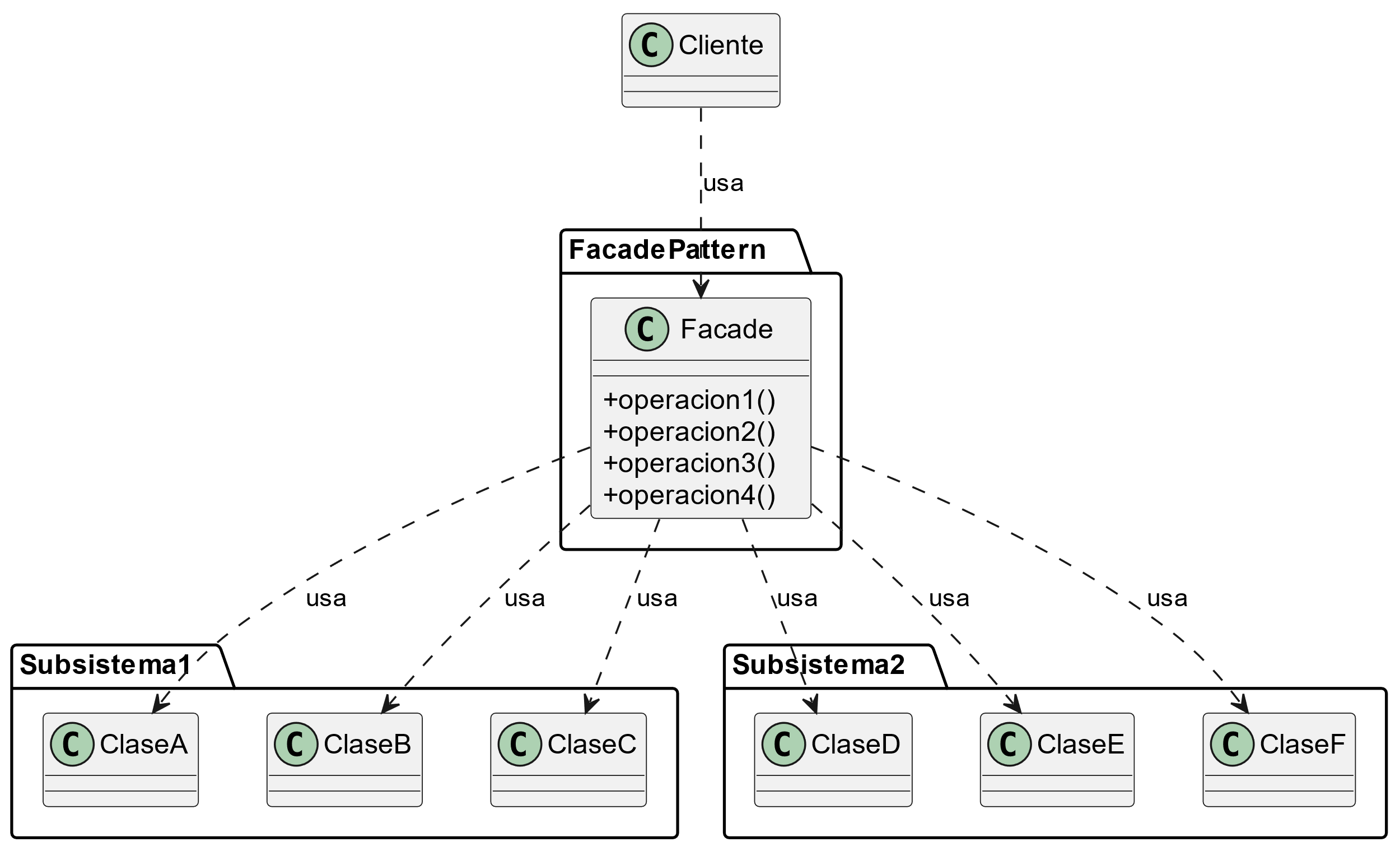
Semana 11: Patrones Estructurales: Facade, Proxy y Flyweight

# Patrón Facade

## ¿Qué es el Patrón Facade?

El patrón Facade proporciona una interfaz unificada y simplificada para un conjunto de interfaces en un subsistema complejo. Facilita la interacción del cliente escondiendo la complejidad interna del sistema.



### 🛠 ¿Cuándo usarlo?

Cuando tienes un subsistema complejo y quieres ofrecer una interfaz más amigable.

Para desacoplar el cliente del subsistema subyacente.

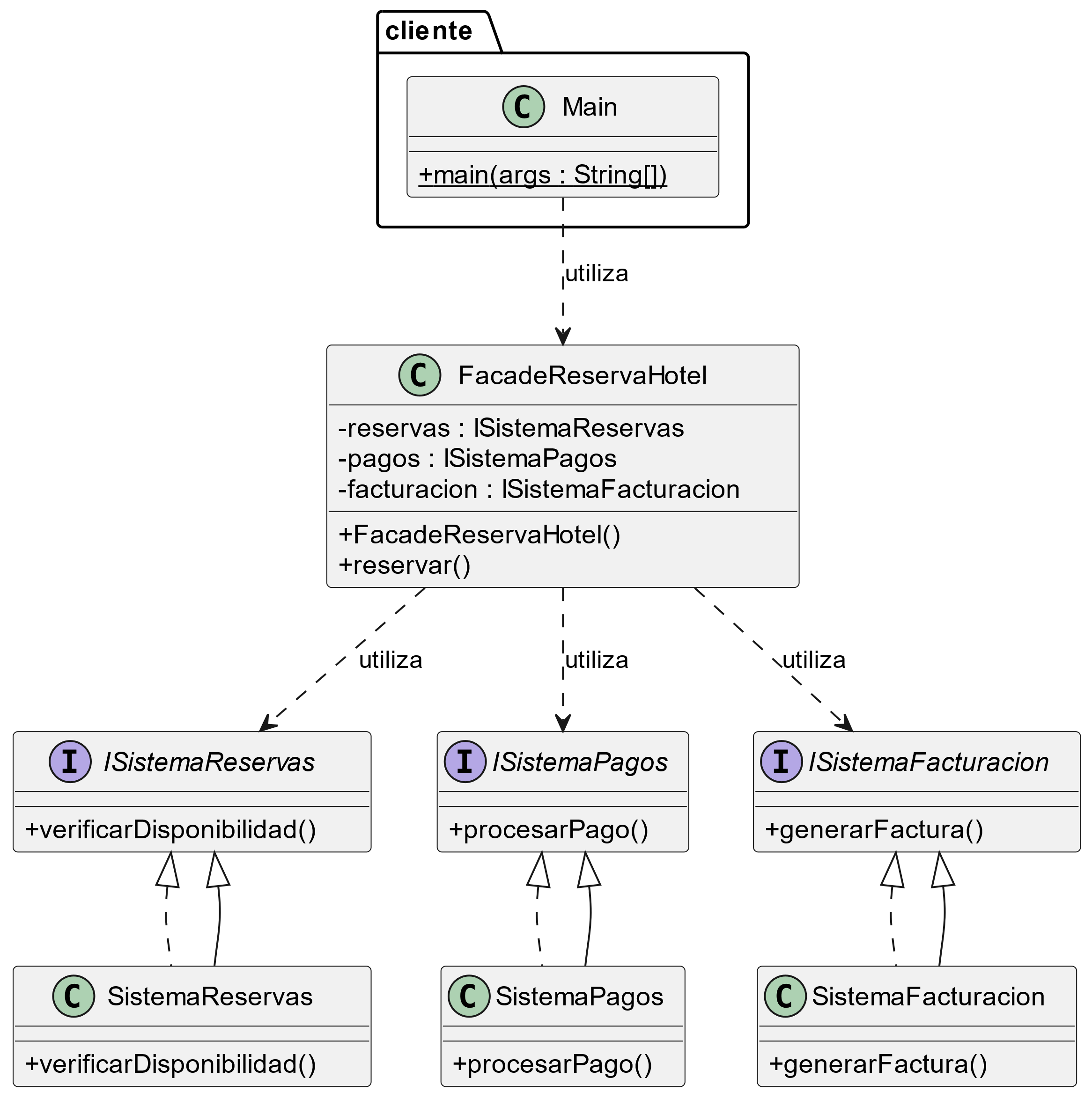
### ⚠️ Antipatrones comunes

Crear un "Objeto Dios" (Facade demasiado grande).

Usarlo para esconder mal diseño interno en vez de mejorar la arquitectura.

### 🧑‍💼 Historia de Usuario – Plataforma de Reservas de Hoteles

Como usuario, quiero reservar un hotel en línea sin tener que interactuar con reservas, pagos y facturación manualmente.



## 🗂 Estructura del Proyecto

- interfaces/

- ISistemaReservas.java

- ISistemaPagos.java

- ISistemaFacturacion.java

- servicios/

- SistemaReservas.java

- SistemaPagos.java

- SistemaFacturacion.java

- fachada/

- FacadeReservaHotel.java

- cliente/

- Main.java

## 💻 Código Java Expandido

interfaces/ISistemaReservas.java

public interface ISistemaReservas {

void verificarDisponibilidad();

}

interfaces/ISistemaPagos.java

public interface ISistemaPagos {

void procesarPago();

}

interfaces/ISistemaFacturacion.java

public interface ISistemaFacturacion {

void generarFactura();

}

servicios/SistemaReservas.java

public class SistemaReservas implements ISistemaReservas {

public void verificarDisponibilidad() {

System.out.println("Hotel disponible.");

}

}

servicios/SistemaPagos.java

public class SistemaPagos implements ISistemaPagos {

public void procesarPago() {

System.out.println("Pago procesado.");

}

}

servicios/SistemaFacturacion.java

public class SistemaFacturacion implements ISistemaFacturacion {

public void generarFactura() {

System.out.println("Factura generada.");

}

}

fachada/FacadeReservaHotel.java

public class FacadeReservaHotel {

private ISistemaReservas reservas;

private ISistemaPagos pagos;

private ISistemaFacturacion facturacion;

public FacadeReservaHotel() {

this.reservas = new SistemaReservas();

this.pagos = new SistemaPagos();

this.facturacion = new SistemaFacturacion();

}

public void reservar() {

reservas.verificarDisponibilidad();

pagos.procesarPago();

facturacion.generarFactura();

System.out.println("Reserva completada.");

}

}

cliente/Main.java

public class Main {

public static void main(String[] args) {

FacadeReservaHotel facade = new FacadeReservaHotel();

facade.reservar();

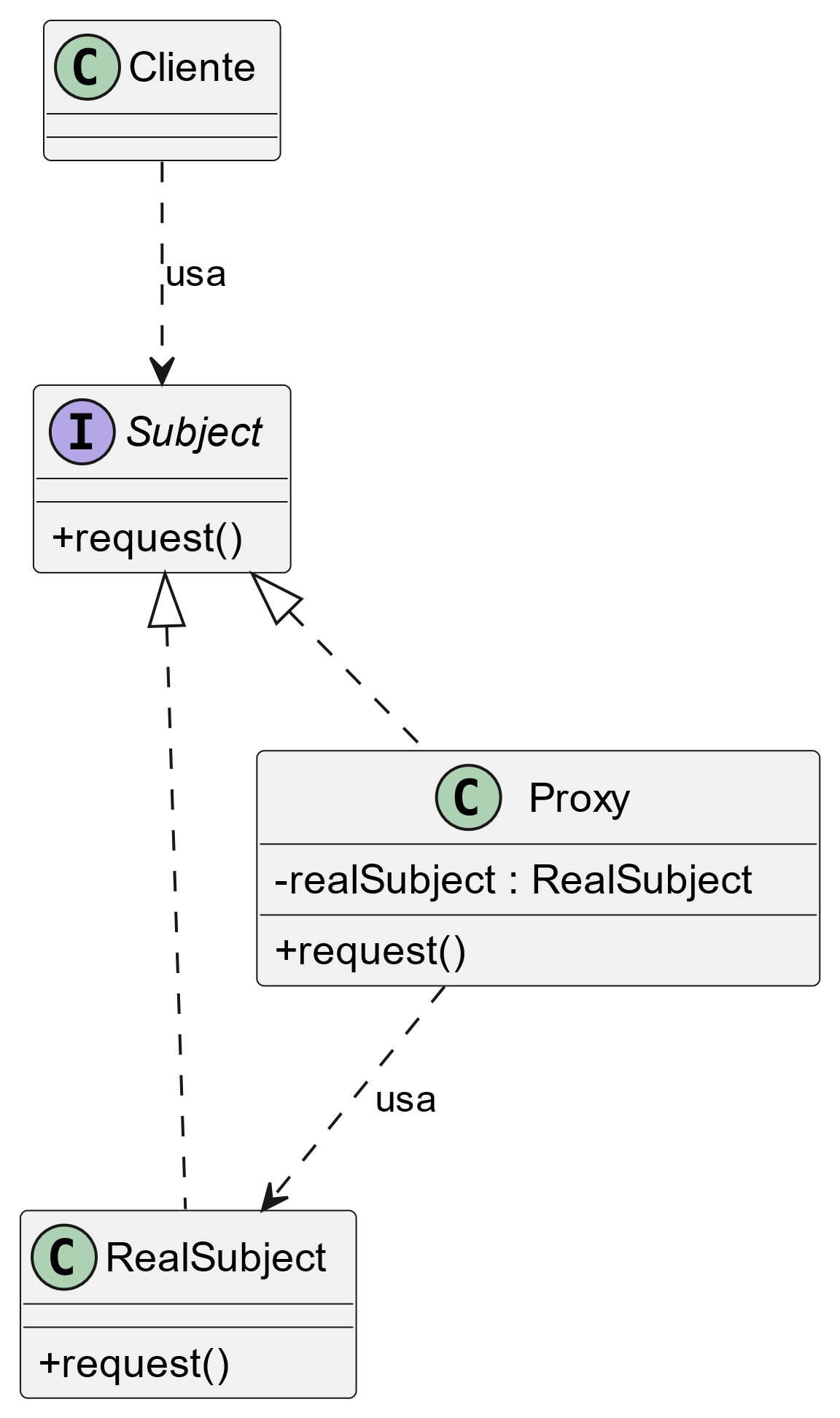
}

}

# Patrón Proxy

## ¿Qué es el Patrón Proxy?

El patrón Proxy proporciona un sustituto o representante de otro objeto para controlar el acceso a él. Puede ser utilizado para optimizar recursos, seguridad o agregar lógica adicional.



### 🛠 ¿Cuándo usarlo?

Cuando necesitas control de acceso a objetos costosos.

Cuando deseas agregar seguridad o validación previa a las operaciones.

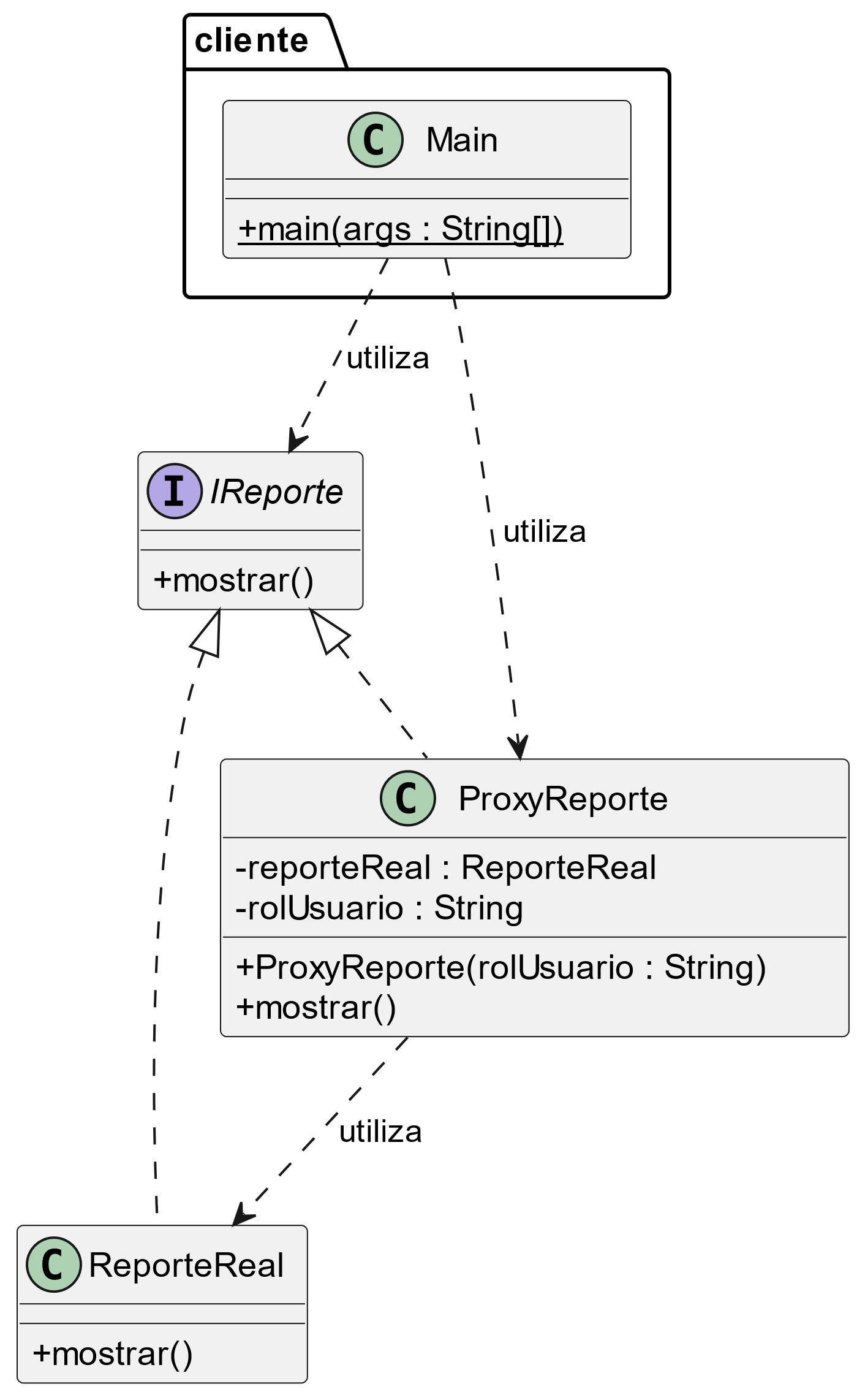
### ⚠️ Antipatrones comunes

Introducir latencia innecesaria si se abusa del proxy.

No considerar concurrencia en proxies compartidos.

### 🧑‍💼 Historia de Usuario – Acceso Seguro a Reportes

Como gerente, quiero visualizar reportes confidenciales solo si tengo permisos válidos.



## 🗂 Estructura del Proyecto

- interfaces/

- IReporte.java

- implementacion/

- ReporteReal.java

- proxy/

- ProxyReporte.java

- cliente/

- Main.java

## 💻 Código Java Expandido

interfaces/IReporte.java

public interface IReporte {

void mostrar();

}

implementacion/ReporteReal.java

public class ReporteReal implements IReporte {

public void mostrar() {

System.out.println("Mostrando reporte confidencial.");

}

}

proxy/ProxyReporte.java

public class ProxyReporte implements IReporte {

private ReporteReal reporteReal;

private String rolUsuario;

public ProxyReporte(String rolUsuario) {

this.rolUsuario = rolUsuario;

}

public void mostrar() {

if ("gerente".equalsIgnoreCase(rolUsuario)) {

if (reporteReal == null) {

reporteReal = new ReporteReal();

}

reporteReal.mostrar();

} else {

System.out.println("Acceso denegado: " + rolUsuario);

}

}

}

cliente/Main.java

public class Main {

public static void main(String[] args) {

IReporte reporte1 = new ProxyReporte("gerente");

reporte1.mostrar();

IReporte reporte2 = new ProxyReporte("empleado");

reporte2.mostrar();

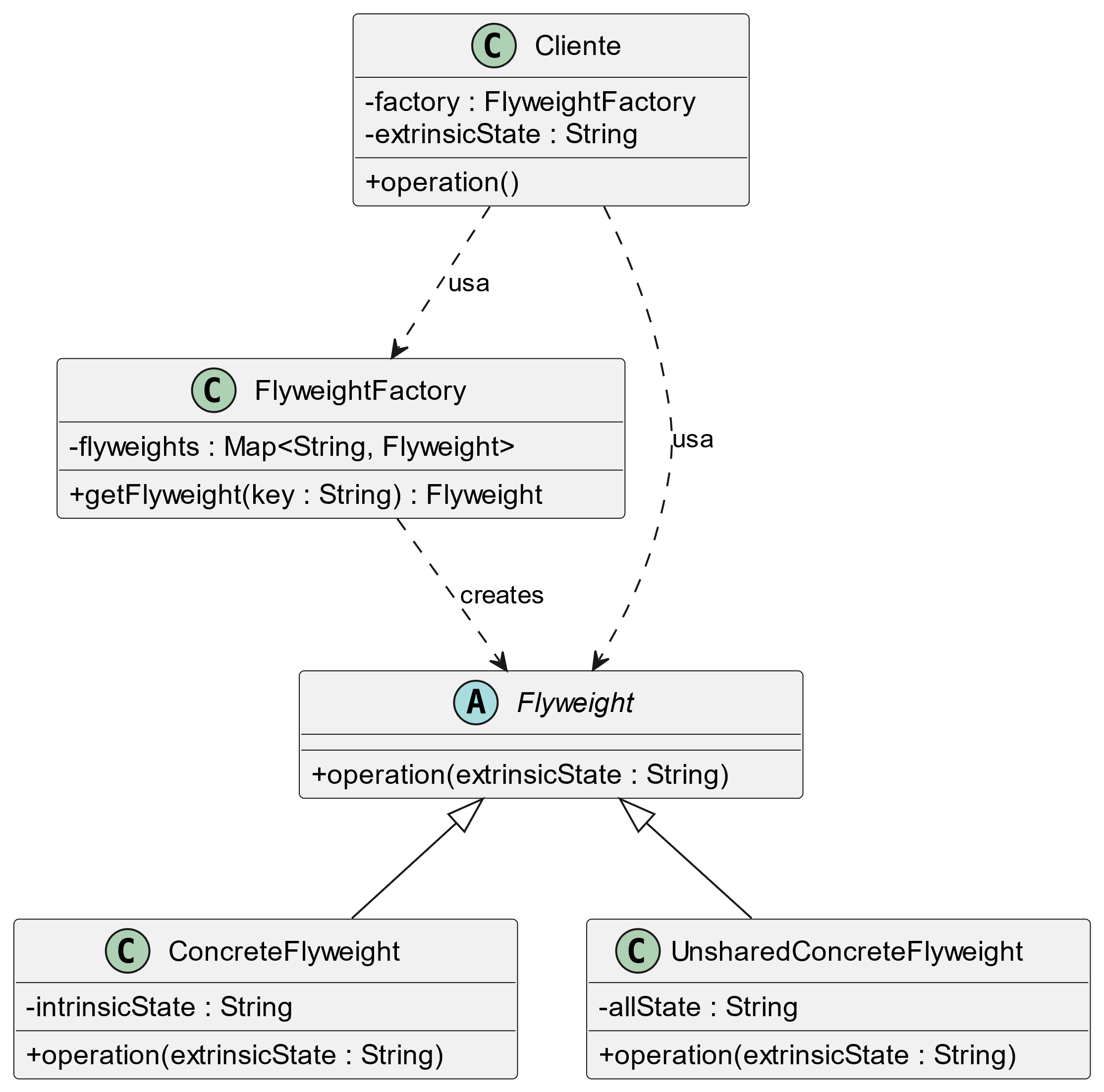
}

}

# Patrón Flyweight

## ¿Qué es el Patrón Flyweight?

El patrón Flyweight permite compartir instancias comunes en vez de duplicarlas, reduciendo el consumo de memoria en aplicaciones con grandes volúmenes de objetos similares.



### 🛠 ¿Cuándo usarlo?

Cuando necesitas manejar millones de objetos similares.

Cuando los objetos comparten estados inmutables.

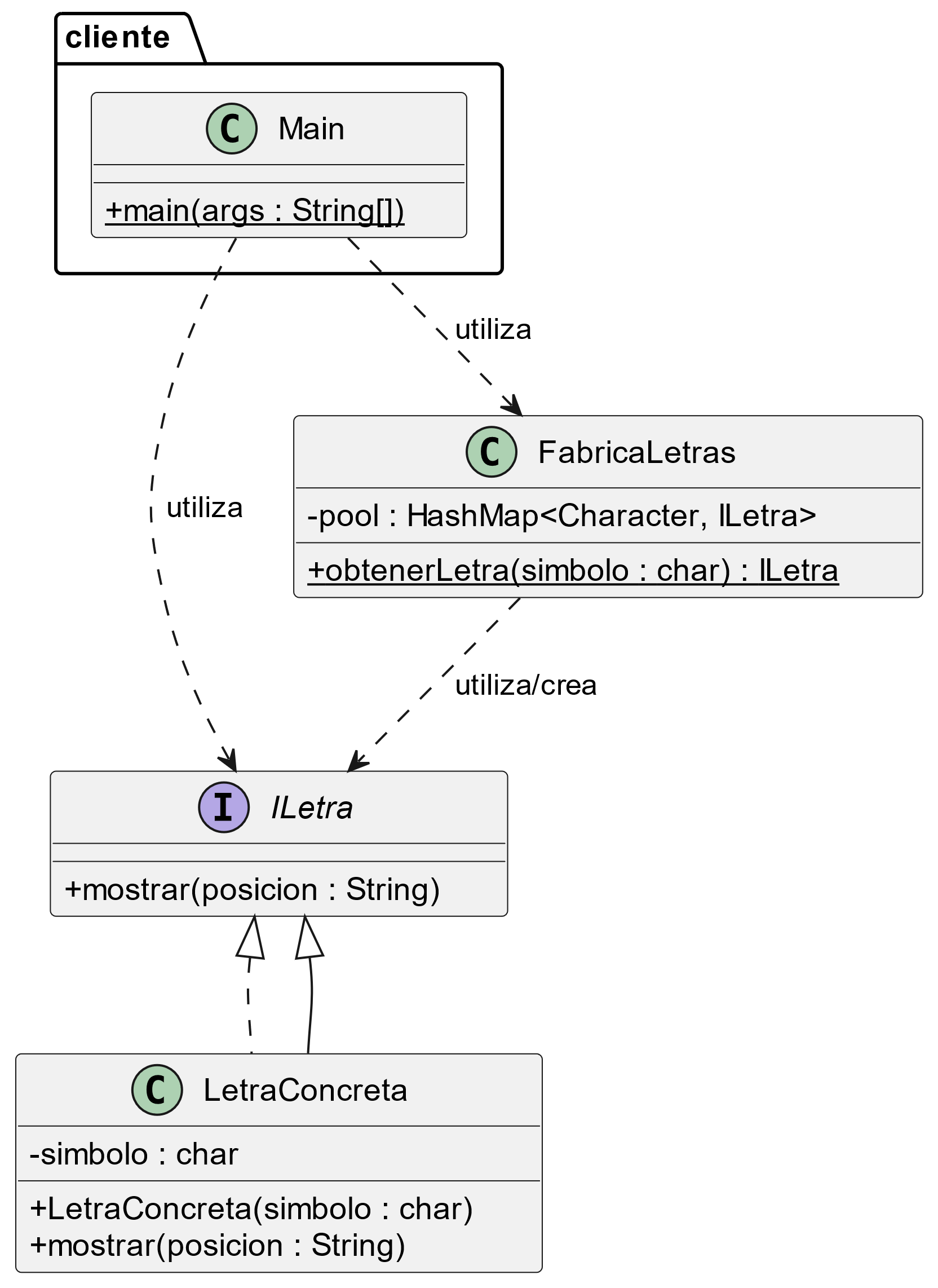
### ⚠️ Antipatrones comunes

Uso innecesario en sistemas pequeños.

Manejo incorrecto de estados internos vs externos.

### 🧑‍💼 Historia de Usuario – Manejo de Letras en un Editor de Texto

Como desarrollador, quiero ahorrar memoria en el editor de texto reutilizando las mismas letras en diferentes posiciones.



## 🗂 Estructura del Proyecto

- interfaces/

- ILetra.java

- implementacion/

- LetraConcreta.java

- fabrica/

- FabricaLetras.java

- cliente/

- Main.java

## 💻 Código Java Expandido

interfaces/ILetra.java

public interface ILetra {

void mostrar(String posicion);

}

implementacion/LetraConcreta.java

public class LetraConcreta implements ILetra {

private final char simbolo;

public LetraConcreta(char simbolo) {

this.simbolo = simbolo;

System.out.println("Creando letra: " + simbolo);

}

public void mostrar(String posicion) {

System.out.println("Letra " + simbolo + " en posición " + posicion);

}

}

fabrica/FabricaLetras.java

import java.util.HashMap;

public class FabricaLetras {

private static final HashMap<Character, ILetra> pool = new HashMap<>();

public static ILetra obtenerLetra(char simbolo) {

if (!pool.containsKey(simbolo)) {

pool.put(simbolo, new LetraConcreta(simbolo));

}

return pool.get(simbolo);

}

}

cliente/Main.java

public class Main {

public static void main(String[] args) {

ILetra letraA1 = FabricaLetras.obtenerLetra('A');

ILetra letraA2 = FabricaLetras.obtenerLetra('A');

ILetra letraB = FabricaLetras.obtenerLetra('B');

letraA1.mostrar("1");

letraA2.mostrar("2");

letraB.mostrar("3");

}

}