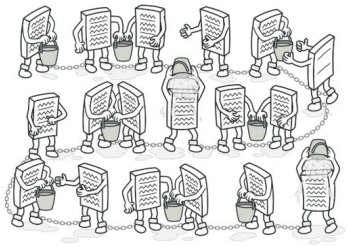
Padrões de Comportamentais

Exemplos Práticos

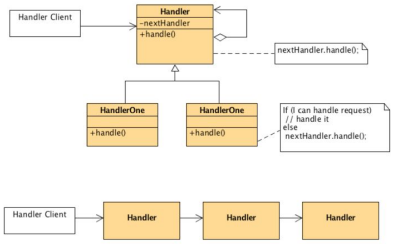
Slides + https://refactoring.guru/design-patterns

Carina Neves 90451

Chain of Responsibility:



Permite passar solicitações ao longo de uma cadeia de manipuladores. Ao receber uma solicitação, cada manipulador decide processar a solicitação ou transmiti-la ao próximo manipulador da cadeia.

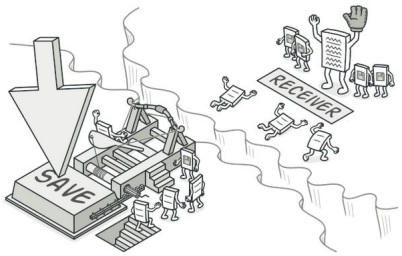



```
abstract class Parser {
	private Parser successor = null;
	public void parse(String fileName) {
		if (successor != null)
			successor.parse(fileName);
		else
			System. out.println("No parser for the file: " + fileName);
	}
	protected boolean canHandleFile(String fileName, String format) {
		return (fileName == null) || (fileName.endsWith(format));
	}
	public Parser setSuccessor(Parser successor) {
		this.successor = successor;
		return this;
	}
}

class JsonParser extends Parser {
	@Override
	public void parse(String fileName) {
		if (canHandleFile(fileName, ".json"))
			System. out.println("A JSON parser for: " + fileName);
		else
			super.parse(fileName);
	}
}

class CsvParser extends Parser {
	@Override
	public void parse(String fileName) {
		if (canHandleFile(fileName, ".csv"))
			System. out .println("A CSV parser for: " + fileName);
		else
			super.parse(fileName);
	}
}
```

```
class TextParser extends Parser {
	@Override
	public void parse(String fileName) {
		if (canHandleFile(fileName, ".txt")) {
			System. out.println("A text parser for: " + fileName);
		} else {
			super.parse(fileName);
		}
	}
}

public class ChainOfResponsibilityDemo {
	public static void main(String[] args) {
		List<String> fileList = new ArrayList<>();
		fileList.add("someFile.txt");
		fileList.add("otherFile.json");
		fileList.add("csvFile.csv");
		fileList.add("somethingelse.doc");
		Parser textParser =
			new CsvParser().setSuccessor(
			new TextParser().setSuccessor(
			new JsonParser()));
		for (String fileName : fileList) {
			textParser.parse(fileName);
		}
	}
}
```

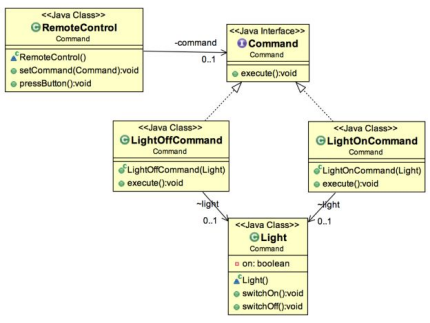
```
A text parser for: someFile.txt
A JSON parser for: otherFile.json
A CSV parser for: csvFile.csv
No parser for the file: somethingelse.doc
```

Command:



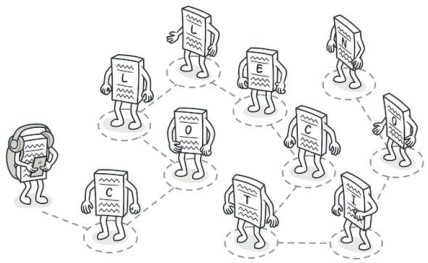
Transforma uma solicitação em um objeto independente que contém todas as informações sobre a solicitação. Essa transformação permite parametrizar métodos com diferentes solicitações, atrasar ou enfileirar a execução de uma solicitação e dar suporte a operações que podem ser desfeitas.

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



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

//Command
interface Command {
	public void execute();
}

// Concrete Command
class LightOnCommand implements Command {
	// reference to the light
	Light light;
	public LightOnCommand(Light light) { this.light = light; }
	public void execute() { light.switchOn(); }
}

// Concrete Command
class LightOffCommand implements Command {
	// reference to the light
	Light light;
	public LightOffCommand(Light light) { this.light = light; }
	public void execute() { light.switchOff(); }
}

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

Iterator:



Permite percorrer elementos de uma coleção sem expor sua representação subjacente (lista, pilha, árvore etc.).

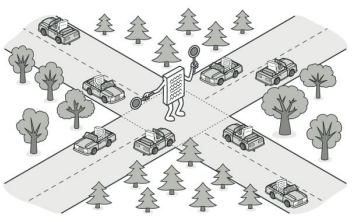
```
//Classe Item, representa um item num menu. Um item tem um nome e um preço.
public class Item{
	String nome;
	float preço;
	public Item(String nome, float preço){
		this.nome = nome;
		this.preço = preço;
	}
	public String toString(){
		return nome + ":$" + preço;
	}
}
```

Menu.java

```
//Classe Menu tem lista de itens do tipo Item. Os items podem ser adicionados
//através do método addItem( ). O método iterador( ) retorna um iterado de itens de
menu.
//A classe MenuIterator é uma classe interna de Menu que implementa a interface de
//iterado para objetos de Item.

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Menu{
	List<Item> menuItems;
	public Menu(){
		menuItems = new ArrayList<Item>();
	}
	public void addItem(Item item){
		menuItems.add(item);
	}
	public Iterator<Item> iterator(){
		return new MenuIterator();
	}
	class MenuIterator implements Iterator<Item>{
		int currentIndex = 0;
```

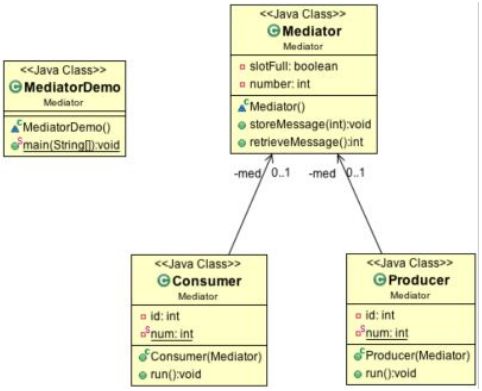
```
		@Override public boolean hasNext(){
		if(currentIndex >= menuItems.size()){
			return false;
		} else return true;
	}
	@Override public Item next(){
		return menuItems.get(currentIndex++);
	}
	@Override public void remove(){
		menuItems.remove(--currentIndex);
	}
}

Demo.java
public class Demo{
	public static void main(String[] args){
		Item i1 = new Item("spaghetti", 7.50f);
		Item i2 = new Item("hamburger", 6.00f);
		Item i3 = new Item("sandes", 6.50f);
		Menu menu = new Menu();
		menu.addItem(i1);
		menu.addItem(i2);
		menu.addItem(i3);
		System.out.println("Mostrar menu:");
		Iterator<Item> iterator = menu.iterator();
		while(iterator.hasNext()){
			Item item = iterator.next();
			System.out.println(item);
		}
		System.out.println("Removendo o último item retornado:");
		iterator.remove();
		System.out.println("Mostrar menu:");
		iterator = menu.iterator();
		while(iterator.hasNext()){
		Item item = iterator.next();
		System.out.println(item);
	}
}
```

Mediator:



Permite reduzir as dependências caóticas entre os objetos. O padrão restringe as comunicações diretas entre os objetos e força-os a colaborar apenas por meio de um objeto mediador.

```
class Mediator {
	private boolean slotFull = false;
	private int number;
	public synchronized void storeMessage(int num) {
		while (slotFull == true) {
			try {
				wait();
			} catch (InterruptedException e) {
				// ...
			}
		}
		slotFull = true;
		number = num;
		notifyAll();
	}
	public synchronized int retrieveMessage() {
		// ...
	}
}

class Producer extends Thread {
	// 2. Producers are coupled only to the Mediator
	private Mediator med;
	private int id;
	private static int num = 1;
	public Producer(Mediator m) {
		med = m;
		id = num++;
	}
	public void run() {
		int num;
		while (true) {
			med.storeMessage(num = (int) (Math.random() * 100));
			System.out.print("p" + id + "-" + num + " ");
		}
	}
}

class Consumer extends Thread {
	// 3. Consumers are coupled only to the Mediator
	private Mediator med;
	private int id;
	private static int num = 1;
	public Consumer(Mediator m) {
		med = m;
		id = num++;
	}
```

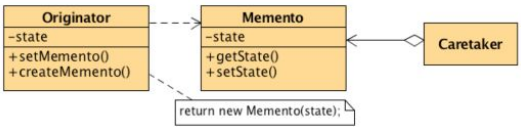


```
        public void run() {
                while (true) {
                System.out.print("c" + id + "-" + med.retrieveMessage() + " “ );
                }
        }
}

class MediatorDemo {
        public static void main(String[] args) {
                Mediator mb = new Mediator();
                new Producer(mb).start();
                new Producer(mb).start();
                new Consumer(mb).start();
                new Consumer(mb).start();
                new Consumer(mb).start();
                new Consumer(mb).start();
        }
}
```

```
p1-68 p1-42 c2-73 c4-65 p2-73 p2-49 c1-68 c4-49 p2-95 p1-65 c1-76 c3-42 c1-3 p1-3 p1-31 p2-76
```

Memento:

Permite salvar e restaurar o estado anterior de um objeto sem revelar os detalhes de sua implementação.




```
class Memento {
        private String state;
        public Memento(String stateToSave) {
                state = stateToSave;
        }
        public String getSavedState() { return state; }
}

class Originator {
        private String state; // simple example
        public void set(String state) { this.state = state; }
        public Memento saveToMemento() { return new Memento(state); }
        public void restoreFromMemento(Memento m) { state = m.getSavedState(); }
        @Override public String toString() { return state; }
}
```

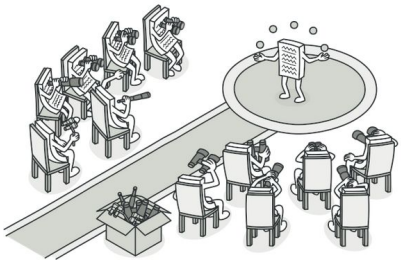
```
class Caretaker {
	private Stack<Memento> savedStates = new Stack<Memento>();
	public void addMemento(Memento m) {
		savedStates.push(m);
	}
	public boolean hasMemento() {
		return !savedStates.isEmpty();
	}
	public Memento getMemento() {
		return savedStates.pop();
	}
}

public class MementoDemo {
	public static void main(String[] args) {
		Caretaker caretaker = new Caretaker();
		Originator originator = new Originator();
		for (int i= 1; i<=5; i++) {
			originator.set("State " + i);
			System.out.println("Originator: state set to "+ originator);
			caretaker.addMemento( originator.saveToMemento() );
			System.out.println("Memento saved");
		}
		while (caretaker.hasMemento()) {
			originator.restoreFromMemento( caretaker.getMemento() );
			System.out.println("Originator: after restore: "+originator);
		}
	}
}
```

```
Originator: state set to State 1
Memento saved
Originator: state set to State 2
Memento saved
Originator: state set to State 3
Memento saved
Originator: state set to State 4
Memento saved
Originator: state set to State 5
Memento saved
Originator: after restore: State 5
Originator: after restore: State 4
Originator: after restore: State 3
Originator: after restore: State 2
Originator: after restore: State 1
```

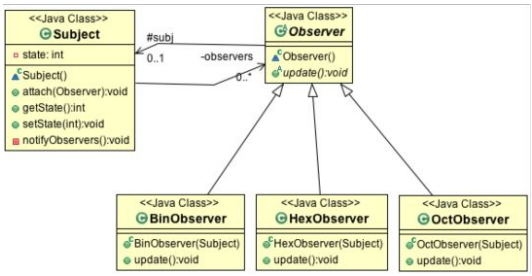
Null Object:

```
Request request = someObj.getRequest(command);
if (request != null) {
	// do something useful
}

public void doSomethingUseful(Request request) {
	if (request == null) {
		logger.warning("null command");
	}
	// do the real stuff
}
```

Observer:



Permite definir um mecanismo de assinatura para notificar vários objetos sobre quaisquer eventos que aconteçam ao objeto que eles estão a observar

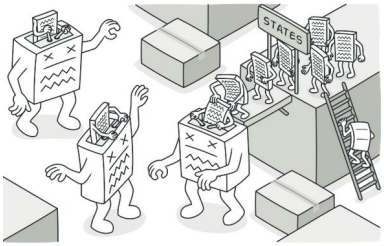



```
class Subject {
	private List<Observer> observers = new
ArrayList<>();
	private int state;
	public void attach(Observer o) {
		observers.add(o);
	}
	public int getState() {
		return state;
	}
	public void setState(int in) {
		state = in;
		notifyObservers();
	}
	private void notifyObservers() {
		for (Observer obs : observers)
			obs.update();

	}
}.

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

class HexObserver extends Observer {
	public HexObserver(Subject s) {
		subj = s;
		subj.attach(this); // Observers register themselves
	}
	public void update() {
		System.out.println("HexObserver saw " + Integer.toHexString(subj.getState()));
	} // Observers "pull" information
}
```

```
class OctObserver extends Observer {
        public OctObserver(Subject s) {
                subj = s;
                subj.attach(this);
        }
        public void update() {
                System.out.println("OctObserver saw " +
        Integer.toOctalString(subj.getState()));
        }
}

class BinObserver extends Observer {
        public BinObserver(Subject s) {
                subj = s;
                subj.attach(this);
        }
        public void update() {
                System.out.println("BinObserver saw " +
        Integer.toBinaryString(subj.getState()));
        }
}

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 OctObserver(sub);
                new BinObserver(sub);
                Scanner scan = new Scanner(System.in);
                while (true) {
                        System.out.print("\nEnter a number: ");
                        sub.setState(scan.nextInt());
                }
        }
}
```

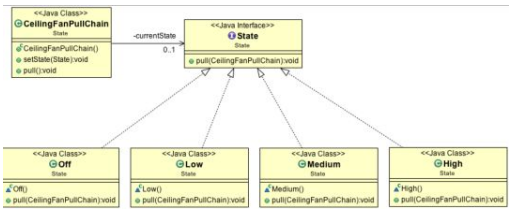
```
Enter a number: 25
HexObserver saw 19
OctObserver saw 31
BinObserver saw 11001

Enter a number: 77
HexObserver saw 4d
OctObserver saw 115
BinObserver saw 1001101
```

State:

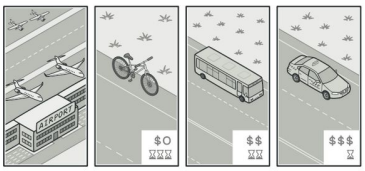


Permite que um objeto altere seu comportamento quando seu estado interno é alterado. Parece que o objeto mudou de classe.

```
class CeilingFanPullChain {
	private State currentState;
	public CeilingFanPullChain() {
		currentState = new Off();
	}
	public void setState(State s) {
		currentState = s;
	}
	public void pull() {
		currentState.pull(this);
	}
}

interface State {
	void pull(CeilingFanPullChain wrapper);
}

class Off implements State {
	public void pull(CeilingFanPullChain wrapper) {
		wrapper.setState(new Low()); System.out.println(" low speed");
	}
}

class Low implements State {
	public void pull(CeilingFanPullChain wrapper) {
		wrapper.setState(new Medium()); System.out.println(" medium speed");
	}
}

class Medium implements State {
	public void pull(CeilingFanPullChain wrapper) {
		wrapper.setState(new High()); System.out.println(" high speed");
	}
}

class High implements State {
	public void pull(CeilingFanPullChain wrapper) {
		wrapper.setState(new Off()); System.out.println(" turning off");
	}
}
```



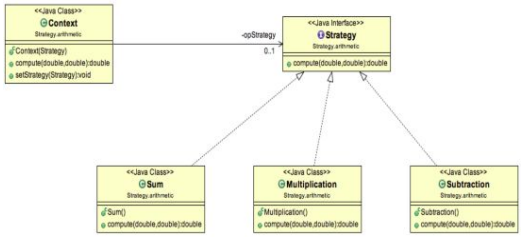
Strategy:



Various strategies for getting to the airport.

Permite definir uma família de algoritmos, colocar cada um deles em uma classe separada e tornar seus objetos intercambiáveis.

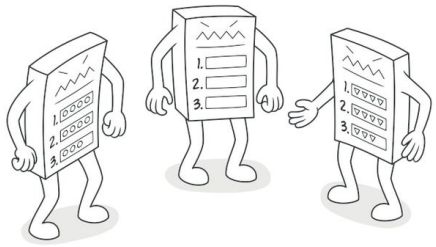
```
public interface Strategy {
	double compute(double elem1, double elem2);
}

public class Sum implements Strategy {
	@Override
	public double compute(double elem1, double
elem2) {
		return elem1 + elem2;
	}
}

public class Multiplication implements Strategy {
	@Override
	public double compute(double elem1, double elem2) {
		return elem1 * elem2;
	}
}

public class Subtraction implements Strategy {
	@Override
	public double compute(double elem1, double elem2) {
		return elem1 - elem2;
	}
}

public class Context {
	private Strategy opStrategy;
	public Context(Strategy operation) {
		this.opStrategy = operation;
	}
	public double compute(double firstNumber, double secondNumber){
		return opStrategy.compute(firstNumber, secondNumber);
	}
```



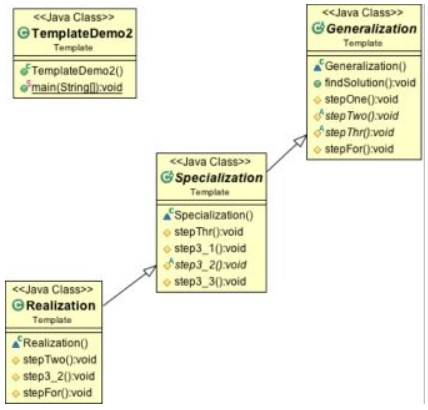
```
        public void setStrategy(Strategy strategy){
                opStrategy = strategy;
        }
}

public class StrategyDemo {
        public static void main(String[] args) {
                double e1 = 5, e2 = 33;
                Context c = new Context(new Sum());
                System.out.println("Result: " + c.compute(e1, e2));
                c.setStrategy(new Subtraction());
                System.out.println("Result: " + c.compute(e1, e2));
                c.setStrategy(new Multiplication());
                System.out.println("Result: " + c.compute(e1, e2));
        }
}
```

```
Result: 38.0
Result: -28.0
Result: 165.0
```

Template Method:



Define o esqueleto de um algoritmo na superclasse, mas permite que as subclasses substituam etapas específicas do algoritmo sem alterar sua estrutura.

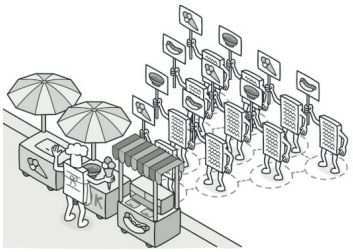



```
abstract class Generalization {
        // 1. Standardize the skeleton of an algorithm in a "template" method
        public void findSolution() {
                stepOne();
                stepTwo();
                stepThr();
                stepFor();
        }
        // 2. Common implementations of individual steps are defined in base class
        protected void stepOne(){
                System.out.println("Generalization.stepOne" );
        }
        // 3. Steps requiring peculiar impls are ";placeholders" in the base class
        abstract protected void stepTwo();
        abstract protected void stepThr();
```

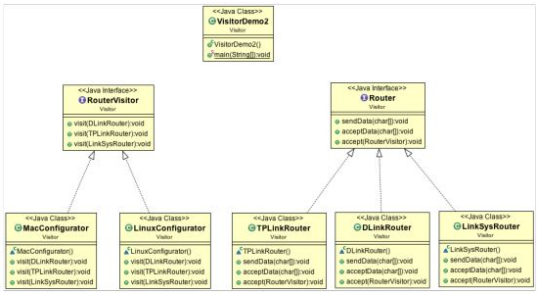
```
	protected void stepFor(){
		System.out.println( "Generalization.stepFor" );
	}
}

abstract class Specialization extends Generalization {
	// 4. Derived classes can override placeholder methods
	// 1. Standardize the skeleton of an algorithm in a "template" method
	protected void stepThr() {
		step3_1();
		step3_2();
		step3_3();
	}
	// 2. Common implementations of individual steps are defined in base class
	protected void step3_1() {
		System.out.println( "Specialization.step3_1" );
	}
	// 3. Steps requiring peculiar impls are "placeholders" in the base class
abstract protected void step3_2();
	protected void step3_3() {
		System.out.println( "Specialization.step3_3" );
	}
}

class Realization extends Specialization {
	// 4. Derived classes can override placeholder methods
	protected void stepTwo(){
		System.out.println("Realization.stepTwo" );
	}
	protected void step3_2(){
		System.out.println("Realization.step3_2" );
	}
	// 5. Derived classes can override implemented methods
	// 6. Derived classes can override and "call back to" base class methods
	protected void stepFor() {
		System.out.println( "Realization.stepFor" );
		super.stepFor();
	}
}

public class TemplateDemo2 {
	public static void main( String[] args ) {
		Generalization algorithm =
		new Realization();
		algorithm.findSolution();
	}
}
```

```
Generalization.stepOne
Realization.stepTwo
Specialization.step3_1
Realization.step3_2
Specialization.step3_3
Realization.stepFor
Generalization.stepFor
```

Visitor:



Permite separar algoritmos dos objetos nos quais eles operam.




```
interface Router {
        public void sendData(char[] data);
        public void acceptData(char[] data);
        public void accept(RouterVisitor v); // for
        visitor
}

class DLinkRouter implements Router {
        @Override public void sendData(char[]
data)
        { /* ... */ }
        @Override public void acceptData(char[] data) { /* ... */ }
        @Override public void accept(RouterVisitor v) { v.visit(this); }
}

class LinkSysRouter implements Router {
        @Override public void sendData(char[] data) { /* ... */ }
        @Override public void acceptData(char[] data) { /* ... */ }
        @Override public void accept(RouterVisitor v) { v.visit(this); }
}

class TPLinkRouter implements Router {
        @Override public void sendData(char[] data) { /* ... */ }
        @Override public void acceptData(char[] data) { /* ... */ }
        @Override public void accept(RouterVisitor v) { v.visit(this); }
}

interface RouterVisitor {
        public void visit(DLinkRouter router);
        public void visit(TPLinkRouter router);
        public void visit(LinkSysRouter router);
}
class MacConfigurator implements RouterVisitor {
        @Override public void visit(DLinkRouter router) {
                // .. configuration here
                System.out.println("DLinkRouter Configuration for Mac complete !!");
```

```
	}
	@Override public void visit(TPLinkRouter router) {
		// .. configuration here
		System.out.println("TPLinkRouter Configuration for Mac complete !!");
	}
	@Override public void visit(LinkSysRouter router) {
		// .. configuration here
		System.out.println("LinkSysRouter Configuration for Mac complete !!");
	}
}

class LinuxConfigurator implements RouterVisitor{
	@Override public void visit(DLinkRouter router) {
		// .. configuration here
		System.out.println("DLinkRouter Configuration for Linux complete !!");
}
	@Override public void visit(TPLinkRouter router) {
		// .. configuration here
		System.out.println("TPLinkRouter Configuration for Linux complete !!");
	}
	@Override public void visit(LinkSysRouter router) {
		// .. configuration here
		System.out.println("LinkSysRouter Configuration for Linux complete !!");
	}
}

public class VisitorDemo2 {
	public static void main(String s[]) {
		Router[] routers = {
			new DLinkRouter(),
			new TPLinkRouter(),
			new LinkSysRouter()
		}
		RouterVisitor[] visitors= {
			new MacConfigurator(),
			new LinuxConfigurator()
		}
		for (Router router: routers)
			for (RouterVisitor rvisitor : visitors)
				router.accept(rvisitor);
	}
}
```

```
DLinkRouter Configuration for Mac complete !!
DLinkRouter Configuration for Linux complete !!
TPLinkRouter Configuration for Mac complete !!
TPLinkRouter Configuration for Linux complete !!
LinkSysRouter Configuration for Mac complete !!
LinkSysRouter Configuration for Linux complete !!
```

Resumo das funcionalidades:

Chain of responsibility- Uma maneira de passar um pedido entre uma cadeia de objetos
Command- Encapsula uma solicitação de comando como um objeto
Interpreter- Uma maneira de incluir elementos de linguagem um programa
Iterator- Acede sequencialmente os elementos de uma coleção
Mediator- Define comunicação simplificada entre classes
Memento- Capturar e restaurar o estado interno de um objeto
Null Object- Projetado para atuar como um valor padrão de um objeto
Observer- Uma maneira de notificar a mudança para um número de classes
State- Alterar o comportamento de um objeto quando o seu estado muda
Strategy- Encapsula um algoritmo dentro de uma classe
Template method- Adia os passos exatos de um algoritmo para uma subclasse
Visiter- Define uma nova operação para uma classe sem alteração