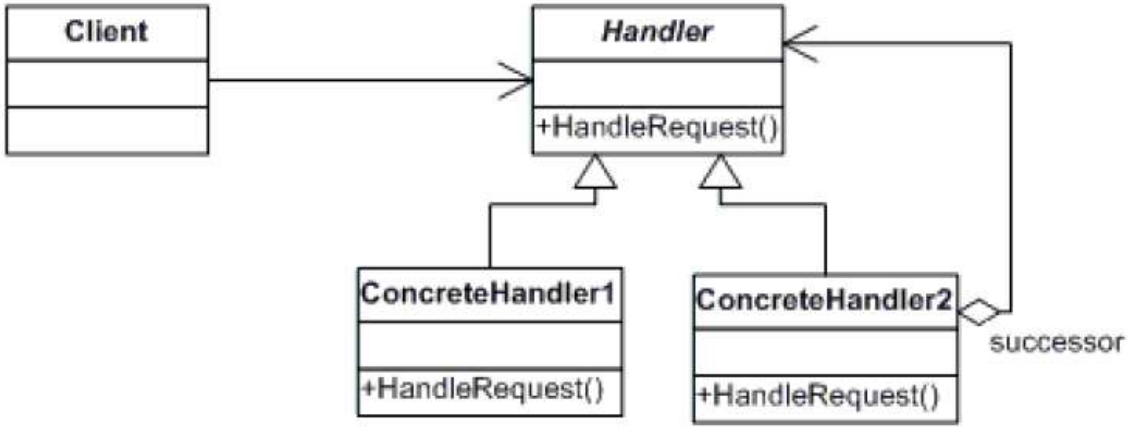
Aplicabilidad

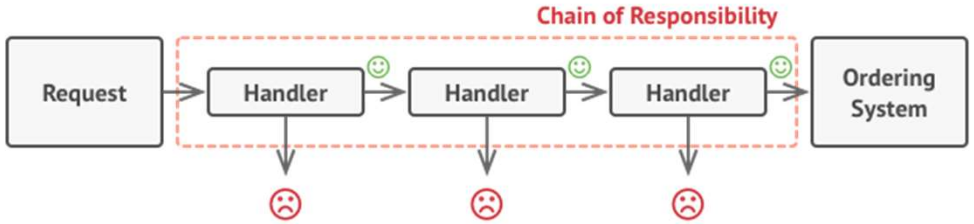
- Cuando más de un objeto podría atender un request, sin conocimiento previo del handler.
- Cuando se requiere enviar un request a uno de varios objetos sin especificar explícitamente al receptor
- Cuando el conjunto de objetos que pueden manejar el request podría estar especificado de forma dinámica.

Chain of Responsibility

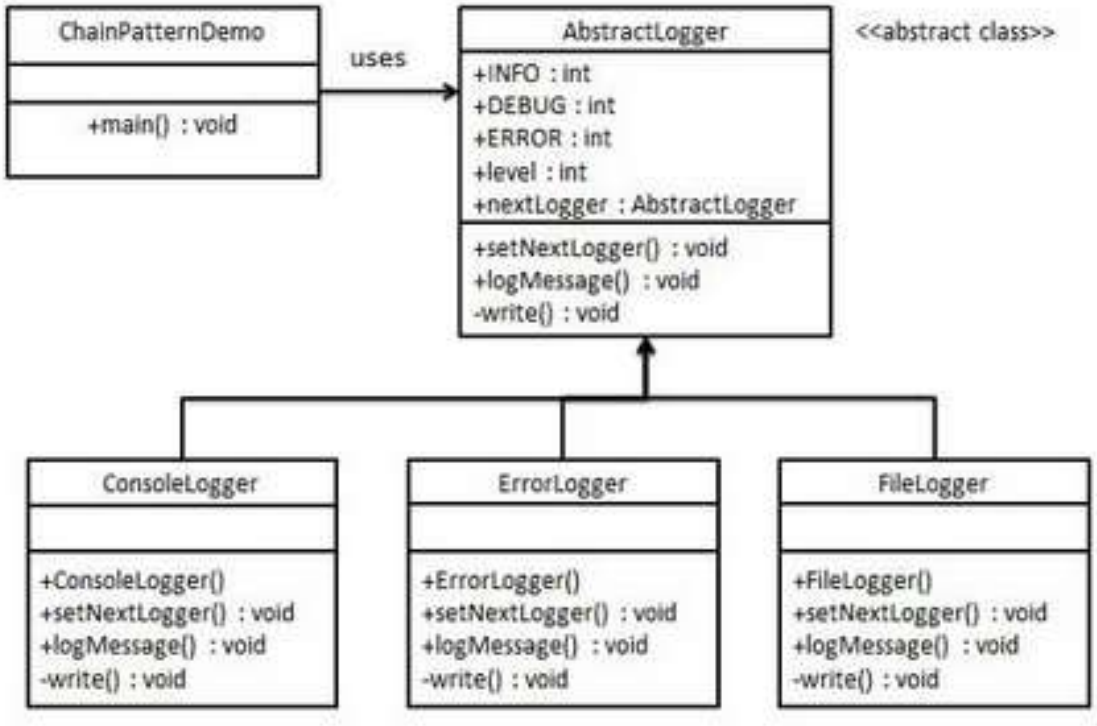
- Se convierten los comportamientos particulares en objetos llamados handlers.
- Se enlazan los hanlders en una cadena.
- Cada handler tiene un campo para almacenar la referencia al siguiente handler en la cadena

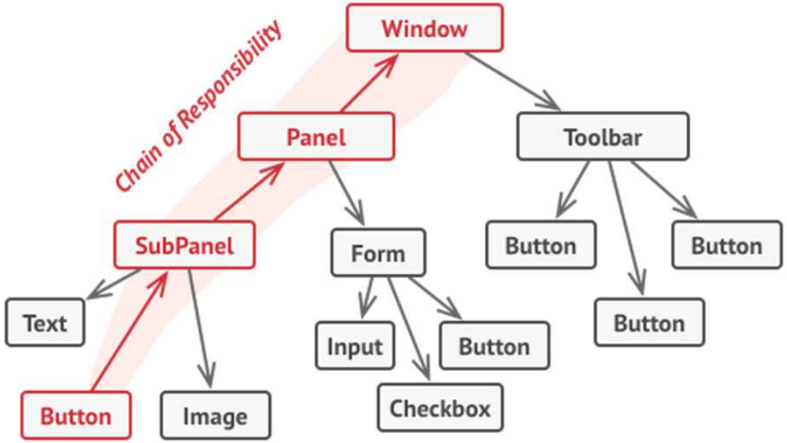






Chain of Responsibility





Chain of Responsibility

AbstractLogger.java

```
public abstract class AbstractLogger {
   public static int INFO = 1;
   public static int DEBUG = 2;
   public static int ERROR = 3;

   protected int level;

   //next element in chain or responsibility
   protected AbstractLogger nextLogger;

   public void setNextLogger(AbstractLogger nextLogger){
      this.nextLogger = nextLogger;
   }

   public void logMessage(int level, String message){
      if(this.level <= level){
         write(message);
      }
      if(nextLogger !=null){
         nextLogger.logMessage(level, message);
      }
   }

   abstract protected void write(String message);

}
```

ErrorLogger.java

```
public class ErrorLogger extends AbstractLogger {

   public ErrorLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
      System.out.println("Error Console::Logger: " + message);
   }
}
```

FileLogger.java

```
public class FileLogger extends AbstractLogger {

   public FileLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
      System.out.println("File::Logger: " + message);
   }
}
```

ConsoleLogger.java

```
public class ConsoleLogger extends AbstractLogger {

   public ConsoleLogger(int level){
      this.level = level;
   }

   @Override
   protected void write(String message) {
      System.out.println("Standard Console::Logger: " + message);
   }
}
```

Chain of Responsibility

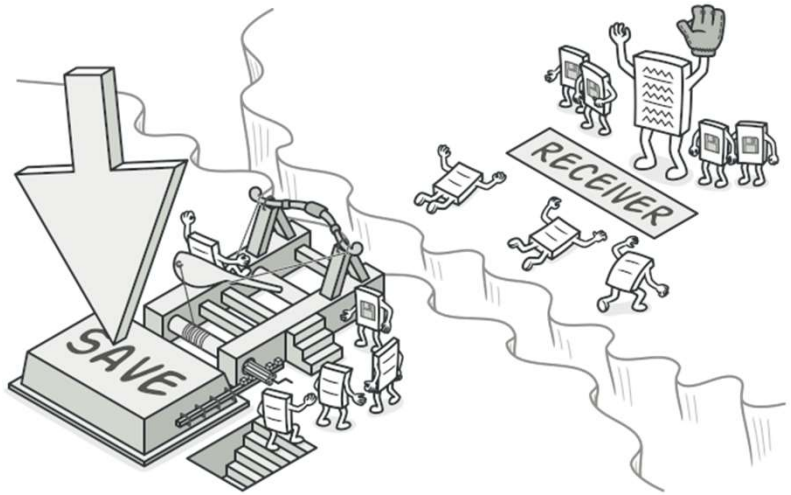
ChainPatternDemo.java

```
public class ChainPatternDemo {

   private static AbstractLogger getChainOfLoggers(){

      AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
      AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
      AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO)

      errorLogger.setNextLogger(fileLogger);
      fileLogger.setNextLogger(consoleLogger);

      return errorLogger;
   }

   public static void main(String[] args) {
      AbstractLogger loggerChain = getChainOfLoggers();

      loggerChain.logMessage(AbstractLogger.INFO,
         "This is an information.");

      loggerChain.logMessage(AbstractLogger.DEBUG,
         "This is an debug level information.");

      loggerChain.logMessage(AbstractLogger.ERROR,
         "This is an error information.");
   }
}
```

Chain of Responsibility

```
interface Chain {
    public abstract void setNext(Chain nextInChain);
    public abstract void process(Number request);
}

class Number {
    private int number;

    public Number(int number) {
        this.number = number;
    }

    public int getNumber() {
        return number;
    }

}

class NegativeProcessor implements Chain {
    private Chain nextInChain;

    public void setNext(Chain c) {
        nextInChain = c;
    }

    public void process(Number request) {
        if (request.getNumber() < 0) {
            System.out.println("NegativeProcessor : " + request.getNumber());
        }
        else{
            nextInChain.process(request);
        }
    }
}

class ZeroProcessor implements Chain {

    private Chain nextInChain;

    public void setNext(Chain c) {
        nextInChain = c;
    }

    public void process(Number request) {
        if (request.getNumber() == 0) {
            System.out.println("ZeroProcessor : " + request.getNumber());
        }
        else{
            nextInChain.process(request);
        }
    }
}
```

```
class PositiveProcessor implements Chain {

    private Chain nextInChain;

    public void setNext(Chain c) {
        nextInChain = c;
    }

    public void process(Number request) {
        if (request.getNumber() > 0) {
            System.out.println("PositiveProcessor : " + request.getNumber());
        }
        else{
            nextInChain.process(request);
        }
    }
}

class TestChain {
    public static void main(String[] args) {
        //configure Chain of Responsibility
        Chain c1 = new NegativeProcessor();
        Chain c2 = new ZeroProcessor();
        Chain c3 = new PositiveProcessor();
        c1.setNext(c2);
        c2.setNext(c3);

        //calling chain of responsibility
        c1.process(new Number(90));
        c1.process(new Number(-50));
        c1.process(new Number(0));
        c1.process(new Number(91));
    }
}
```



Behavioral Design Patterns
Command

Desacoplar una abstracción de su implementación, para que ambas varíen de forma independiente.

Es utilizado para crear objetos que representan acciones y eventos en una aplicación.

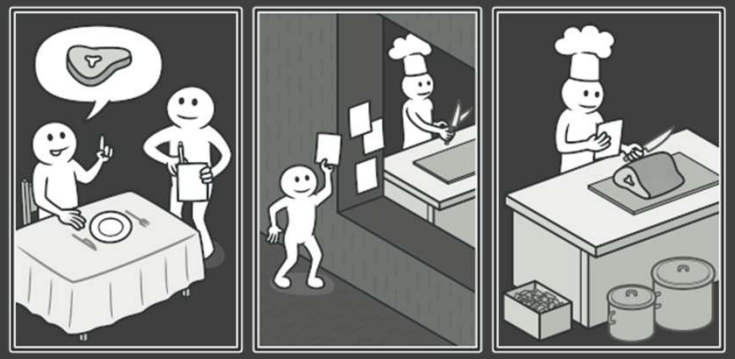
Un objeto Command encapsula un acción o evento y contiene toda la información requerida para entender exactamente que esta sucediendo.

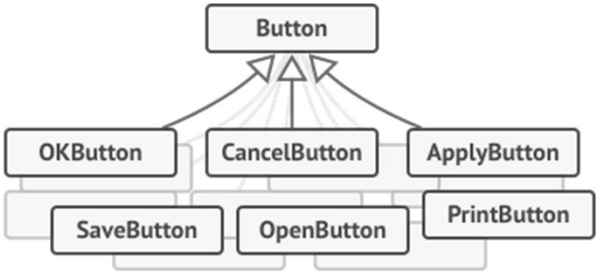
Command

Aplicabilidad

- Cuando se requiere una acción que puede ser representada de varias maneras, como un menú drop down, botones o un menú popup.
- Para crear la funcionalidad undo/redo.

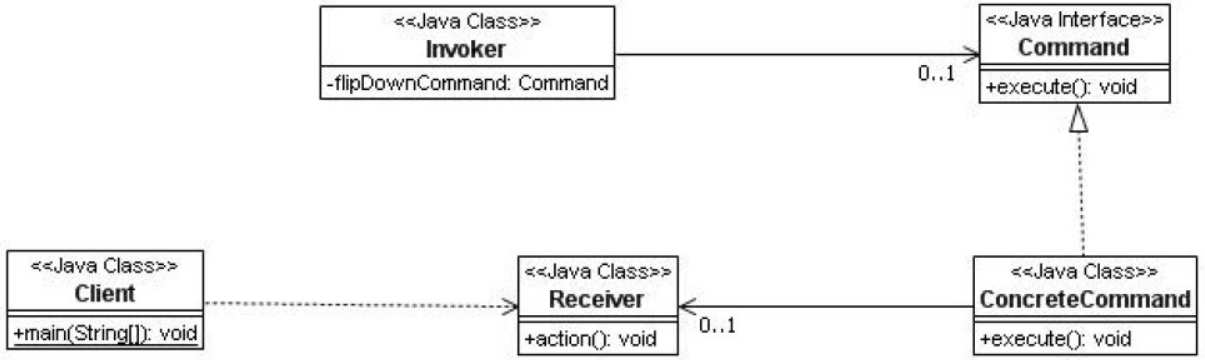
Command



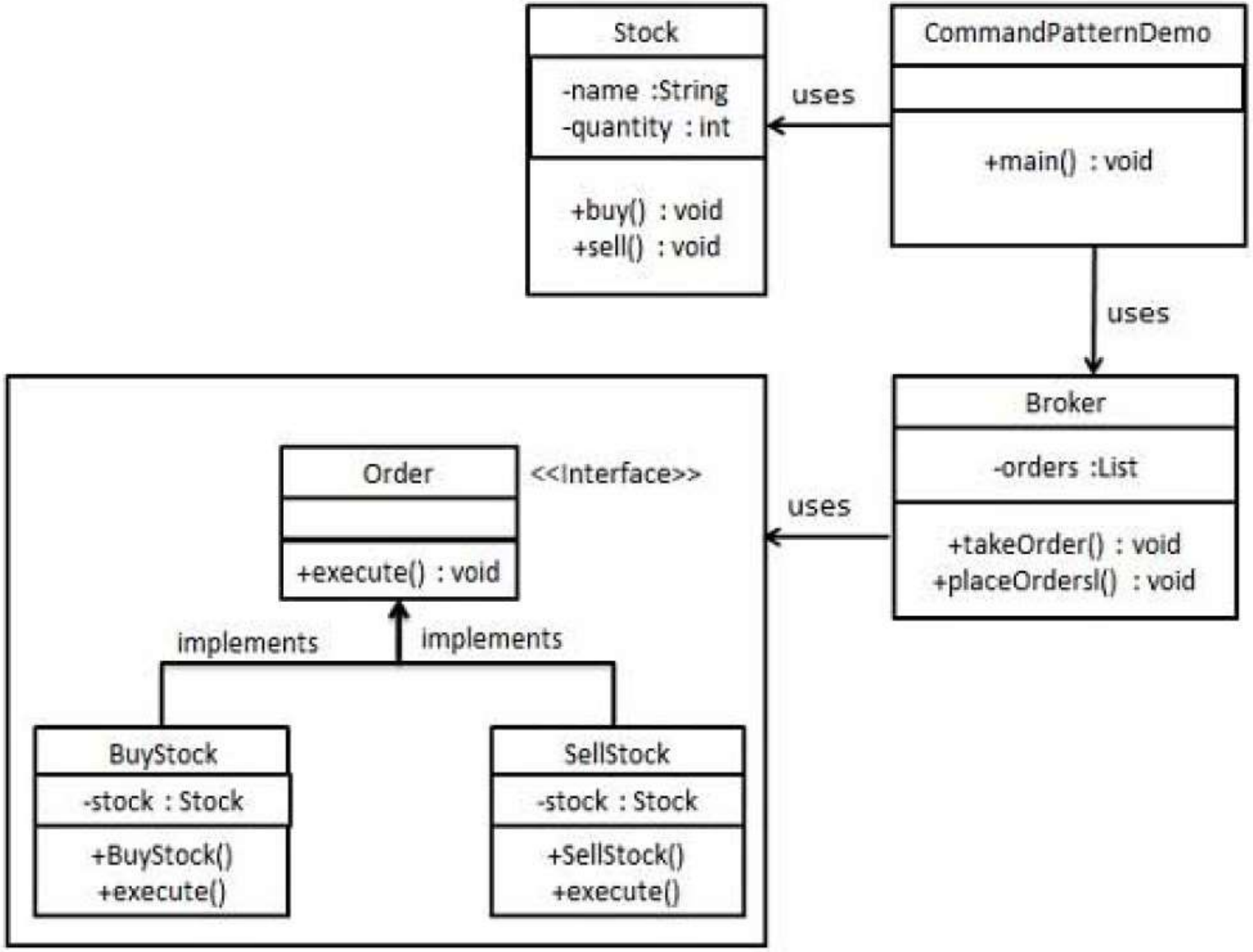

Button
OKButton
CancelButton
ApplyButton
SaveButton
OpenButton
PrintButton


SaveButton
Code
SaveMenuItem
Code
SaveShortcut
Code


User Interface
Business Logic
Button
MenuItem
Shortcut
update(2, "John Smith")
update(2, "John Smith")
update(2, "John Smith")


<<Java Class>>
Invoker
-flipDownCommand: Command
0..1
<<Java Interface>>
Command
+execute(): void
<<Java Class>>
Client
+main(String[]): void
<<Java Class>>
Receiver
+action(): void
0..1
<<Java Class>>
ConcreteCommand
+execute(): void

Command


Stock
-name :String
-quantity : int
+buy() : void
+sell() : void
uses
CommandPatternDemo
+main() : void
uses
Broker
-orders :List
+takeOrder() : void
+placeOrdersl() : void
uses
Order
<<Interface>>
+execute() : void
implements
implements
BuyStock
-stock : Stock
+BuyStock()
+execute()
SellStock
-stock : Stock
+SellStock()
+execute()

Command

Order.java

```
public interface Order {
  void execute();
}
```

BuyStock.java

```
public class BuyStock implements Order {
  private Stock abcStock;
  public BuyStock(Stock abcStock){
    this.abcStock = abcStock;
  }
  public void execute() {
    abcStock.buy();
  }
}
```

Stock.java

```
public class Stock {
  private String name = "ABC";
  private int quantity = 10;

  public void buy(){
    System.out.println("Stock [ Name: "+name+",
      Quantity: " + quantity +" ] bought");
  }
  public void sell(){
    System.out.println("Stock [ Name: "+name+",
      Quantity: " + quantity +" ] sold");
  }
}
```

Command

SellStock.java

```
public class SellStock implements Order {
  private Stock abcStock;

  public SellStock(Stock abcStock){
    this.abcStock = abcStock;
  }
  public void execute() {
    abcStock.sell();
  }
}
```

Broker.java

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

  public class Broker {
  private List<Order> orderList = new ArrayList<Order>();
  public void takeOrder(Order order){
    orderList.add(order);
  }
  public void placeOrders(){
      for (Order order : orderList) {
      order.execute();
    }
    orderList.clear();
  }
}
```

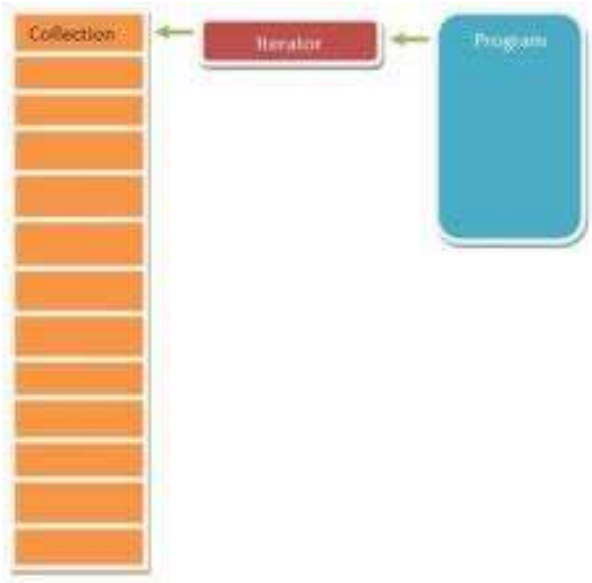
Command

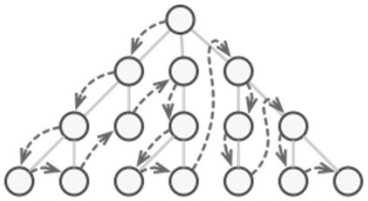
CommandPatternDemo.java

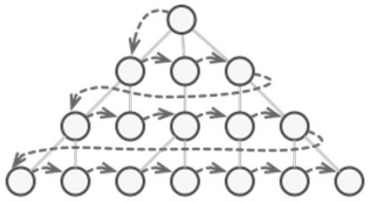
```
public class CommandPatternDemo {
   public static void main(String[] args) {
      Stock abcStock = new Stock();
      BuyStock buyStockOrder = new BuyStock(abcStock);
      SellStock sellStockOrder = new SellStock(abcStock);
      Broker broker = new Broker();
      broker.takeOrder(buyStockOrder);
      broker.takeOrder(sellStockOrder);
      broker.placeOrders();
   }
}
```

Command

```
// A simple Java program to demonstrate
// implementation of Command Pattern using
// a remote control example.

// An interface for command
interface Command {
   public void execute();
}

// Light class and its corresponding command
// classes
class Light {
   public void on() {
      System.out.println("Light is on");
   }
   public void off() {
      System.out.println("Light is off");
   }
}
class LightOnCommand implements Command {
   Light light;

   // The constructor is passed the light it
   // is going to control.
   public LightOnCommand(Light light) {
     this.light = light;
   }
   public void execute() {
     light.on();
   }
}
class LightOffCommand implements Command {
   Light light;
   public LightOffCommand(Light light) {
      this.light = light;
   }
   public void execute() {
       light.off();
   }
}

// Stereo and its command classes
class Stereo {
   public void on() {
      System.out.println("Stereo is on");
   }
   public void off() {
      System.out.println("Stereo is off");
   }
   public void setCD() {
      System.out.println("Stereo is set " +
                  "for CD input");
   }
   public void setDVD() {
      System.out.println("Stereo is set"+
                  " for DVD input");
   }
   public void setRadio() {
      System.out.println("Stereo is set" +
                  " for Radio");
   }
   public void setVolume(int volume) {
     // code to set the volume
     System.out.println("Stereo volume set"
                  + " to " + volume);
   }
}
class StereoOffCommand implements Command {
   Stereo stereo;
   public StereoOffCommand(Stereo stereo) {
      this.stereo = stereo;
   }
   public void execute() {
     stereo.off();
   }
}
class StereoOnWithCDCommand implements Command{
    Stereo stereo;
    public StereoOnWithCDCommand(Stereo stereo) {
       this.stereo = stereo;
    }
    public void execute() {
       stereo.on();
       stereo.setCD();
       stereo.setVolume(11);
    }
}

// A Simple remote control with one button
class SimpleRemoteControl {
   Command slot;  // only one button

   public SimpleRemoteControl() {
   }

   public void setCommand(Command command) {
      // set the command the remote will
      // execute
      slot = command;
   }

   public void buttonWasPressed() {
      slot.execute();
   }
}

// Driver class
class RemoteControlTest {
   public static void main(String[] args) {
      SimpleRemoteControl remote =
            new SimpleRemoteControl();
      Light light = new Light();
      Stereo stereo = new Stereo();

      // we can change command dynamically
      remote.setCommand(new
            LightOnCommand(light));
      remote.buttonWasPressed();
      remote.setCommand(new
          StereoOnWithCDCommand(stereo));
      remote.buttonWasPressed();
      remote.setCommand(new
            StereoOffCommand(stereo));
      remote.buttonWasPressed();
    }
}
```







Behavioral Design Patterns

Iterator

Proporciona una manera de acceder a los elementos de un objeto agregado secuencialmente sin exponer su representación subyacente.

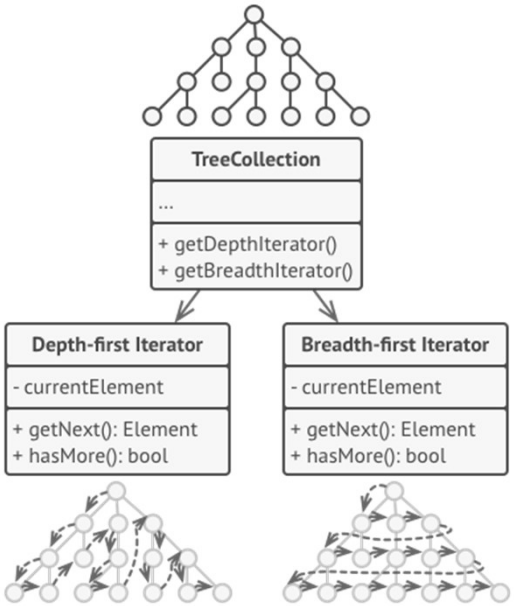
Iterator

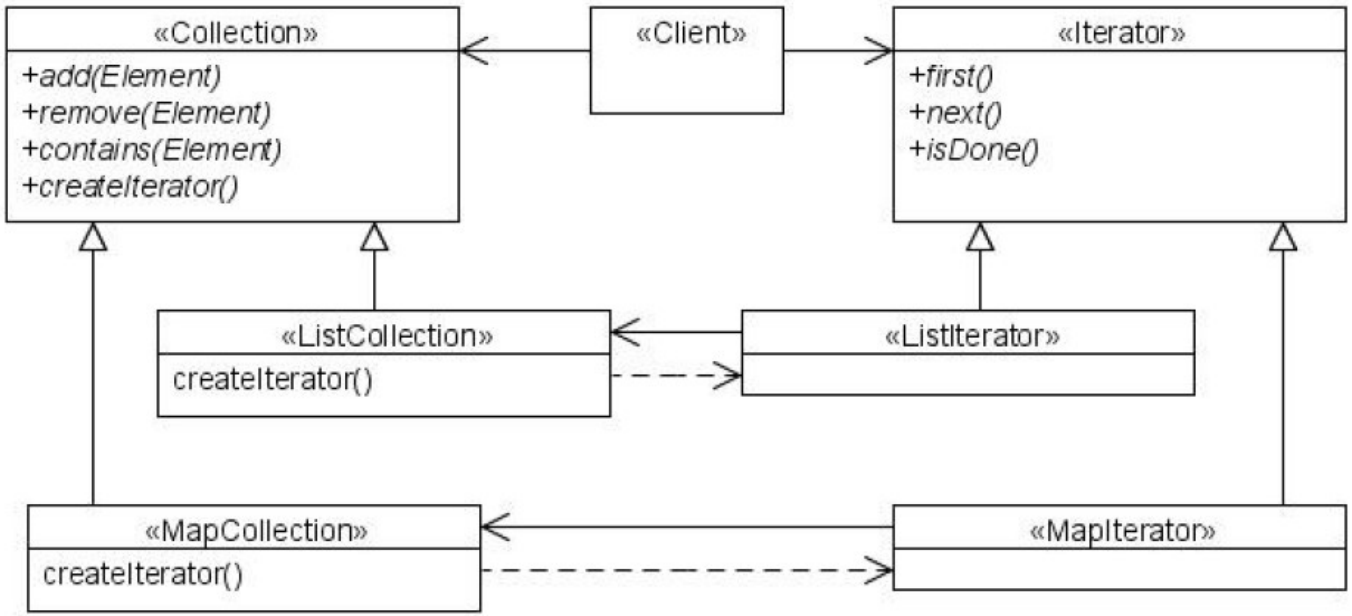
Aplicabilidad

Utilizado para acceder a los elementos de un objeto agregado secuencialmente.

Las colecciones de Java como ArrayList y HashMap tienen implementado este patrón.

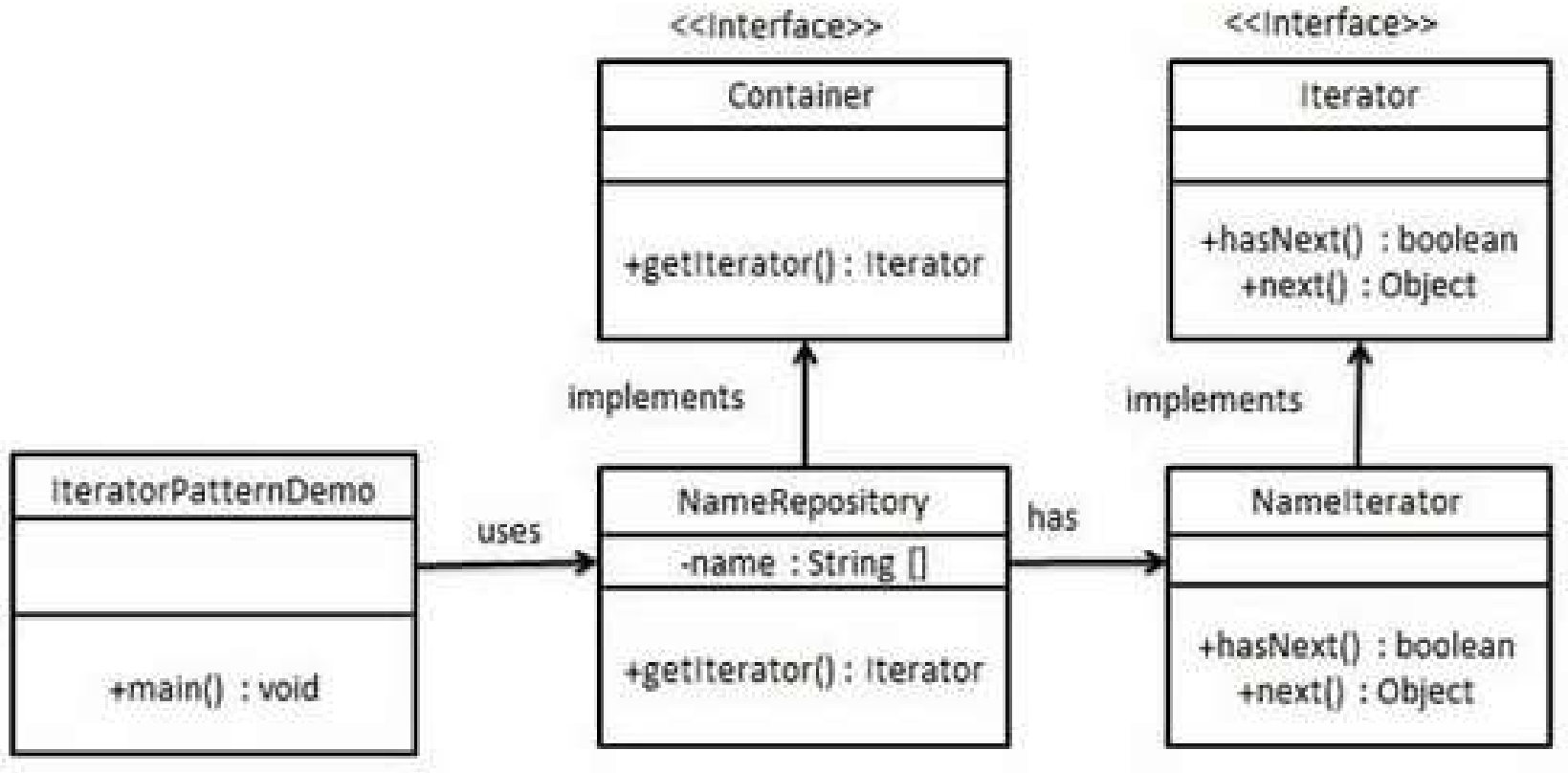
Iterator


TreeCollection
...
+ getDepthIterator()
+ getBreadthIterator()
Depth-first Iterator
- currentElement
+ getNext(): Element
+ hasMore(): bool
Breadth-first Iterator
- currentElement
+ getNext(): Element
+ hasMore(): bool


«Collection»
+add(Element)
+remove(Element)
+contains(Element)
+createIterator()
«Client»
«Iterator»
+first()
+next()
+isDone()
«ListCollection»
createIterator()
«ListIterator»
«MapCollection»
createIterator()
«MapIterator»

Iterator

Proporciona una manera de acceder a los elementos de un objeto agregado secuencialmente sin exponer su representación subyacente



Iterator

Container.java

```
public interface Container {
   public Iterator getIterator();
}
```

Iterator.java

```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
```

Iterator

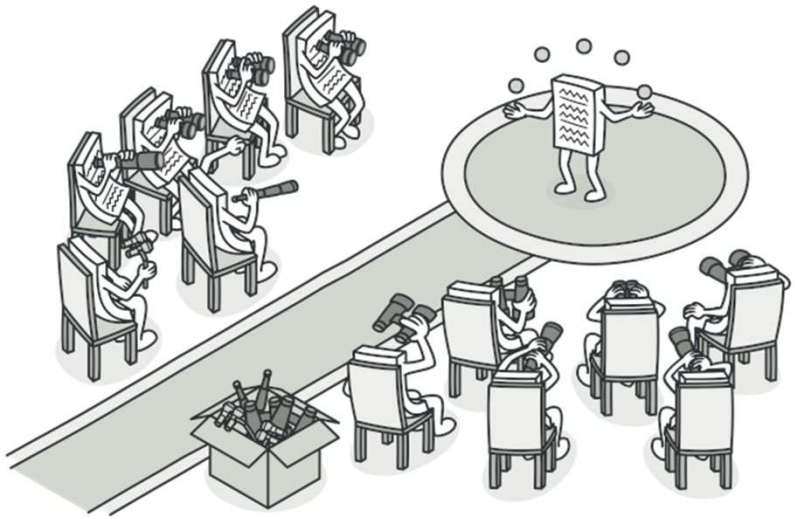
NameRepository.java

```
public class NameRepository implements Container {
   public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

   @Override
   public Iterator getIterator() {
      return new NameIterator();
   }

   private class NameIterator implements Iterator {

      int index;

      @Override
      public boolean hasNext() {

         if(index < names.length){
            return true;
         }
         return false;
      }

      @Override
      public Object next() {

         if(this.hasNext()){
            return names[index++];
         }
         return null;
      }
   }
}
```

IteratorPatternDemo.java

```
public class IteratorPatternDemo {

   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();

      for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
         String name = (String)iter.next();
         System.out.println("Name : " + name);
      }
   }
}
```

```
Name : Robert
Name : John
Name : Julie
Name : Lora
```



Behavioral Design Patterns

Observer

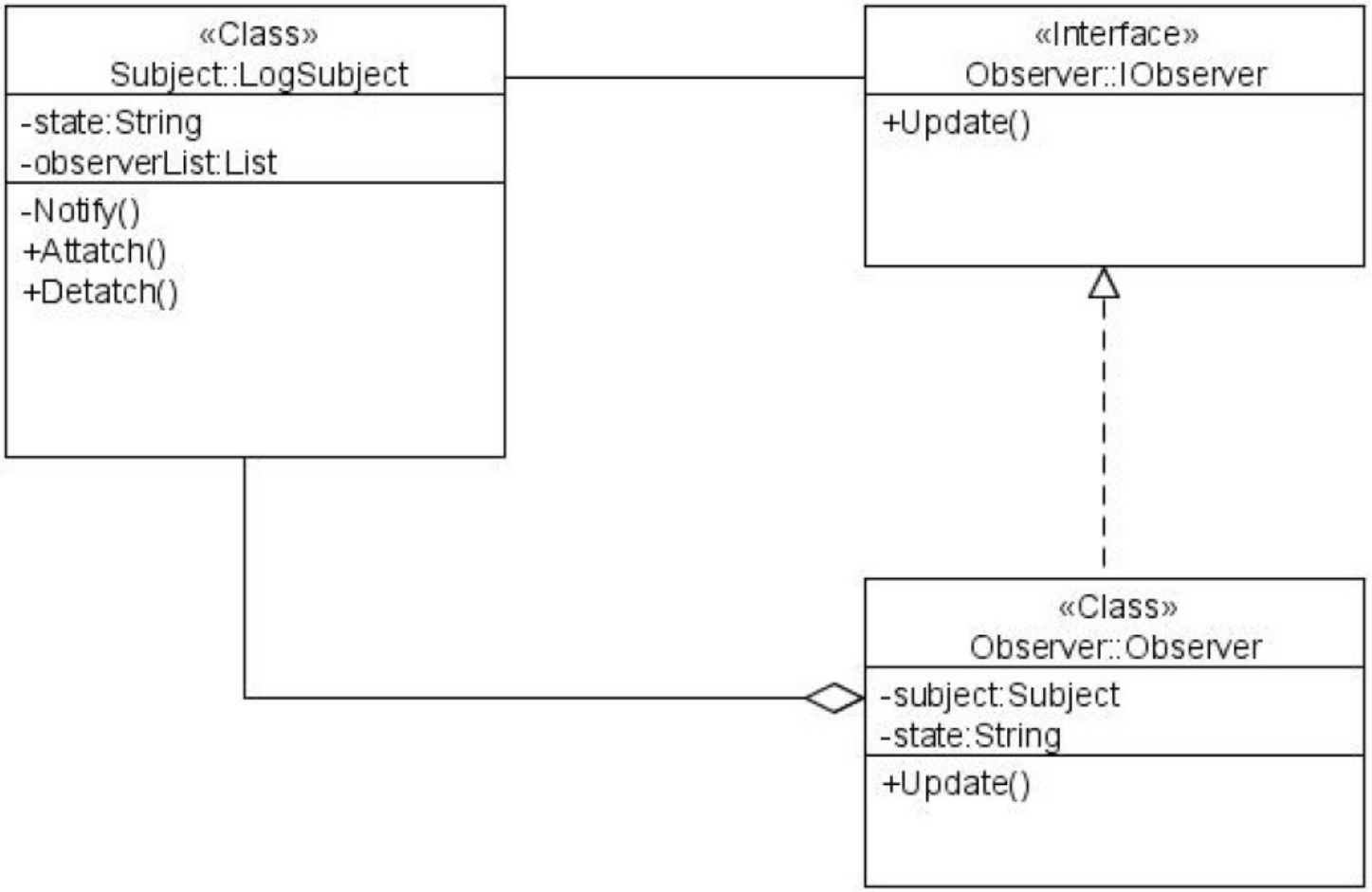
Define un mecanismo de subscripción para notificar a múltiples objetos acerca de eventos que le suceden al objeto que están observando .

Observer

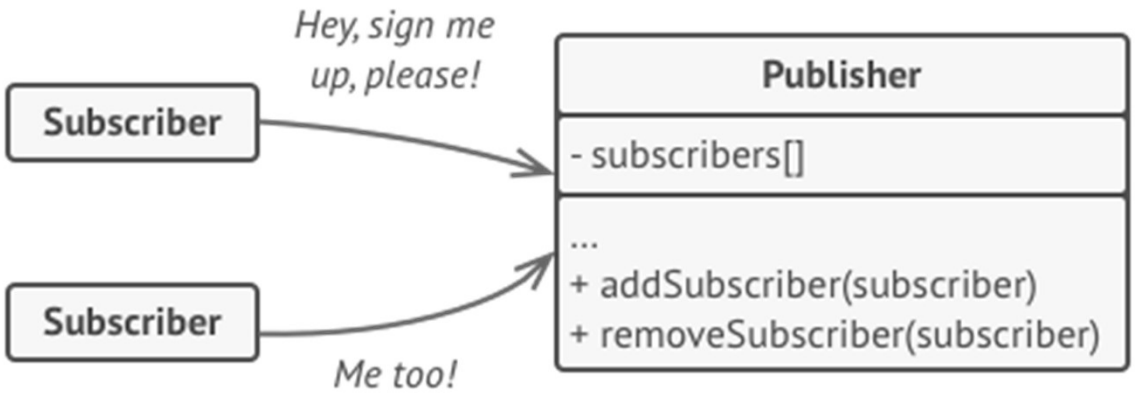
Aplicabilidad

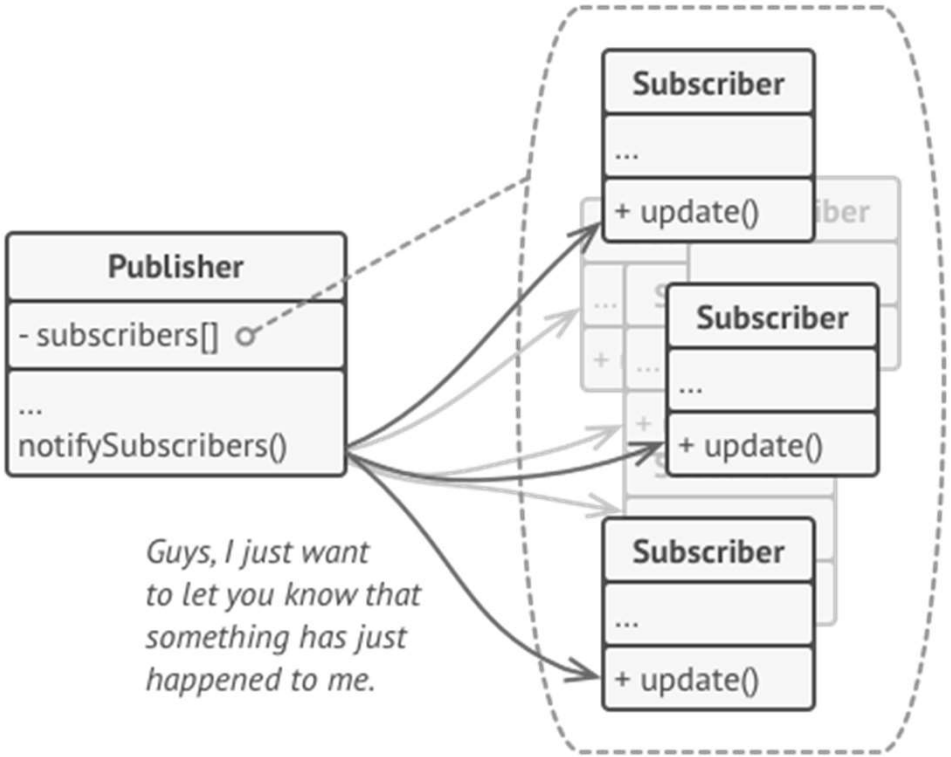
Cuando un objeto requiere publicar información y muchos objetos necesitan recibir la información.

Observer

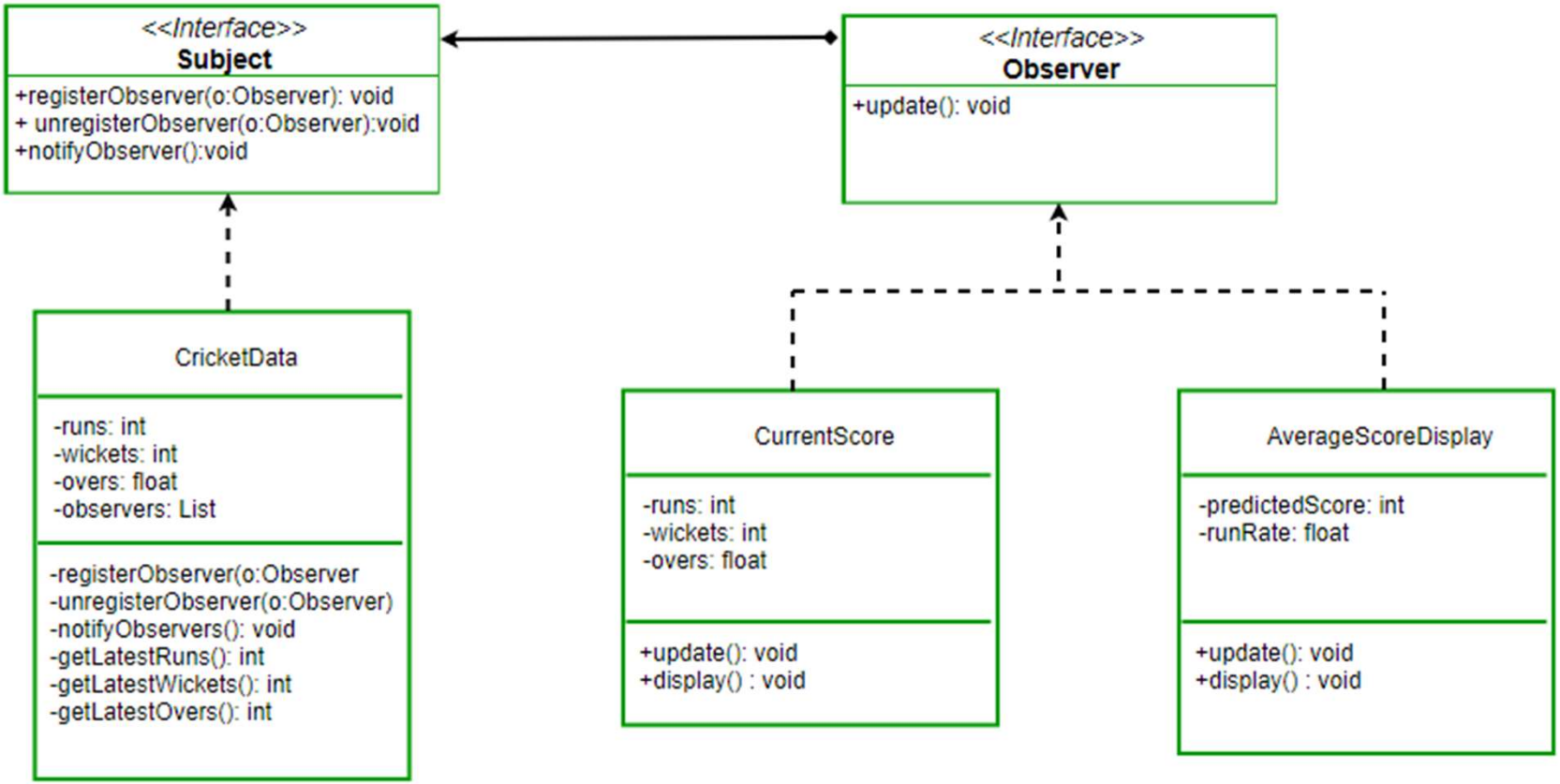

«Class»
Subject::LogSubject
-state:String
-observerList:List
-Notify()
+Attatch()
+Detatch()
«Interface»
Observer::IObserver
+Update()
«Class»
Observer::Observer
-subject:Subject
-state:String
+Update()

Observer


Hey, sign me up, please!
Subscriber
Subscriber
Me too!
Publisher
- subscribers[]
...
+ addSubscriber(subscriber)
+ removeSubscriber(subscriber)

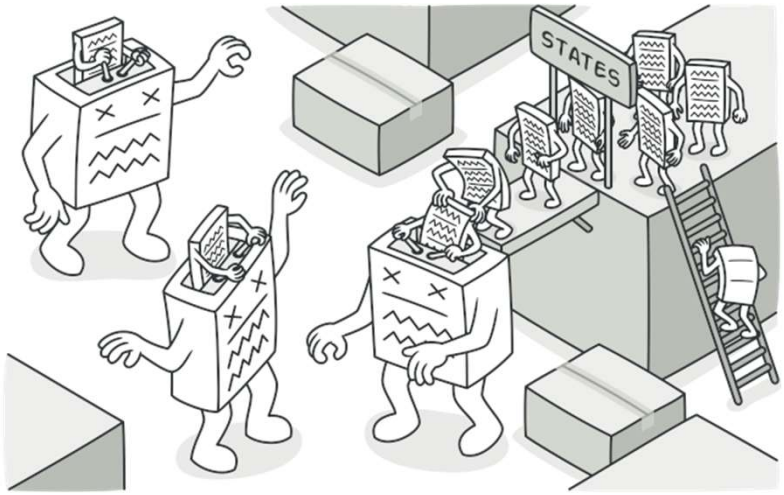

Publisher
- subscribers[]
...
notifySubscribers()
Guys, I just want to let you know that something has just happened to me.
Subscriber
...
+ update()
Subscriber
...
+ update()
Subscriber
...
+ update()

Observer


<<Interface>>
Subject
+registerObserver(o:Observer): void
+ unregisterObserver(o:Observer):void
+notifyObserver():void
<<Interface>>
Observer
+update(): void
CricketData
-runs: int
-wickets: int
-overs: float
-observers: List
-registerObserver(o:Observer
-unregisterObserver(o:Observer)
-notifyObservers(): void
-getLatestRuns(): int
-getLatestWickets(): int
-getLatestOvers(): int
CurrentScore
-runs: int
-wickets: int
-overs: float
+update(): void
+display() : void
AverageScoreDisplay
-predictedScore: int
-runRate: float
+update(): void
+display() : void

Observer

```
// Java program to demonstrate working of
// onserver pattern
import java.util.ArrayList;
import java.util.Iterator;

// Implemented by Cricket data to communicate
// with observers
interface Subject {
    public void registerObserver(Observer o);
    public void unregisterObserver(Observer o);
    public void notifyObservers();
}

class CricketData implements Subject {
    int runs;
    int wickets;
    float overs;
    ArrayList<Observer> observerList;

    public CricketData() {
        observerList = new ArrayList<Observer>();
    }

    @Override
    public void registerObserver(Observer o) {
        observerList.add(o);
    }

    @Override
    public void unregisterObserver(Observer o) {
        observerList.remove(observerList.indexOf(o));
    }

    @Override
    public void notifyObservers() {
        for (Iterator<Observer> it =
             observerList.iterator(); it.hasNext();)
        {
            Observer o = it.next();
            o.update(runs,wickets,overs);
        }
    }

    // get latest runs from stadium
    private int getLatestRuns() {
        // return 90 for simplicity
        return 90;
    }

    // get latest wickets from stadium
    private int getLatestWickets(){
        // return 2 for simplicity
        return 2;
    }

    // get latest overs from stadium
    private float getLatestOvers() {
        // return 90 for simplicity
        return (float)10.2;
    }

    // This method is used update displays
    // when data changes
    public void dataChanged() {
        //get latest data
        runs = getLatestRuns();
        wickets = getLatestWickets();
        overs = getLatestOvers();

        notifyObservers();
    }
}

// This interface is implemented by all those
// classes that are to be updated whenever there
// is an update from CricketData
interface Observer {
    public void update(int runs, int wickets,
                       float overs);
}

class AverageScoreDisplay implements Observer {
    private float runRate;
    private int predictedScore;

    public void update(int runs, int wickets,
                       float overs) {
        this.runRate =(float)runs/overs;
        this.predictedScore = (int)(this.runRate * 50);
        display();
    }

    public void display() {
        System.out.println("\nAverage Score Display: \n"
                + "Run Rate: " + runRate +
                "\nPredictedScore: " +
                predictedScore);
    }
}

class CurrentScoreDisplay implements Observer {
    private int runs, wickets;
    private float overs;

    public void update(int runs, int wickets,
                       float overs) {
        this.runs = runs;
        this.wickets = wickets;
        this.overs = overs;
        display();
    }

    public void display() {
        System.out.println("\nCurrent Score Display:\n"
                    + "Runs: " + runs +
                    "\nWickets:" + wickets +
                    "\nOvers: " + overs );
    }
}

// Driver Class
class Main {
    public static void main(String args[]) {
        // create objects for testing
        AverageScoreDisplay averageScoreDisplay =
                    new AverageScoreDisplay();
        CurrentScoreDisplay currentScoreDisplay =
                    new CurrentScoreDisplay();

        // pass the displays to Cricket data
        CricketData cricketData = new CricketData();

        // register display elements
        cricketData.registerObserver(averageScoreDisplay);
        cricketData.registerObserver(currentScoreDisplay);

        // in real app you would have some logic to
        // call this function when data changes
        cricketData.dataChanged();

        //remove an observer
        cricketData.unregisterObserver(averageScoreDisplay);

        // now only currentScoreDisplay gets the
        // notification
        cricketData.dataChanged();
    }
}
```



Behavioral Design Patterns

States

Permite a un objeto alterar su comportamiento interno cuando cambia su estado, aparece como si el objeto cambiara su clase

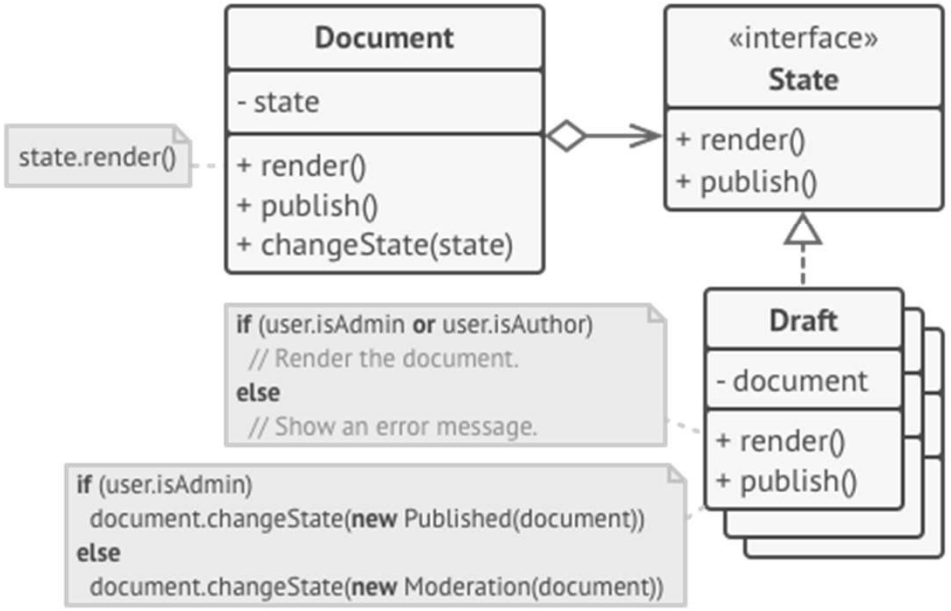
States

Aplicabilidad

- Cuando se requiere definir una clase contexto para presentar una única interface al mundo exterior.
- Cuando se requiere representar diferentes “estados” de una máquina de estados como clases derivadas de la clase base “State”

States


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


Document
- state
+ render()
+ publish()
+ changeState(state)
state.render()
«interface»
State
+ render()
+ publish()
Draft
- document
+ render()
+ publish()
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))

Behavioral Design Patterns

Others

Strategy

Cuando se requiere definir una familia de algoritmos, encapsular cada uno y hacerlos intercambiables. Permite al algoritmo variar independientemente de los clientes que lo utilizan.

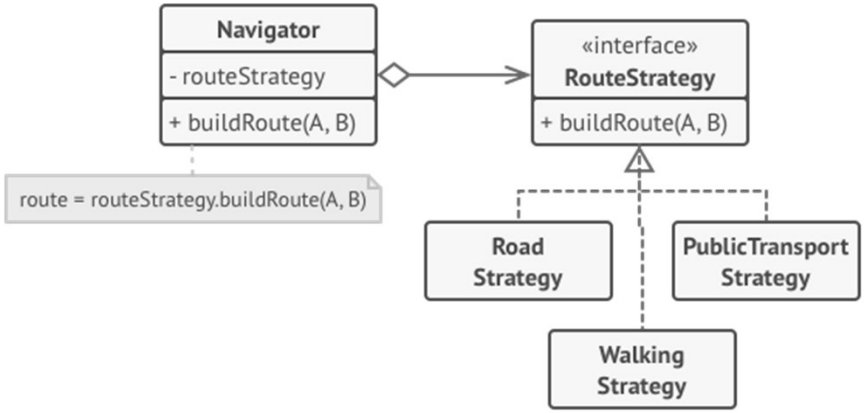
Memento

Para registrar el estado interno de un objeto sin violar la encapsulación y reclamarlo luego sin conocimiento del objeto original. Un memento es un objeto que almacena un “snapshot” del estado interno de otro objeto.

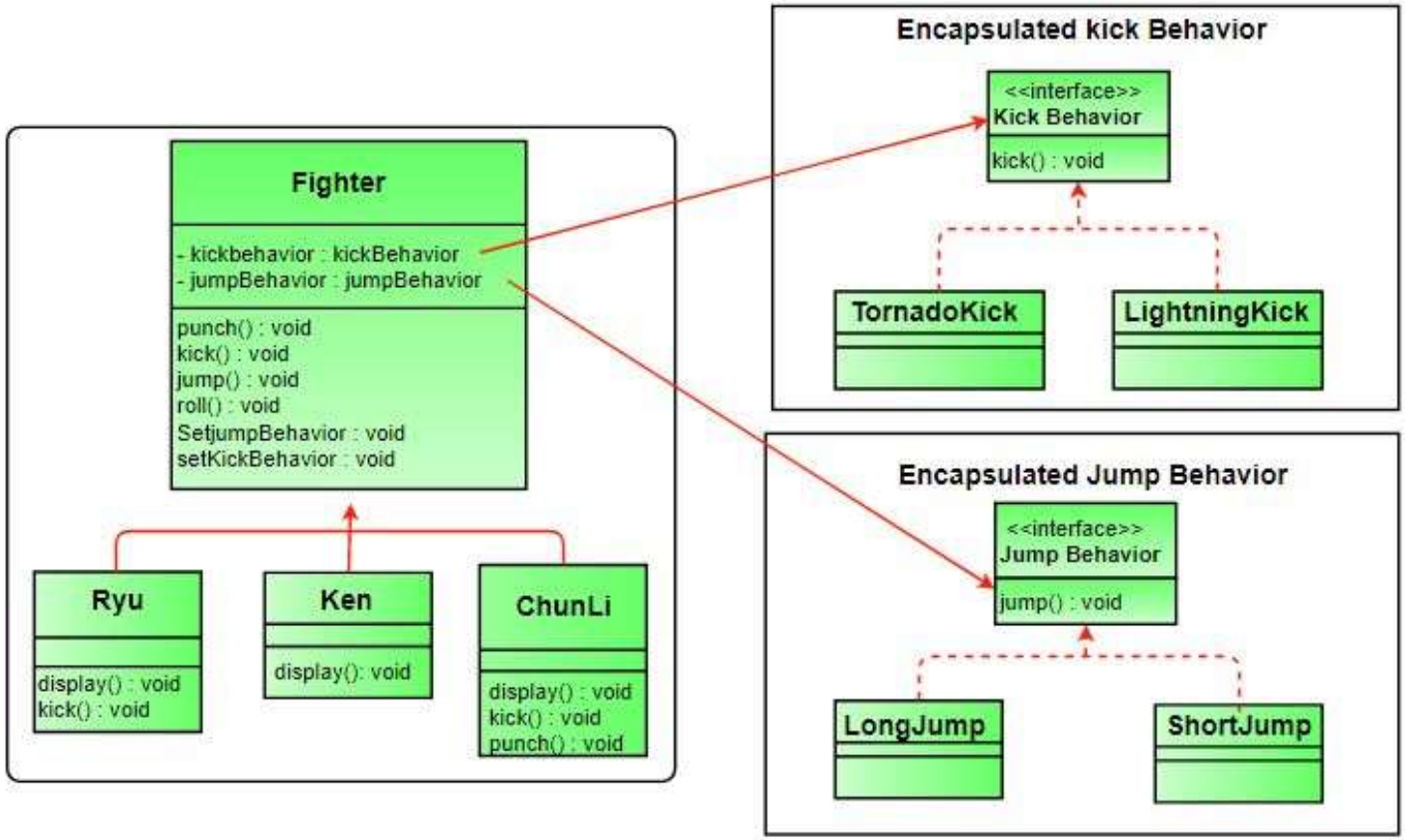
Mediator

La comunicación entre objetos está encapsulada con el objeto Mediator. Los objetos no se comunican directamente unos con otros lo hacen a través del Mediator.

Strategy


Navigator
- routeStrategy
+ buildRoute(A, B)
route = routeStrategy.buildRoute(A, B)
«interface»
RouteStrategy
+ buildRoute(A, B)
Road Strategy
PublicTransport Strategy
Walking Strategy

Strategy

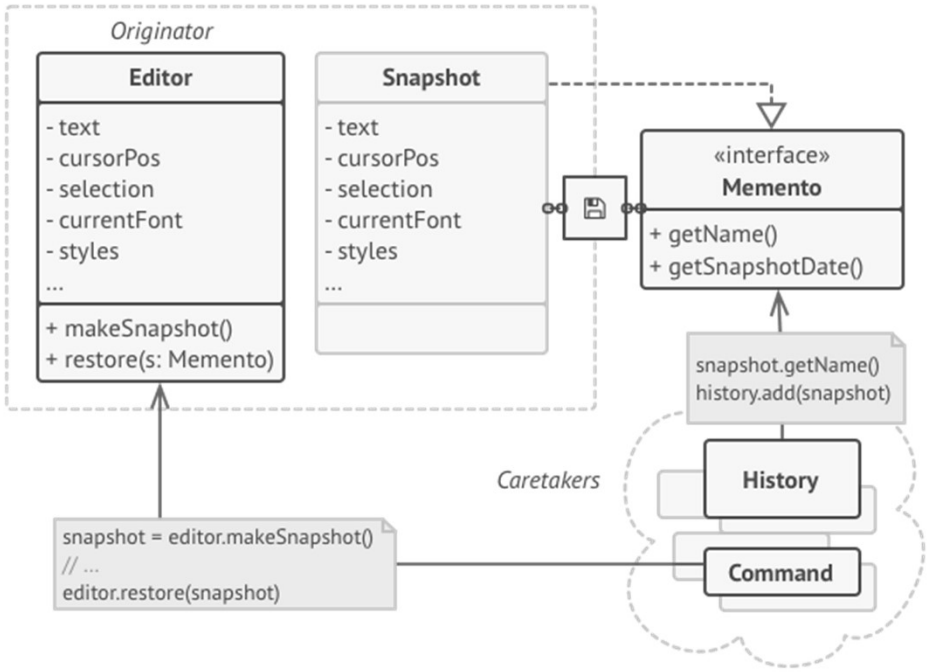

Encapsulated kick Behavior
<<interface>>
Kick Behavior
kick() : void
TornadoKick
LightningKick
Fighter
- kickbehavior : kickBehavior
- jumpBehavior : jumpBehavior
punch() : void
kick() : void
jump() : void
roll() : void
SetjumpBehavior : void
setKickBehavior : void
Ryu
display() : void
kick() : void
Ken
display(): void
ChunLi
display() : void
kick() : void
punch() : void
Encapsulated Jump Behavior
<<interface>>
Jump Behavior
jump() : void
LongJump
ShortJump

Strategy

```
// Java program to demonstrate implementation of
// Strategy Pattern

// Abstract as you must have a specific fighter
abstract class Fighter {
   KickBehavior kickBehavior;
   JumpBehavior jumpBehavior;

   public Fighter(KickBehavior kickBehavior,
            JumpBehavior jumpBehavior) {
      this.jumpBehavior = jumpBehavior;
      this.kickBehavior = kickBehavior;
   }
   public void punch() {
      System.out.println("Default Punch");
   }
   public void kick() {
      // delegate to kick behavior
      kickBehavior.kick();
   }
   public void jump() {

      // delegate to jump behavior
      jumpBehavior.jump();
   }
   public void roll() {
      System.out.println("Default Roll");
   }
   public void setKickBehavior(KickBehavior kickBehavior) {
      this.kickBehavior = kickBehavior;
   }
   public void setJumpBehavior(JumpBehavior jumpBehavior) {
      this.jumpBehavior = jumpBehavior;
   }
   public abstract void display();
}

// Encapsulated kick behaviors
interface KickBehavior {
   public void kick();
}
class LightningKick implements KickBehavior {
   public void kick() {
      System.out.println("Lightning Kick");
   }
}
class TornadoKick implements KickBehavior {
   public void kick() {
      System.out.println("Tornado Kick");
   }
}

// Encapsulated jump behaviors
interface JumpBehavior {
   public void jump();
}
class ShortJump implements JumpBehavior {
   public void jump() {
      System.out.println("Short Jump");
   }
}
class LongJump implements JumpBehavior {
   public void jump() {
      System.out.println("Long Jump");
   }
}

// Characters
class Ryu extends Fighter {
   public Ryu(KickBehavior kickBehavior,
         JumpBehavior jumpBehavior) {
      super(kickBehavior,jumpBehavior);
   }
   public void display() {
      System.out.println("Ryu");
   }
}
class Ken extends Fighter {
   public Ken(KickBehavior kickBehavior,
         JumpBehavior jumpBehavior) {
      super(kickBehavior,jumpBehavior);
   }
   public void display() {
      System.out.println("Ken");
   }
}
class ChunLi extends Fighter {
   public ChunLi(KickBehavior kickBehavior,
          JumpBehavior jumpBehavior) {
      super(kickBehavior,jumpBehavior);
   }
   public void display()
   {
      System.out.println("ChunLi");
   }
}

// Driver class
class StreetFighter {
   public static void main(String args[]) {
      // let us make some behaviors first
      JumpBehavior shortJump = new ShortJump();
      JumpBehavior LongJump = new LongJump();
      KickBehavior tornadoKick = new TornadoKick();

      // Make a fighter with desired behaviors
      Fighter ken = new Ken(tornadoKick,shortJump);
      ken.display();

      // Test behaviors
      ken.punch();
      ken.kick();
      ken.jump();

      // Change behavior dynamically (algorithms are
      // interchangeable)
      ken.setJumpBehavior(LongJump);
      ken.jump();
   }
}
```

Memento


HISTORY


Originator
Editor
- text
- cursorPos
- selection
- currentFont
- styles
...
+ makeSnapshot()
+ restore(s: Memento)
Snapshot
- text
- cursorPos
- selection
- currentFont
- styles
...
«interface»
Memento
+ getName()
+ getSnapshotDate()
snapshot.getName()
history.add(snapshot)
Caretakers
History
Command
snapshot = editor.makeSnapshot()
// ...
editor.restore(snapshot)

Behavioral Design Patterns

Others

Visitor

Permite definir una operación sobre objetos de una jerarquía de clases sin modificar las clases sobre las que opera. Representa una operación que se realiza sobre los elementos que conforman la estructura de un objeto.

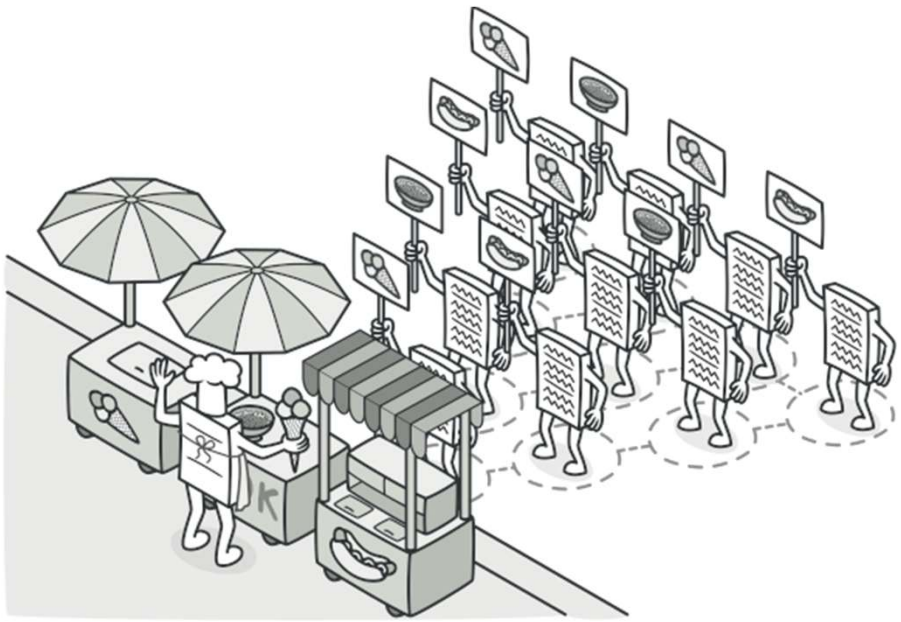
Interpreter

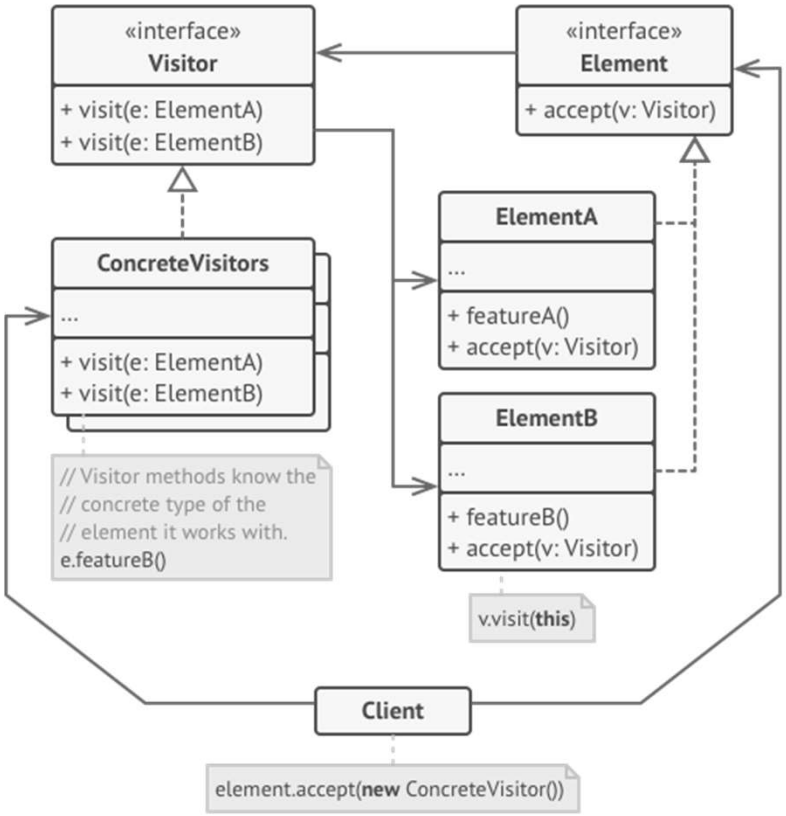
Define la representación de la gramática de un lenguaje junto con un intérprete.

Template

Define el esqueleto programático de un algoritmo. Uno o más de los pasos del algoritmo puede ser sobreescrito para permitir comportaminetos diferentes mientras que se asegura que el algoritmo como tal es seguido.

Visitor




«interface»
Visitor
+ visit(e: ElementA)
+ visit(e: ElementB)
«interface»
Element
+ accept(v: Visitor)
ConcreteVisitors
...
+ visit(e: ElementA)
+ visit(e: ElementB)
// Visitor methods know the
// concrete type of the
// element it works with.
e.featureB()
ElementA
...
+ featureA()
+ accept(v: Visitor)
ElementB
...
+ featureB()
+ accept(v: Visitor)
v.visit(this)
Client
element.accept(new ConcreteVisitor())

RESUMEN

Recordemos

Behavioral Design Patterns facilitan el manejo de algoritmos y la asignación de responsabilidades entre objetos.

REFERENCIAS

Para profundizar

- Design Patterns- Libro de Erich Gamma, John Vlissides, Ralph Johnson y Richard Helm.
- http://www.blackwasp.co.uk/gofpatterns.aspx
- http://www.w3sdesign.com/

PREGRADO

Ingeniería de Software

Escuela de Ingeniería de Sistemas y Computación | Facultad de Ingeniería

UPC
Universidad Peruana
de Ciencias Aplicadas

Prolongación Primavera 2390,
Monterrico, Santiago de Surco
Lima 33 - Perú
T 511 313 3333
https://www.upc.edu.pe

exígete, innova