

SISTEMA DE VUELOS

Moises Ariel Urrutia Membreño - JV20

Rodrigo Salomón Cristales Escobar – JV20

OBJETIVO

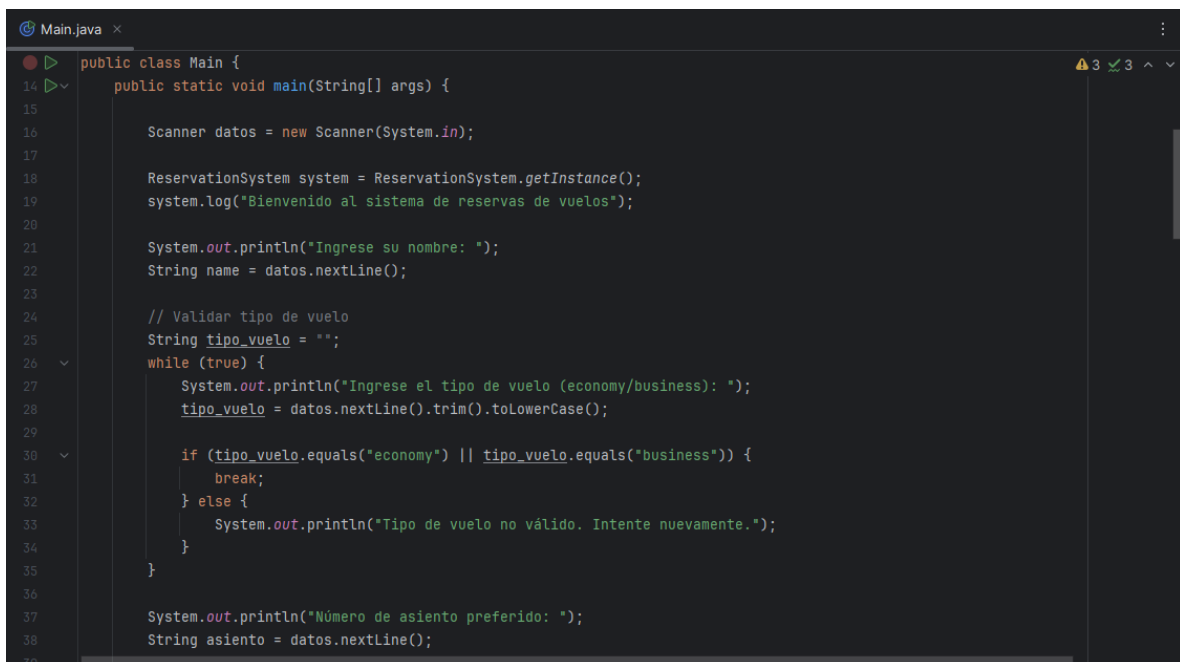
Explicar las razones detrás de las decisiones de diseño y refactorización del sistema de reservas, y cómo estas decisiones están alineadas con los principios SOLID para lograr un software robusto, mantenible y escalable.

SISTEMA DE VUELOS.JAVA

Función General: Permitir a los usuarios realizar, gestionar y cancelar reservas de vuelos mediante una interfaz interactiva por consola, aplicando principios de diseño orientado a objetos y patrones como Singleton, Factory, Builder, Strategy, Command y Observer, para asegurar modularidad, flexibilidad y facilidad de mantenimiento del sistema.

1. Single Responsibility Principle (SRP) – Principio de Responsabilidad Única.

- **Main.java:** solo orquesta la ejecución del sistema.



```
14 public class Main {
15     public static void main(String[] args) {
16
17         Scanner datos = new Scanner(System.in);
18
19         ReservationSystem system = ReservationSystem.getInstance();
20         system.log("Bienvenido al sistema de reservas de vuelos");
21
22         System.out.println("Ingrese su nombre: ");
23         String name = datos.nextLine();
24
25         // Validar tipo de vuelo
26         String tipo_vuelo = "";
27         while (true) {
28             System.out.println("Ingrese el tipo de vuelo (economy/business): ");
29             tipo_vuelo = datos.nextLine().trim().toLowerCase();
30
31             if (tipo_vuelo.equals("economy") || tipo_vuelo.equals("business")) {
32                 break;
33             } else {
34                 System.out.println("Tipo de vuelo no válido. Intente nuevamente.");
35             }
36         }
37
38         System.out.println("Número de asiento preferido: ");
39         String asiento = datos.nextLine();
40     }
```

```

40     Flight flight;
41     double precio = 0;
42
43     try {
44         flight = FlightFactory.vuelos(tipo_vuelo);
45         system.log("Vuelo creado: " + flight.getDescripcion());
46
47         precio = tipo_vuelo.equals("economy") ? 120.0 : 140.0;
48     } catch (IllegalArgumentException e) {
49         system.log("Error: " + e.getMessage());
50         datos.close();
51         return;
52     }
53
54     // Builder
55     ConcreteFlightReservationBuilder builder = new ConcreteFlightReservationBuilder();
56     ReservationDirector director = new ReservationDirector(builder);
57     FlightReservation reservation = director.crearReservacion(name, flight.getDescripcion(), asiento);
58
59     // Observer
60     ReservationNotifier notifier = new ReservationNotifier();
61     UserNotificacion notificacion = new UserNotificacion(name);
62     notifier.addObserver(notificacion);
63     notifier.notifyObservers("Tu reserva fue registrada con éxito para un " + flight.getDescripcion() + ", asiento " + asiento .

```

```

65     // Command - Confirmar reserva
66     ReservationInvoker invoker = new ReservationInvoker();
67     Command confirmarReserva = new MakeReservationCommand(reservation);
68     invoker.executeCommand();
69
70     // Precio
71     System.out.println("Total a pagar: $" + precio);
72
73     // Strategy - Pago
74     PaymentStrategy metodoDePago = null;
75     int intentos = 0;
76     while (metodoDePago == null && intentos < 3) {
77         System.out.println("Seleccione un método de pago (paypal/credit):");
78         String metodoPago = datos.nextLine().trim().toLowerCase();
79
80         switch (metodoPago) {
81             case "credit":
82                 System.out.println("Ingrese número de tarjeta:");
83                 String numeroTarjeta = datos.nextLine();
84                 System.out.println("Nombre del titular:");
85                 String titular = datos.nextLine();
86                 metodoDePago = new CreditCardPayment(numeroTarjeta, titular);
87                 break;
88             case "paypal":
89                 System.out.println("Ingrese email de cuenta PayPal:");

```

```

88             case "paypal":
89                 System.out.println("Ingrese email de cuenta PayPal:");
90                 String email = datos.nextLine();
91                 metodoDePago = new PayPalPayment(email);
92                 break;
93             default:
94                 System.out.println("Método de pago no válido. Intente nuevamente.");
95                 intentos++;
96         }
97     }
98
99     if (metodoDePago == null) {
100         System.out.println("Demasiados intentos fallidos. Se cancela el pago.");
101         datos.close();
102         return;
103     }
104
105     metodoDePago.pay(precio);
106
107     // Command - Cancelar reserva
108     System.out.println("¿Deseas cancelar tu reserva? (si/no)");
109     String cancelar = datos.nextLine().trim().toLowerCase();
110
111     if (cancelar.equals("si")) {
112         Command cancelCommand = new CancelReservationCommand(name);

```

```

111         if (cancelar.equals("si")) {
112             Command cancelCommand = new CancelReservationCommand(name);
113             invoker.setCommand(cancelCommand);
114             invoker.executeCommand();
115
116             // Notificar cancelación
117             notifier.notifyObservers("Tu reserva ha sido cancelada.");
118         }
119
120         datos.close();
121     }
122 }
123

```

- **FlightFactory**: se encarga exclusivamente de la creación de vuelos.

```

© FlightFactory.java x
1 package Factory;
2
3 // Crear objetos de _vuelos sin especificar clase exacta
4
5 public class FlightFactory { 2 usages
6
7     @ public static Flight vuelos(String type){ 1 usage
8         switch(type.toLowerCase()){
9             case "economy":
10                 return new EconomyFlight();
11             case "business":
12                 return new BusinessFlight();
13             default:
14                 throw new IllegalArgumentException("Tipo de vuelo no reconocida, escoja una opcion valida");
15         }
16     }
17 }
18 }
19

```

- **ConcreteFlightReservationBuilder**: construye una reserva paso a paso.

```

© ConcreteFlightReservationBuilder.java x
1 package builder;
2
3 public class ConcreteFlightReservationBuilder implements FlightReservationBuilder { 3 usages
4
5     private FlightReservation reservation; 5 usages
6
7     > public ConcreteFlightReservationBuilder(){ reservation = new FlightReservation(); }
10
11     @Override 1 usage
12     > public void setNombrePasajero(String name){ reservation.setNombrePasajero(name); }
15
16     @Override 1 usage
17     > public void setTipoVuelo(String type){ reservation.setTipoVuelo(type); }
20
21     @Override 1 usage
22     > public void setAsiento(String seat){ reservation.setAsiento(seat); }
25
26     @Override 1 usage
27     > public FlightReservation build(){ return reservation; }
30
31 }

```

- **ReservationNotifier y UserNotificacion:** manejan la notificación de eventos (Observer).

```
© ReservationNotifier.java x
1 package observer;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class ReservationNotifier implements ReservationSubject { 2 usages
7     private List<Observer> observers = new ArrayList<>(); 3 usages
8
9     public void addObserver(Observer observer) { observers.add(observer); }
10
11
12     public void removeObserver(Observer observer) { observers.remove(observer); }
13
14
15
16
17     public void notifyObservers(String message) { 2 usages
18         for(Observer o : observers){
19             o.update(message);
20         }
21     }
22
23 }
24
```

```
© ReservationNotifier.java x © UserNotificacion.java x
1 package observer;
2
3 public class UserNotificacion implements Observer { 2 usages
4     private String userName; 2 usages
5
6     public UserNotificacion(String userName) { this.userName = userName; }
7
8
9
10     public void update(String mensaje) { System.out.println("Notificacion para " + userName + ": " +mensaje
11
12 }
13
14
```

- **ReservationSystem:** centraliza el registro de eventos (Singleton, Logging).

```
© ReservationSystem.java x
1 package Singleton;
2
3 //Controlar que exista una sola instancia global del sistema de reservas
4 public class ReservationSystem { 6 usages
5
6     private static ReservationSystem instance; 3 usages
7
8     private ReservationSystem() { System.out.println("Sistema de reservas para vuelos"); }
9
10
11
12     public static ReservationSystem getInstance() { 1 usage
13         if(instance == null){
14             instance = new ReservationSystem();
15         }
16         return instance;
17     }
18
19     public void log(String message) { System.out.println(message); }
20
21 }
22
23
```

- **MakeReservationCommand** y **CancelReservationCommand**: encapsulan acciones de reserva.

```
1 package command;
2
3 import builder.*;
4
5 public class MakeReservationCommand implements Command { 1 usage
6
7     private FlightReservation reservation; 2 usages
8
9     public MakeReservationCommand(FlightReservation reservation) { this.reservation = reservation; }
10
11     public void execute() { 1 usage
12         System.out.println("Reserva realizada");
13         System.out.println(reservation.getDetalles());
14     }
15 }
16
17
18
```

```
1 package command;
2
3
4 public class CancelReservationCommand implements Command { 1 usage
5
6     private String userName; 2 usages
7
8     public CancelReservationCommand(String userName) { this.userName = userName; }
9
10    public void execute() { System.out.println("La reserva de " + userName + " Ha sido cancelada con exito"); }
11
12
13
14
15
16
17
18
```

2. Open/Closed Principle (OCP) – Principio Abierto/Cerrado

Descripción: El software debe estar abierto a la extensión, pero cerrado a la modificación.

```
1 package Factory;
2
3 // Crear objetos de __vuelos sin especificar clase exacta
4
5 public class FlightFactory { 2 usages
6
7     public static Flight vuelos(String type){ 1 usage
8         switch(type.toLowerCase()){
9             case "economy":
10                 return new EconomyFlight();
11             case "business":
12                 return new BusinessFlight();
13             default:
14                 throw new IllegalArgumentException("Tipo de vuelo no reconocida, escoja una opcion valida");
15         }
16     }
17 }
18
19
```

```

24      // Validar tipo de vuelo
25      String tipo_vuelo = "";
26      while (true) {
27          System.out.println("Ingrese el tipo de vuelo (economy/business): ");
28          tipo_vuelo = datos.nextLine().trim().toLowerCase();
29
30          if (tipo_vuelo.equals("economy") || tipo_vuelo.equals("business")) {
31              break;
32          } else {
33              System.out.println("Tipo de vuelo no válido. Intente nuevamente.");
34          }
35      }

```

PaymentStrategy metodo = new PayPalPayment("correo@example.com");

```

88      case "paypal":
89          System.out.println("Ingrese email de cuenta PayPal:");
90          String email = datos.nextLine();
91          metodoDePago = new PayPalPayment(email);
92          break;

```

Impacto: Se pueden extender funcionalidades sin modificar código existente.

3. Liskov Substitution Principle (LSP) – Principio de Sustitución de Liskov

Descripción: Una clase hija debe poder reemplazar a su clase padre sin alterar el comportamiento esperado.

Aplicación en el sistema:

- Flight puede ser sustituido por BusinessFlight o EconomyFlight sin afectar el sistema.

```

7 @ public static Flight vuelos(String type){ 1 usage
8     switch(type.toLowerCase()){
9         case "economy":
10             return new EconomyFlight();
11         case "business":
12             return new BusinessFlight();
13         default:
14             throw new IllegalArgumentException("Tipo de vuelo no reconocida, escoja una opcion valida");
15     }
16 }
17

```

- **PaymentStrategy** puede ser sustituido por cualquier clase que implemente `pay()`.

```
© PayPalPayment.java x
1 package strategy;
2 //cambiar dinamicamente la forma de pagar
3
4 public class PayPalPayment implements PaymentStrategy { 1 usage
5
```

Impacto: Permite cambiar dinámicamente comportamientos sin romper la lógica existente.

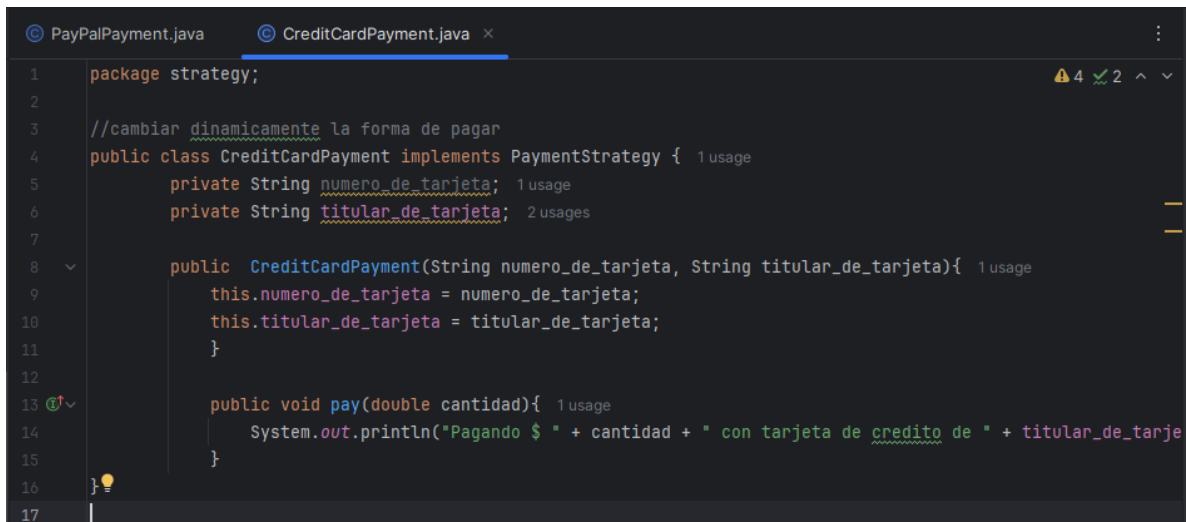
4. Interface Segregation Principle (ISP) – Principio de Segregación de Interfaces

Descripción: Las interfaces deben ser específicas y no forzar a implementar métodos que no se necesitan.

Aplicación en el sistema:

- **PaymentStrategy** define solo el método necesario `pay(double amount)`.

```
© PayPalPayment.java x © CreditCardPayment.java
1 package strategy;
2 //cambiar dinamicamente la forma de pagar
3
4 public class PayPalPayment implements PaymentStrategy { 1 usage
5
6     private String email; 2 usages
7
8     public PayPalPayment(String email) { this.email = email; }
9
10
11
12     public void pay(double cantidad){ 1 usage
13         System.out.println("Pagando $ " + cantidad + " Usando PayPal con la cuenta : " + email);
14     }
15
16 }
17
```

```

1 package strategy;
2
3 //cambiar dinamicamente la forma de pagar
4 public class CreditCardPayment implements PaymentStrategy { 1 usage
5     private String numero_de_tarjeta; 1 usage
6     private String titular_de_tarjeta; 2 usages
7
8     public CreditCardPayment(String numero_de_tarjeta, String titular_de_tarjeta){ 1 usage
9         this.numero_de_tarjeta = numero_de_tarjeta;
10        this.titular_de_tarjeta = titular_de_tarjeta;
11    }
12
13    public void pay(double cantidad){ 1 usage
14        System.out.println("Pagando $ " + cantidad + " con tarjeta de credito de " + titular_de_tarje
15    }
16 }
17

```

- Observer define únicamente **update(String mensaje)**.



```

1 package observer;
2
3 public class UserNotificacion implements Observer { 2 usages
4     private String userName; 2 usages
5
6     public UserNotificacion(String userName){ this.userName = userName; }
7
8
9     public void update(String menssage){ System.out.println("Notificacion para " + userName + ": " +menssag
10 }
11
12
13
14

```

Impacto: Evita que las clases implementen métodos innecesarios, promoviendo interfaces ligeras y específicas.

5. Dependency Inversion Principle (DIP) – Principio de Inversión de Dependencias

Descripción: Los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones.

Aplicación en el sistema:

- Main.java trabaja con **Command**, **PaymentStrategy**, **Flight**, no con clases concretas.

```
65 // Command - Confirmar reserva
66 ReservationInvoker invoker = new ReservationInvoker();
67 Command confirmarReserva = new MakeReservationCommand(reservation);
68 invoker.executeCommand();
69
```

```
73 // Strategy - Pago
74 PaymentStrategy metodoDePago = null;
75 int intentos = 0;
76 while (metodoDePago == null && intentos < 3) {
77     System.out.println("Seleccione un método de pago (paypal/credit):");
78     String metodoPago = datos.nextLine().trim().toLowerCase();
```

```
26 while (true) {
27     System.out.println("Ingrese el tipo de vuelo (economy/business): ");
28     tipo_vuelo = datos.nextLine().trim().toLowerCase();
29
30     if (tipo_vuelo.equals("economy") || tipo_vuelo.equals("business")) {
31         break;
32     } else {
33         System.out.println("Tipo de vuelo no válido. Intente nuevamente.");
34     }
35 }
36
37 System.out.println("Número de asiento preferido: ");
38 String asiento = datos.nextLine();
39
40 Flight flight;
```

- **ReservationInvoker** usa **Command** sin saber si se trata de reservar o cancelar.

```
65 // Command - Confirmar reserva
66 ReservationInvoker invoker = new ReservationInvoker();
67 Command confirmarReserva = new MakeReservationCommand(reservation);
68 invoker.executeCommand();
69
```

<https://github.com/Rodrigo-Cristales/Sistema-vuelos.git> - Repositorio Github