# Temperatura

💡 **Separación de la lógica de conversión** de Celsius a Fahrenheit fuera del main, cumpliendo con el principio de **responsabilidad única** (SRP). Bien hecho.

🧱 Clase Temperatura para encapsular datos y su comportamiento → esto es clave en orientación a objetos.

## Modificaciones

### Inicial

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**double**[] temperaturas = **new** **double**[100];

// Toma de datos

**for** (**int** i = 0; i < temperaturas.length; i++)

temperaturas[i] = *leeSensorCelsius*();

// Cálculo con los datos en Celsius

**double** mediaCelsius = 0;

**for** (**int** i = 0; i < temperaturas.length; i++)

mediaCelsius += temperaturas[i];

mediaCelsius = mediaCelsius / temperaturas.length;

### Final

Clase Main:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Temperatura[] temperaturas = **new** Temperatura[100];

// Toma de datos

**for** (**int** i = 0; i < temperaturas.length; i++)

temperaturas[i] = **new** Temperatura(*leeSensorCelsius*());

// Cálculo con los datos en Celsius

**double** mediaCelsius = 0;

**for** (**int** i = 0; i < temperaturas.length; i++)

mediaCelsius += temperaturas[i].getCelsius();

mediaCelsius = mediaCelsius / temperaturas.length;

Clase Temperatura:

**public** **class** Temperatura {

**final** **double** celsius;

**public** Temperatura(**double** celsius) {

**this**.celsius = celsius;

}

**public** **double** getCelsius() {

**return** celsius;

}

**public** **double** getFaren() {

**return** celsius \* 1.8 + 32;

}

}

# Videoclub

 ✅ **Principio de Abierto/Cerrado (OCP)**: añades nuevas categorías sin tocar el código existente.

 ✅ **Responsabilidad única (SRP)**: la lógica de cálculo de importe y puntos está donde debe estar (las categorías).

 ✅ **Uso del patrón Strategy**: las categorías implementan una estrategia diferente de cálculo.

## Modficiaciones

### Inicial

Clase Main:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Customer raul = **new** Customer("Raúl");

Movie hotFuzz = **new** Movie("Hot Fuzz", Movie.***NEW\_RELEASE***);

Movie toyStory = **new** Movie("Toy Story", Movie.***CHILDRENS***);

Movie zombiesParty = **new** Movie("Zombies Party", Movie.***REGULAR***);

raul.addRental(**new** Rental(hotFuzz, 2));

raul.addRental(**new** Rental(toyStory, 6));

raul.addRental(**new** Rental(zombiesParty, 8));

System.***out***.println(raul.status());

}

}

Clase Rental:

**public** **class** Rental // Alquiler

{

**private** Movie movie;

**private** **int** days;

**public** Rental(Movie movie, **int** days) {

**this**.movie = movie;

**this**.days = days;

}

**public** **int** getDays() {

**return** days;

}

**public** Movie getMovie() {

**return** movie;

}

}

Clase Customer:

**public** **class** Customer {

**private** String name;

**private** List<Rental> rentals = **new** ArrayList<Rental>();

**public** Customer(String name) {

**this**.name = name;

}

**public** **void** addRental(Rental rental) {

rentals.add(rental);

}

**public** String getName() {

**return** name;

}

**public** String status() {

**double** totalPrice = 0;

**int** totalPoints = 0;

String result = "Alquileres de: " + getName() + "\n";

**for** (Rental rental : rentals) {

// 1) Calcula el importe de cada alquiler

**double** price = 0;

**switch** (rental.getMovie().getCategory()) {

**case** Movie.***NEW\_RELEASE***:

price += rental.getDays() \* 3;

**break**;

**case** Movie.***REGULAR***:

price += 2;

**if** (rental.getDays() > 2)

price += (rental.getDays() - 2) \* 1.5;

**break**;

**case** Movie.***CHILDRENS***:

price += 1.5;

**if** (rental.getDays() > 3)

price += (rental.getDays() - 3) \* 1.5;

**break**;

}

// 2) Cada alquiler da 1 punto. Punto extra para novedades alquiladas 2 o más

// días

**int** points = 1;

**if** ((rental.getMovie().getCategory() == Movie.***NEW\_RELEASE***) && rental.getDays() > 1)

points = 2;

// 3) Acumula los totales

totalPrice += price;

totalPoints += points;

// 4) Muestra el importe de esta película alquilada

result += "\t" + rental.getMovie().getTitle() + "\t" + price + " €\n";

}

result += "La deuda es de " + totalPrice + " €\n";

result += "Has obtenido " + totalPoints + " puntos en tu tarjeta";

**return** result;

}

}

Clase Movie:

**public** **class** Movie // Pel�cula

{

**private** String title;

**private** **int** category;

**public** **static** **final** **int** ***CHILDRENS*** = 2;

**public** **static** **final** **int** ***NEW\_RELEASE*** = 1;

**public** **static** **final** **int** ***REGULAR*** = 0;

**public** Movie(String title, **int** category) {

**this**.title = title;

**this**.category = category;

}

**public** **int** getCategory() {

**return** category;

}

**public** **void** setCategory(**int** category) {

**this**.category = category;

}

**public** String getTitle() {

**return** title;

}

}

### Final

Clase Main:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Customer raul = **new** Customer("Raúl");

Movie hotFuzz = **new** Movie("Hot Fuzz", **new** NewReleaseCategory());

Movie toyStory = **new** Movie("Toy Story", **new** ChildrenCategory());

Movie zombiesParty = **new** Movie("Zombies Party", **new** RegularCategory());

Movie documental = **new** Movie("Documental", **new** DocumentalCategory());

raul.addRental(**new** Rental(hotFuzz, 2));

raul.addRental(**new** Rental(toyStory, 6));

raul.addRental(**new** Rental(zombiesParty, 8));

raul.addRental(**new** Rental(documental,2));

System.***out***.println(raul.status());

}

}

Clase Customer:

**public** **class** Customer {

**private** String name;

**private** List<Rental> rentals = **new** ArrayList<Rental>();

**public** Customer(String name) {

**this**.name = name;

}

**public** **void** addRental(Rental rental) {

rentals.add(rental);

}

**public** String getName() {

**return** name;

}

**public** String status() {

**double** totalPrice = 0;

**int** totalPoints = 0;

String result = "Alquileres de: " + getName() + "\n";

**for** (Rental rental : rentals) {

// 1) Calcula el importe de cada alquiler

**double** price = rental.getImporte();

// 2) Cada alquiler da 1 punto. Punto extra para novedades alquiladas 2 o m��s

// d��as

**int** points = rental.getPuntos();

// 3) Acumula los totales

totalPrice += price;

totalPoints += points;

// 4) Muestra el importe de esta pel��cula alquilada

result += "\t" + rental.getMovie().getTitle() + "\t" + price + " €\n";

}

result += "La deuda es de " + totalPrice + " €\n";

result += "Has obtenido " + totalPoints + " puntos en tu tarjeta";

**return** result;

}

}

Clase Movie:

**public** **class** Movie // Pel�cula

{

**private** String title;

**private** Category category;

**public** Movie(String title, Category category) {

**this**.title = title;

**this**.category = category;

}

**public** Category getCategory() {

**return** category;

}

**public** **void** setCategory(Category category) {

**this**.category = category;

}

**public** String getTitle() {

**return** title;

}

**double** getImporte(**int** days) {

**return** category.getImporte(days);

}

**int** getPuntos(**int** days) {

**return** category.getPuntos(days);

}

}

Clase Rental:

**public** **class** Rental // Alquiler

{

**private** Movie movie;

**private** **int** days;

**public** Rental(Movie movie, **int** days) {

**this**.movie = movie;

**this**.days = days;

}

**public** **int** getDays() {

**return** days;

}

**public** Movie getMovie() {

**return** movie;

}

**int** getPuntos() {

**return** movie.getPuntos(days);

}

**double** getImporte() {

**return** movie.getImporte(days);

}

}

Interfac Category:

**public** **interface** Category {

**double** getImporte(**int** daysRental);

**int** getPuntos(**int** daysRental);

}

Clase NewReleaseCategory:

**public** **class** NewReleaseCategory **implements** Category {

@Override

**public** **double** getImporte(**int** daysRental) {

**double** price = 0;

**return** price += daysRental \* 3;

}

@Override

**public** **int** getPuntos(**int** daysRental) {

**return** (daysRental > 1) ? 2 : 1;

}

}

## Mejoras

💡 Aplicar el **patrón Factory** para la creación de películas en el main.

Clase Factoría:

**public** **class** MovieFactory {

**public** Movie create(String titulo, TipoPeliculas tipo) {

Category category = **null**;

**switch** (tipo) {

**case** ***NewRelease***:

category = **new** NewReleaseCategory();

**break**;

**case** ***Children***:

category = **new** ChildrenCategory();

**break**;

**case** ***Documental***:

category = **new** DocumentalCategory();

**break**;

**case** ***Regular***:

category = **new** RegularCategory();

**break**;

**default**:

**break**;

}

**return** **new** Movie(titulo, category);

}

}

Clase Main:

**public** **class** Main {

**public** **enum** TipoPeliculas{

***NewRelease***, ***Children***, ***Documental***, ***Regular***

}

**public** **static** **void** main(String[] args) {

Customer raul = **new** Customer("Raúl");

MovieFactory mf = **new** MovieFactory();

Movie hotFuzz1 = mf.create("Hot Fuzz", TipoPeliculas.***NewRelease***);

Movie toyStory = mf.create("Toy Story", TipoPeliculas.***Children***);

Movie zombiesParty = mf.create("Zombies Party", TipoPeliculas.***Regular***);

Movie documental = mf.create("Documental", TipoPeliculas.***Documental***);

raul.addRental(**new** Rental(hotFuzz1, 2));

raul.addRental(**new** Rental(toyStory, 6));

raul.addRental(**new** Rental(zombiesParty, 8));

raul.addRental(**new** Rental(documental,2));

System.***out***.println(raul.status());

}

# Interprete

 ✅ Cada instrucción ahora tiene su propio método (addAdd, addPush, etc.)

 ✅ Extrajiste la lógica de ip++ a incrementarIp() — mejora la cohesión

 ✅ Tu switch centraliza el flujo — ideal para luego reemplazarlo por un Map<String, Command>

Map<String, Instruccion> mapa = new HashMap<>();

mapa.put("push", new Push());

 ✅ En vez de guardar List<String[]>, guarda List<Instruccion>:

 ✅ Evitar static

## Modificaciones

### Inicial

Clase Main:

**public** **class** Main {

**private** **static** List<String[]> *instrucciones* = **new** ArrayList<String[]>();

**private** **static** **int** *ip* = 0;

**private** **static** **int**[] *memoria* = **new** **int**[1024];

**private** **static** **int**[] *pila* = **new** **int**[32];

**private** **static** **int** *sp* = 0;

**private** **static** Scanner *console* = **new** Scanner(System.***in***);

**public** **static** **void** main(String[] args) **throws** Exception {

BufferedReader fichero = **new** BufferedReader(**new** FileReader("programa.txt"));

String linea;

**while** ((linea = fichero.readLine()) != **null**)

*cargaInstruccion*(linea);

fichero.close();

*ejecutaPrograma*();

}

**private** **static** **void** cargaInstruccion(String linea) {

**if** (linea.trim().length() == 0)

**return**;

String[] palabras = linea.split(" ");

*instrucciones*.add(palabras);

}

**private** **static** **void** push(**int** valor) {

*pila*[*sp*] = valor;

*sp*++;

}

**private** **static** **int** pop() {

*sp*--;

**return** *pila*[*sp*];

}

**private** **static** **void** ejecutaPrograma() {

**while** (*ip* < *instrucciones*.size()) {

String[] instruccion = *instrucciones*.get(*ip*);

**if** (instruccion[0].equals("push")) {

*push*(Integer.*parseInt*(instruccion[1]));

*ip*++;

} **else** **if** (instruccion[0].equals("add")) {

*push*(*pop*() + *pop*());

*ip*++;

} **else** **if** (instruccion[0].equals("sub")) {

**int** b = *pop*();

**int** a = *pop*();

*push*(a - b);

*ip*++;

} **else** **if** (instruccion[0].equals("mul")) {

*push*(*pop*() \* *pop*());

*ip*++;

} **else** **if** (instruccion[0].equals("jmp")) {

*ip* = Integer.*parseInt*(instruccion[1]);

} **else** **if** (instruccion[0].equals("jmpg")) {

**int** b = *pop*();

**int** a = *pop*();

**if** (a > b)

*ip* = Integer.*parseInt*(instruccion[1]);

**else**

*ip*++;

} **else** **if** (instruccion[0].equals("load")) {

**int** direccion = *pop*();

*push*(*memoria*[direccion]);

*ip*++;

} **else** **if** (instruccion[0].equals("store")) {

**int** valor = *pop*();

**int** direccion = *pop*();

*memoria*[direccion] = valor;

*ip*++;

} **else** **if** (instruccion[0].equals("input")) {

System.***out***.println("Escriba un entero:");

*push*(*console*.nextInt());

*ip*++;

} **else** **if** (instruccion[0].equals("output")) {

System.***out***.println(*pop*());

*ip*++;

}

}

}

}

### Final:

Clase Main:

**public** **class** Main {

**public** **static** Contexto *context* = **new** Contexto();

**public** **static** Interprete *inter* = **new** Interprete();

**public** **static** **void** main(String[] args) **throws** Exception {

FileUtil.*cargarFichero*(*inter*);

Programa.*ejecutaPrograma*(*context*, *inter*);

}

}

Clase FileUtil:

**public** **class** FileUtil {

**public** **static** **void** cargarFichero(Interprete inter) **throws** FileNotFoundException, IOException {

BufferedReader fichero = **new** BufferedReader(**new** FileReader("programa.txt"));

String linea;

**while** ((linea = fichero.readLine()) != **null**)

inter.cargaInstruccion(linea);

fichero.close();

}

}

Clase Programa:

**public** **class** Programa {

**public** **static** **void** ejecutaPrograma(Contexto context, Interprete inter) {

**while** (context.getIp() < inter.*instrucciones*.size()) {

inter.*instrucciones*.get(context.getIp()).ejecutar(context);

}

}

}

Clase Contexto:

**public** **class** Contexto {

**private** **int** ip = 0;

**public** **int**[] memoria = **new** **int**[1024];

**private** **int**[] pila = **new** **int**[32];

**private** **int** sp = 0;

**private** Scanner console = **new** Scanner(System.***in***);

**public** **void** push(**int** valor) {

pila[sp] = valor;

sp++;

}

**public** **int** pop() {

sp--;

**return** pila[sp];

}

**public** **int** incrementarIp() {

**return** ip++;

}

**public** **int** getIp() {

**return** ip;

}

**public** **void** setIp(**int** ip) {

**this**.ip = ip;

}

**public** **int**[] getMemoria() {

**return** memoria;

}

**public** **void** setMemoria(**int**[] memoria) {

**this**.memoria = memoria;

}

**public** **int**[] getPila() {

**return** pila;

}

**public** **void** setPila(**int**[] pila) {

**this**.pila = pila;

}

**public** **int** getSp() {

**return** sp;

}

**public** **void** setSp(**int** sp) {

**this**.sp = sp;

}

**public** Scanner getConsole() {

**return** console;

}

**public** **void** setConsole(Scanner console) {

**this**.console = console;

}

Clase Interprete:

**ublic** **class** Interprete {

**public** **void** cargaInstruccion(String linea) {

**if** (linea.trim().length() == 0)

**return**;

String[] palabras = linea.split(" ");

// instrucciones.add(palabras);

String op = palabras[0];

**switch** (op) {

**case** "push":

// addPush(instruccion);

*instrucciones*.add(**new** Push(Integer.*parseInt*(palabras[1])));

**break**;

**case** "add":

// Add.addAdd();

*instrucciones*.add(**new** Add());

**break**;

**case** "sub":

// addSub();

*instrucciones*.add(**new** Sub());

**break**;

**case** "mul":

// addMull();

*instrucciones*.add(**new** Mul());

**break**;

**case** "jmp":

// addJmp(instruccion);

*instrucciones*.add(**new** Jmp(Integer.*parseInt*(palabras[1])));

**break**;

**case** "jmpg":

// Jmp.addJmpg(instruccion);

*instrucciones*.add(**new** Jmpg(Integer.*parseInt*(palabras[1])));

**break**;

**case** "load":

// addLoad();

*instrucciones*.add(**new** Load());

**break**;

**case** "store":

// addStore();

*instrucciones*.add(**new** Store());

**break**;

**case** "input":

// addInput();

*instrucciones*.add(**new** Input());

**break**;

**case** "output":

// addOutput();

*instrucciones*.add(**new** Output());

**break**;

**default**:

**break**;

}

}

// private static List<String[]> instrucciones = new ArrayList<String[]>();

**public** **static** List<Instruccion> *instrucciones* = **new** ArrayList<>();

Clase Instrucción:

**public** **interface** Instruccion {

**void** ejecutar(Contexto ct);

}

Clase Jmp:

**public** **class** Jmp **implements** Instruccion {

**private** **int** valor;

**public** Jmp(**int** valor) {

**this**.valor = valor;

}

@Override

**public** **void** ejecutar(Contexto ct) {

ct.setIp(valor);

}

}

Clase Jmpg:

**public** **class** Jmpg **implements** Instruccion {

**private** **int** valor;

**public** Jmpg(**int** valor) {

**this**.valor = valor;

}

@Override

**public** **void** ejecutar(Contexto ct) {

**int** b = ct.pop();

**int** a = ct.pop();

**if** (a > b)

// new Jmp(valor).ejecutar(ct);

ct.setIp(valor);

**else**

ct.incrementarIp();

}

# Rover

## Modficiaciones

### Inicial

Clase Main:

**public** **static** **void** main(String[] args) {

Rover rover = **new** Rover(**new** Terrain());

RoverController controller = **new** RoverController(rover);

List<String> task = **new** ArrayList<>();

task.add("forward"); // 0:1

task.add("sample");

task.add("backward"); // 0:0

task.add("left");

task.add("backward"); // 1:0

task.add("right");

task.add("forward"); // obstáculo en 1:1

task.add("forward"); // obstáculo en 1:1

task.add("right");

task.add("forward");

task.add("left");

task.add("forward");

task.add("forward");

task.add("forward");

task.add("photo");

task.add("sample");

task.add("left");

task.add("backward");

task.add("photo");

controller.executeTask(task);

rover.transmit();

Clase RoverController:

**public** **class** RoverController {

**public** RoverController(Rover rover) {

**this**.rover = rover;

}

**public** **boolean** executeTask(List<String> task) {

**boolean** success = **true**;

**while** (ip < task.size() && success) {

String instruction = task.get(ip);

**if** (instruction.equals("forward")) {

// se mueve una posición hacia delante salvo que se encuentre con un obstáculo

success = rover.moveForward();

} **else** **if** (instruction.equals("backward")) {

// se mueve una posición hacia atrás salvo que se encuentre con un obstáculo

success = rover.moveBackward();

} **else** **if** (instruction.equals("left")) {

rover.turnLeft();

} **else** **if** (instruction.equals("right")) {

rover.turnRight();

} **else** **if** (instruction.equals("sample")) {

rover.takeSample();

} **else** **if** (instruction.equals("photo")) {

rover.takePhoto();

} **else** {

rover.trace("Unknown instruction!");

}

**if** (success)

ip++;

}

**if** (success)

rover.trace("Task finished");

**else**

rover.trace("Task couldn't have been completed");

**return** success;

}

**private** Rover rover;

**private** **int** ip;

}

Clase Rover (no se toca):

Contiene los métodos, hace la función de Receiver.

### Final:

Clase RoverController:

**public** RoverController(Rover rover) {

**this**.rover = rover;

}

**public** RoverController(Rover rover2, Estrategia parar) {

**this**.rover = rover2;

**this**.estrategia = parar;

}

**public** **boolean** executeTask(List<String> task) {

**boolean** success = **true**;

**while** (ip < task.size() && success) {

String instruction = task.get(ip);

Command cmd = **null**;

**if** (instruction.equals("forward")) {

// se mueve una posición hacia delante salvo que se encuentre con un obstáculo

success = rover.moveForward();

cmd = **new** Forward();

} **else** **if** (instruction.equals("backward")) {

// se mueve una posición hacia atrás salvo que se encuentre con un obstáculo

success = rover.moveBackward();

cmd = **new** BackWard();

} **else** **if** (instruction.equals("left")) {

rover.turnLeft();

cmd = **new** Left();

} **else** **if** (instruction.equals("right")) {

rover.turnRight();

cmd = **new** Right();

} **else** **if** (instruction.equals("sample")) {

rover.takeSample();

} **else** **if** (instruction.equals("photo")) {

rover.takePhoto();

} **else** {

rover.trace("Unknown instruction!");

}

**if** (success) {

ip++;

**if** (cmd != **null**) {

undos.add(0, cmd);

// undos.push(cmd);

}

} **else** {

undos.forEach(c -> c.execute(rover));

success = **true**;

// success = estrategia.algoritmo(this);

}

}

**if** (success)

rover.trace("Task finished");

**else**

rover.trace("Task couldn't have been completed");

**return** success;

}

**public** **void** incrementar() {

ip++;

}

**public** **int** getIp() {

**return** ip;

}

**public** **void** setIp(**int** ip) {

**this**.ip = ip;

}

**public** Rover getRover() {

**return** rover;

}

**public** **void** setRover(Rover rover) {

**this**.rover = rover;

}

**public** Stack<Command> getUndos() {

**return** undos;

}

**public** **void** setUndos(Stack<Command> undos) {

**this**.undos = undos;

}

**private** Rover rover;

**private** **int** ip;

Stack<Command> undos = **new** Stack<>();

Estrategia estrategia;

Clase Main:

**public** **static** **void** main(String[] args) {

Rover rover = **new** Rover(**new** Terrain());

RoverController controller = **new** RoverController(rover);

// RoverController controller = new RoverController(rover, new Saltar());

List<String> task = **new** ArrayList<>();

task.add("forward"); // 0:1

Interfaz Command:

**public** **interface** Command {

**boolean** execute(Rover rover);

}

Command Concretos:

**public** **class** Forward **implements** Command {

@Override

**public** **boolean** execute(Rover rover) {

// **TODO** Auto-generated method stub

**return** rover.moveBackward();

}

}

@Override

**public** **boolean** execute(Rover rover) {

rover.turnRight();

**return** **true**;

}

Interfaz Estrategia:

**public** **interface** Estrategia {

**boolean** algoritmo(RoverController roverController);

}

Estretegias concretas:

**public** **class** Deshacer **implements** Estrategia {

@Override

**public** **boolean** algoritmo(RoverController roverController) {

// **TODO** Auto-generated method stub

roverController.getUndos().forEach(c -> c.execute(roverController.getRover()));

**return** **true**;

}

}

# Editor (State)

## Modficiaciones

### Inicial

### Final

### ✅ COMPONENTES DEL PATRÓN STATE

#### 🔹 Interfaz de Estado: Herramienta

interface Herramienta {

void pinchar(Editor editor, int x, int y);

void mover(Editor editor, int x, int y);

void soltar(Editor editor, int x, int y);

}

#### 🔹 Estados concretos: HerramientaRectangulo, HerramientaCirculo, etc.

class HerramientaRectangulo implements Herramienta {

public void pinchar(Editor e, int x, int y) {

// Guardar punto inicial, crear figura

}

public void mover(Editor e, int x, int y) {

// Modificar dimensiones

}

public void soltar(Editor e, int x, int y) {

// Añadir figura al dibujo

}

}

#### 🔹 Contexto: Editor

class Editor {

private Herramienta herramienta;

public void setHerramienta(Herramienta h) {

this.herramienta = h;

}

public void pinchar(int x, int y) {

herramienta.pinchar(this, x, y);

}

public void mover(int x, int y) {

herramienta.mover(this, x, y);

}

public void soltar(int x, int y) {

herramienta.soltar(this, x, y);

}

public void clickToolButton(String toolName) {

switch (toolName) {

case "rectangulo" -> setHerramienta(new HerramientaRectangulo());

case "circulo" -> setHerramienta(new HerramientaCirculo());

case "triangulo" -> setHerramienta(new HerramientaTriangulo());

case "seleccion" -> setHerramienta(new HerramientaSeleccion());

}

}

}

# Alineacion

## Inicial

## Final

1. **📄 PRÁCTICA 2: Alineación de Texto**
2. **✳️ INTENCIÓN**

Cambiar cómo se dibujan las líneas en el área de texto, según si están alineadas a la izquierda, derecha o justificadas.

1. **✅ COMPONENTES DEL PATRÓN STATE**
2. **🔹 Interfaz de Estado: Alineacion**

interface Alineacion {

String alinear(Line line, int lineWidth);

}

1. **🔹 Estados concretos**

* AlineacionIzquierda
* AlineacionDerecha
* AlineacionJustificada

Cada uno implementa alinear(...) con su propia lógica.

1. **🔹 Contexto: Canvas**

CopiarEditar

class Canvas extends JTextArea {

private Alineacion alineacion;

public void setAlineacion(Alineacion alineacion) {

this.alineacion = alineacion;

}

public void drawDocument() {

for (Line line : document.getLines()) {

String alineada = alineacion.alinear(line, lineWidth);

append(alineada + "\n");

}

}

}

1. **🔹 Cambio de estado: desde los botones (en Window)**

addButton("Left", new AlineacionIzquierda());

addButton("Right", new AlineacionDerecha());

addButton("Justified", new AlineacionJustificada());

Cada botón cambia el estado del Canvas:

button.addActionListener(e -> {

canvas.setAlineacion(alineacion);

canvas.drawDocument();

});

# Glosario

## Strategia

### Esquema

1) Strategy → TarifaAlquiler

- Método: calcularPrecio(int dias)

2) ConcreteStrategy → Normal, Nueva, Infantil

- Método: calcularPrecio(...)

3) Context → Pelicula

- Tiene un atributo TarifaAlquiler

4) Client → Main o Cliente

- Usa la estrategia para calcular precios

### Ejemplos

#### Video

En el main, ves como se crean a partir de tipos numéricos…

Clase Main:

**public** **static** **void** main(String[] args) {

Customer raul = **new** Customer("Raúl");

Movie hotFuzz = **new** Movie("Hot Fuzz", Movie.***NEW\_RELEASE***);

Clase Main tras aplicación del patrón:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Customer raul = **new** Customer("Raúl");

Movie hotFuzz = **new** Movie("Hot Fuzz", **new** NewReleaseCategory());

Clase Movie:

**public** **class** Movie // Pel�cula

{

**private** String title;

**private** **int** category;

**public** **static** **final** **int** ***CHILDRENS*** = 2;

**public** **static** **final** **int** ***NEW\_RELEASE*** = 1;

**public** **static** **final** **int** ***REGULAR*** = 0;

Clase Movie tras aplicar el patrón:

**public** **class** Movie // Pel�cula

{

**private** String title;

**private** Category category;

**public** Movie(String title, Category category) {

**this**.title = title;

**this**.category = category;

}

**public** Category getCategory() {

**return** category;

}

**public** **void** setCategory(Category category) {

**this**.category = category;

}

**double** getImporte(**int** days) {

**return** category.getImporte(days);

}

**int** getPuntos(**int** days) {

**return** category.getPuntos(days);

}

}

Muchos condicionales y métodos muy largos

Clase Customer:

**public** String status() {

**double** totalPrice = 0;

**int** totalPoints = 0;

String result = "Alquileres de: " + getName() + "\n";

**for** (Rental rental : rentals) {

// 1) Calcula el importe de cada alquiler

**double** price = 0;

**switch** (rental.getMovie().getCategory()) {

**case** Movie.***NEW\_RELEASE***:

price += rental.getDays() \* 3;

**break**;

**case** Movie.***REGULAR***:

price += 2;

**if** (rental.getDays() > 2)

price += (rental.getDays() - 2) \* 1.5;

**break**;

**case** Movie.***CHILDRENS***:

price += 1.5;

**if** (rental.getDays() > 3)

price += (rental.getDays() - 3) \* 1.5;

**break**;

}

// 2) Cada alquiler da 1 punto. Punto extra para novedades alquiladas 2 o más

// días

**int** points = 1;

**if** ((rental.getMovie().getCategory() == Movie.***NEW\_RELEASE***) && rental.getDays() > 1)

points = 2;

// 3) Acumula los totales

totalPrice += price;

totalPoints += points;

// 4) Muestra el importe de esta película alquilada

result += "\t" + rental.getMovie().getTitle() + "\t" + price + " €\n";

}

result += "La deuda es de " + totalPrice + " €\n";

result += "Has obtenido " + totalPoints + " puntos en tu tarjeta";

**return** result;

}

}

Clases concretas e interfaz tras aplicar el patrón:

Interfac Category:

**public** **interface** Category {

**double** getImporte(**int** daysRental);

**int** getPuntos(**int** daysRental);

}

Clase NewReleaseCategory:

**public** **class** NewReleaseCategory **implements** Category {

@Override

**public** **double** getImporte(**int** daysRental) {

**double** price = 0;

**return** price += daysRental \* 3;

}

@Override

**public** **int** getPuntos(**int** daysRental) {

**return** (daysRental > 1) ? 2 : 1;

}

}

#### Rover

Te piden 3 FORMAS DIFERENTES de reacción frente a un

obstáculo:

Clase Main:

**public** **static** **void** main(String[] args) {

Rover rover = **new** Rover(**new** Terrain());

RoverController controller = **new** RoverController(rover);

Clase Main tras aplicar el patron:

**public** **static** **void** main(String[] args) {

Rover rover = **new** Rover(**new** Terrain());

RoverController controller = **new** RoverController(rover);

// RoverController controller = new RoverController(rover, new Saltar());

Clase RoverController:

**public** **class** RoverController {

**public** RoverController(Rover rover) {

**this**.rover = rover;

}

{…}

**if** (success)

ip++;

}

Clase RoverController tras aplicar el patrón:

**public** RoverController(Rover rover2, Estrategia parar) {

**this**.rover = rover2;

**this**.estrategia = parar;

}

{..}

} **else** {

undos.forEach(c -> c.execute(rover));

success = **true**;

// success = estrategia.algoritmo(this);

}

Interfaz Estrategia:

**public** **interface** Estrategia {

**boolean** algoritmo(RoverController roverController);

}

Estretegias concretas:

**public** **class** Deshacer **implements** Estrategia {

@Override

**public** **boolean** algoritmo(RoverController roverController) {

// **TODO** Auto-generated method stub

roverController.getUndos().forEach(c -> c.execute(roverController.getRover()));

**return** **true**;

}

}

## Command

### Esquema

1) Command → Instruccion

- Método: ejecutar(Contexto)

2) ConcreteCommand → Add, Sub, Jmp, Input...

- Método: ejecutar(Contexto)

3) Receiver → Contexto

- Métodos: push(), pop(), getIp(), setIp(), getConsole()...

4) Invoker → Programa

- Método: ejecutarPrograma(Contexto, Interprete)

5) Client → Main

- Carga el fichero y ejecuta el programa

6) Command Parser → Interprete

- Método: cargarInstruccion(String)

### Ejemplos

#### Interprete

Clase Main:

**private** **static** **void** ejecutaPrograma() {

**while** (*ip* < *instrucciones*.size()) {

String[] instruccion = *instrucciones*.get(*ip*);

**if** (instruccion[0].equals("push")) {

*push*(Integer.*parseInt*(instruccion[1]));

*ip*++;

} **else** **if** (instruccion[0].equals("add")) {

*push*(*pop*() + *pop*());

*ip*++;

Clase Interprete tras aplicar patrón:

**public** **void** cargaInstruccion(String linea) {

**if** (linea.trim().length() == 0)

**return**;

String[] palabras = linea.split(" ");

// instrucciones.add(palabras);

String op = palabras[0];

**switch** (op) {

**case** "push":

// addPush(instruccion);

*instrucciones*.add(**new** Push(Integer.*parseInt*(palabras[1])));

**break**;

**case** "add":

// Add.addAdd();

*instrucciones*.add(**new** Add());

**break**;

{…}

**public** **static** List<Instruccion> *instrucciones* = **new** ArrayList<>();

Interfaz Command Instrucción:

**public** **interface** Instruccion {

**void** ejecutar(Contexto ct);

}

Command concrecto Clase Jmp:

**public** **class** Jmp **implements** Instruccion {

**private** **int** valor;

**public** Jmp(**int** valor) {

**this**.valor = valor;

}

@Override

**public** **void** ejecutar(Contexto ct) {

ct.setIp(valor);

}

}

Receiver:

Clase Contexto:

**public** **class** Contexto {

**private** **int** ip = 0;

**public** **int**[] memoria = **new** **int**[1024];

**private** **int**[] pila = **new** **int**[32];

**private** **int** sp = 0;

**private** Scanner console = **new** Scanner(System.***in***);

**public** **void** push(**int** valor) {

pila[sp] = valor;

sp++;

}

**public** **int** pop() {

sp--;

**return** pila[sp];

}

#### Rover

Clase RoverController:

**public** **boolean** executeTask(List<String> task) {

**boolean** success = **true**;

**while** (ip < task.size() && success) {

String instruction = task.get(ip);

**if** (instruction.equals("forward")) {

// se mueve una posición hacia delante salvo que se encuentre con un obstáculo

success = rover.moveForward();

} **else** **if** (instruction.equals("backward")) {

// se mueve una posición hacia atrás salvo que se encuentre con un obstáculo

success = rover.moveBackward();

} **else** **if** (instruction.equals("left")) {

rover.turnLeft();

Clase Rover Controller (Invocador y Cliente) tras aplicar patrón:

**public** **boolean** executeTask(List<String> task) {

**boolean** success = **true**;

**while** (ip < task.size() && success) {

String instruction = task.get(ip);

Command cmd = **null**;

**if** (instruction.equals("forward")) {

// se mueve una posición hacia delante salvo que se encuentre con un obstáculo

success = rover.moveForward();

cmd = **new** Forward();

} **else** **if** (instruction.equals("backward")) {

// se mueve una posición hacia atrás salvo que se encuentre con un obstáculo

success = rover.moveBackward();

cmd = **new** BackWard();

} **else** **if** (instruction.equals("left")) {

rover.turnLeft();

cmd = **new** Left();

} **else** **if** (instruction.equals("right")) {

rover.turnRight();

cmd = **new** Right();

} **else** **if** (instruction.equals("sample")) {

rover.takeSample();

} **else** **if** (instruction.equals("photo")) {

rover.takePhoto();

} **else** {

rover.trace("Unknown instruction!");

}

**if** (success) {

ip++;

**if** (cmd != **null**) {

undos.add(0, cmd);

// undos.push(cmd);

}

} **else** {

undos.forEach(c -> c.execute(rover));

success = **true**;

// success = estrategia.algoritmo(this);

}

}

Interfaz Command:

**public** **interface** Command {

**boolean** execute(Rover rover);

}

Command Concretos:

**public** **class** Forward **implements** Command {

@Override

**public** **boolean** execute(Rover rover) {

// **TODO** Auto-generated method stub

**return** rover.moveBackward();

}

}

Existe una clase Receiver (Rover) que contiene los métodos. No se modificaba, estaba hecha desde el inicio.

## State

#### Esquema

// 1. INTERFAZ DEL ESTADO

interface State {

void handle(Context context);

}

// 2. ESTADOS CONCRETOS

class StateA implements State {

public void handle(Context context) {

// comportamiento específico de A

context.setState(new StateB()); // transición (opcional)

}

}

class StateB implements State {

public void handle(Context context) {

// comportamiento específico de B

}

}

// 3. CONTEXTO (el que cambia de estado)

class Context {

private State state;

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

public void request() { state.handle(this); }

}

#### ¿Cuándo se usa?

**🛠️ ¿Cómo reutilizarlo?**

Cuando te enfrentes a un problema donde:

* Un objeto **tiene múltiples modos de operación**.
* El comportamiento **varía dinámicamente** según el modo.
* Quieres evitar if/switch repetitivos.

Piensa en:

🔁 **"¿Puedo encapsular estos comportamientos en clases distintas con una interfaz común y delegar la ejecución a la clase actual?"**

Si la respuesta es sí → **usa el patrón State**.

#### Uso practico:

 Los estados **no deben tener if/switch** entre ellos. Cada clase representa un estado claramente.

 El contexto **nunca debe conocer los detalles internos** de cada estado.

 El cambio de estado puede hacerse desde:

* El exterior (setEstado(new EstadoX()))
* O desde el propio estado (context.setEstado(...))

#### Plantilla genérica

##### 1. **Interfaz común del estado**

public interface Estado {

void accion(Contexto contexto);

}

Puedes modificar el nombre del método (accion) y agregar más si es necesario: pinchar, mover, soltar, alinear, etc.

##### 2. **Estados concretos**

public class EstadoA implements Estado {

@Override

public void accion(Contexto contexto) {

System.out.println("Estado A: ejecutando acción...");

// Opcional: cambiar al siguiente estado

// contexto.setEstado(new EstadoB());

}

}

public class EstadoB implements Estado {

@Override

public void accion(Contexto contexto) {

System.out.println("Estado B: otra acción distinta");

}

}

##### 3. **Contexto que mantiene el estado actual**

public class Contexto {

private Estado estado;

public void setEstado(Estado nuevoEstado) {

this.estado = nuevoEstado;

}

public void ejecutarAccion() {

if (estado != null)

estado.accion(this);

else

System.out.println("No hay estado definido");

}

}

##### 4. **Ejemplo de uso en** main

public class Main {

public static void main(String[] args) {

Contexto contexto = new Contexto();

// Comportamiento con EstadoA

contexto.setEstado(new EstadoA());

contexto.ejecutarAccion();

// Cambiar a EstadoB

contexto.setEstado(new EstadoB());

contexto.ejecutarAccion();

}

}

#### Adaptacion a casos:

| **Problema real** | **Interfaz común** | **Estados concretos** | **Contexto** | **Método delegado** |
| --- | --- | --- | --- | --- |
| Herramientas del editor | Herramienta | HerramientaRectangulo, HerramientaCirculo | Editor | pinchar, mover, soltar |
| Alineación del texto | Alineacion | Izquierda, Derecha, Justificada | Canvas | alinear(Line, ancho) |
| Modo de juego | ModoJuego | ModoAtaque, ModoDefensa | Personaje | ejecutarAccion() |
| Botones en GUI | ComportamientoBoton | Guardar, Cancelar, Editar | Boton | accion() |

# Señales Claras

| **SITUACIÓN EN EL CÓDIGO** | **PATRÓN A USAR** | **QUÉ HACER** |
| --- | --- | --- |
| Muchas condiciones if-else por tipo o comando | Strategy / Command | Extrae clases e interfaces |
| Un enum con muchos usos condicionales | Strategy | Usa polimorfismo en lugar del enum |
| Secuencia de acciones interpretadas | Command | Representa acciones como objetos |
| Repetición de lógica en distintos métodos | Template / Strategy | Unifica lógica y parametriza comportamientos |
| Lógica del main extensa | Command / Strategy | Refactoriza y delega |

| **Aspecto** | **Strategy** | **State** |
| --- | --- | --- |
| Propósito principal | Elegir algoritmo | Representar comportamiento por estado |
| ¿Cambia dinámicamente? | Sí, pero según la estrategia | Sí, pero para **simular un estado** |
| A quién se delega | Al algoritmo | Al estado (que puede cambiar internamente) |
| ¿Cambia comportamiento completo del objeto? | No necesariamente | Sí, completamente |

👉 En resumen:

* **Strategy** es como elegir el “cómo”.
* **State** es como cambiar el “quién soy ahora”.

Aquí estás diciendo:  
🗣️ *"El editor está en modo 'rectángulo', ahora en modo 'selección'"*, eso **es un cambio de estado** → por eso **State** es el patrón más adecuado.

| **Elemento** | **Patrón General** | **Ejemplo 1 (Herramientas)** | **Ejemplo 2 (Alineación)** |
| --- | --- | --- | --- |
| Interfaz | State | Herramienta | Alineacion |
| Estados | StateA, StateB, etc. | HerramientaRectangulo... | AlineacionIzquierda... |
| Contexto | Context | Editor | Canvas |
| Cambio de estado | context.setState(...) | editor.setHerramienta(...) | canvas.setAlineacion(...) |
| Motivación | Cambiar comportamiento dinámico | Dibujo según herramienta | Alineación según botón |