



Universidad Politécnica de Madrid

Máster Universitario en Ingeniería Web

Patrones de diseño I

Jesús Bernal Bermúdez

Conceptos de la POO

- ⊙ Cohesión de una clase
- ⊙ Acoplamiento entre clases: dependencia entre clases
- ⊙ Abstracción de clases: herencia e interfaces
- ⊙ Relación de Composición
 - Si la clase A contiene un objeto de la clase B y delega parte de sus funciones en la clase B. Los ciclo de vida coinciden
- ⊙ Relación de Agregación
 - Si un objeto de la clase A delega funcionalidad en los objetos de la clase B, pero el ciclo de vida de ambos no coincide
- ⊙ Relación de Uso
 - Si un objeto de la clase A lanza mensajes al objeto de la clase B para utilizar sus servicios. Es una relación esporádica

Diseño Orientado a Objetos

- ⦿ El paradigma POO es difícil de comprender: polimorfismo
- ⦿ Diseños mantenibles, escalables y reusables
- ⦿ Síntomas de diseños pobres
 - Rigidez. Sistema difícil de cambiar, cualquier cambio sencillo fuerza cambios en cascada
 - Fragilidad. Pequeños cambios hacen que el sistema deje de funcionar
 - Inmovilidad. Reutilizar un módulo requiere muchos cambios
 - Viscosidad. Cuando utilizar un módulo correctamente es muy difícil, incluso puede rentar reimplementarlo
 - Complejidad innecesaria. Contiene estructuras complejas para cambios que nunca llegan
 - Repetición innecesaria. Resultado del copy-paste, muy difícil de mantener
 - Opacidad. Código enrevesado difícil de entender

Antipatrones. Introducción

- ◎ Son soluciones que plantean problemas difíciles de resolver a largo plazo. Resultan tentativos ya que parecen soluciones fáciles a corto plazo
- ◎ Se dan a conocer para no realizarlo en futuros desarrollos
- ◎ Clasificación
 - Desarrollo de software. Se centran en problemas en la implementación de las aplicaciones
 - Arquitectura de software. Relacionados con la estructura de las aplicaciones
 - Gestión de proyectos de software. Relacionados con la estructura organizativa del proyecto

Antipatrones. Desarrollo de software

- ⊙ Clases gigantes (Blob): pocas clases o una sola que realiza toda la operativa de la aplicación
- ⊙ Lava seca (Lava flow): se programa de forma desordenada, con códigos complejos sin documentar y dejando código antiguo inútil. Solución: refactorizar con frecuencia
- ⊙ Código espagueti (Spaghetti code): se basa en el abuso de los if, con estructuras complejas
- ⊙ Cortar y pegar (Cut & paste): se copian trozos de código que se realizan pequeñas modificaciones. Hace el mantenimiento inviable
- ⊙ Fantasmas (Poltergeist): clases y asociaciones que no hacen nada, sólo añaden sobrecarga y complejidad innecesaria
- ⊙ Bucle activo (busy spin): utilizar espera activa cuando existen alternativas: tratamiento de eventos
- ⊙ Comprobación de tipos en lugar de interfaz (checking type instead of interface): comprobar que un objeto es de un tipo concreto (**NO**: `if(c instanceof Clase);`) cuando lo único que se necesita es verificar si cumple un contrato determinado (`UnInterface ui;... ui.m()`)
- ⊙ Ocultación de errores (error hiding): capturar un error antes de que se muestre al usuario, y reemplazarlo por un mensaje sin importancia o ningún mensaje en absoluto

Antipatrones. Desarrollo de software

- ◉ Acoplamiento secuencial (sequential coupling): construir una clase que necesita que sus métodos se invoquen en un orden determinado
- ◉ BaseBean: heredar funcionalidad de una clase utilidad en lugar de delegar en ella. (**NO**: *CuentaKilometro extends Natural*)
- ◉ Singletonitis: abuso de la utilización del patrón singleton
- ◉ Confianza ciega (blind faith): Descuidar la comprobación de los resultados que produce una subrutina, o bien de la efectividad de un parche o solución a un problema
- ◉ Manejo de excepciones (exception handling): emplear el mecanismo de manejo de excepciones del lenguaje para implementar la lógica general del programa. Exception: sólo cuando depende de sistemas ajenos que no se puede garantizar su funcionamiento
- ◉ Manejo de excepciones inútil (useless exception handling): introducir condiciones para evitar que se produzcan excepciones en tiempo de ejecución, pero lanzar manualmente una excepción si dicha condición falla
- ◉ Números mágicos (magic numbers): incluir en los algoritmos números concretos sin explicación aparente
- ◉ Programación por permutación (programming by permutation): Tratar de aproximarse a una solución modificando el código una y otra vez para ver si acaba por funcionar

Antipatrones. Arquitectura de software

- ⦿ Aislamiento en la empresa (Stovepipe enterprise): no se conocen los desarrollos paralelos
- ⦿ Arquitectura dependiente de un fabricante (Vendor Lock-In): los productos de diferentes fabricantes no suelen ser compatibles
- ⦿ Reinventar la rueda (reinventing the wheel): enfrentarse a las situaciones buscando soluciones desde cero, sin tener en cuenta otras que puedan existir ya para afrontar los mismos problemas
- ⦿ Reinventar la rueda cuadrada (reinventing the square wheel): crear una solución pobre cuando ya existe una buena
- ⦿ Martillo de oro (golden hammer): asumir que nuestra solución favorita es universalmente aplicable, haciendo bueno el refrán a un martillo, todo son clavos

Diseño Orientado a Objetos

- ◎ GRASP (General Responsibility Assignment Software Patterns)
Son principios generales de software para asignación de responsabilidades
 - Experto. Asignar una responsabilidad a la clase que cuenta con la información, y dentro de la clase al método oportuno
 - Creador. La instancia debe ser creada por:
 - Quien tiene la información necesaria para realizar la creación del objeto: GUI
 - Usa directamente las instancias creadas del objeto
 - Almacena o maneja varias instancias de la clase: factorías
 - Contiene o agrega la clase
 - Bajo acoplamiento y alta cohesión
 - Polimorfismo
 - Controlador. Hace de intermediario entre la vista y el modelo: MVC
 - Indirección. Utilizar intermediarios entre la comunicación de objetos
 - Variaciones protegidas. Lo que puede ser susceptible de cambiarse se envuelve en un interface permitiendo varias implementaciones

Diseño Orientado a Objetos

- ◎ Principios S.O.L.I.D. Son cinco principios fundamentales en términos de gestión de dependencias, ayudan a NO crear diseños pobres
 - Principio de Responsabilidad Única (Single Responsibility Principle – SRP)
 - Una clase debe ocuparse una sola función, o debe tener un único motivo para cambiar. **No** a la clase *AnguloIntervalo*
 - Principio abierto-cerrado (Open-Closed Principle – OCP)
 - Un módulo debe ser abierto para extender las responsabilidades y añadir nuevos atributos
 - Un módulo debe ser cerrado con su interfaz, para no afectar a otros módulos que dependen de él
 - Principio de Sustitución de Liskov (Liskov Substitution Principle – LSP)
 - Los tipos bases siempre se pueden sustituir por subclases sin cambiar el funcionamiento. **NO** *if(this.getClass().getName().equals("Clase"))...* **NO** a la sobrescritura incompatible con la clase padre
 - Principio de Segregación de Interfaces (Interface Segregation Principle – ISP)
 - No obligar a los clientes a depender de Clases o Interfaces que no necesita. **Mejor** *m(int grados)* a *m(Angulo angulo)*
 - Principio de Inversión de Dependencias (Dependency Inversion Principle – DIP)
 - Los módulos de alto nivel no dependen de los módulos inferiores, deben depender de abstracciones

Diseño Orientado a Objetos

- ◎ Inversión de Control (Inversion of Control- IoC)
 - Cuando una clase (C) tiene clases abstractas colaboradoras: ¿Quién crea las instancias concretas de las clases colaboradoras?
 - Este principio establece que una entidad externa (normalmente a partir de un fichero XML) es la que crea la instancia concreta y se la inyecta a la clase “C” a través de un *constructor* o un método *set*



Patrón de diseño. Introducción

⦿ Motivación

- La POO es difícil, y la POO mantenible y reutilizable todavía mucho más
- Los problemas de los diseño se repiten
- Los diseñadores con experiencia, han depurado buenos diseños

⦿ Un patrón es una solución generalizada a un problema recurrente en el desarrollo de software orientado a objetos

⦿ Describe un conjunto de clases y objetos comunicándose entre sí

⦿ Son soluciones genéricas que se deben adaptar al contexto

⦿ Patrones de diseño. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Addison Wesley, 1995 (GoF- Gang of Four)

Tipos de patrones

<https://github.com/miw-upm/Patrones.git>

⦿ Patrones de creación

- Relacionados con el proceso de creación de objetos
















⦿ Patrones de estructura

- Relacionados con la composición de clases y objetos y su combinación

⦿ Patrones de comportamiento

- Relacionados con el flujo de control y el reparto de responsabilidades

Clasificación de patrones (GoF)

Creación	Estructural	Comportamiento
 Abstract Factory	 Adapter	Chain of Responsibility
Builder	Bridge	 Command
 Factory Method	 Composite	Interpreter
Prototype	 Decorator	 Iterator
 Singleton	 Facade	Mediator
	 Flyweight	 Memento
	 Proxy	 Observer
		 State
		Strategy
		Template Method
		 Visitor

Singleton (Único)

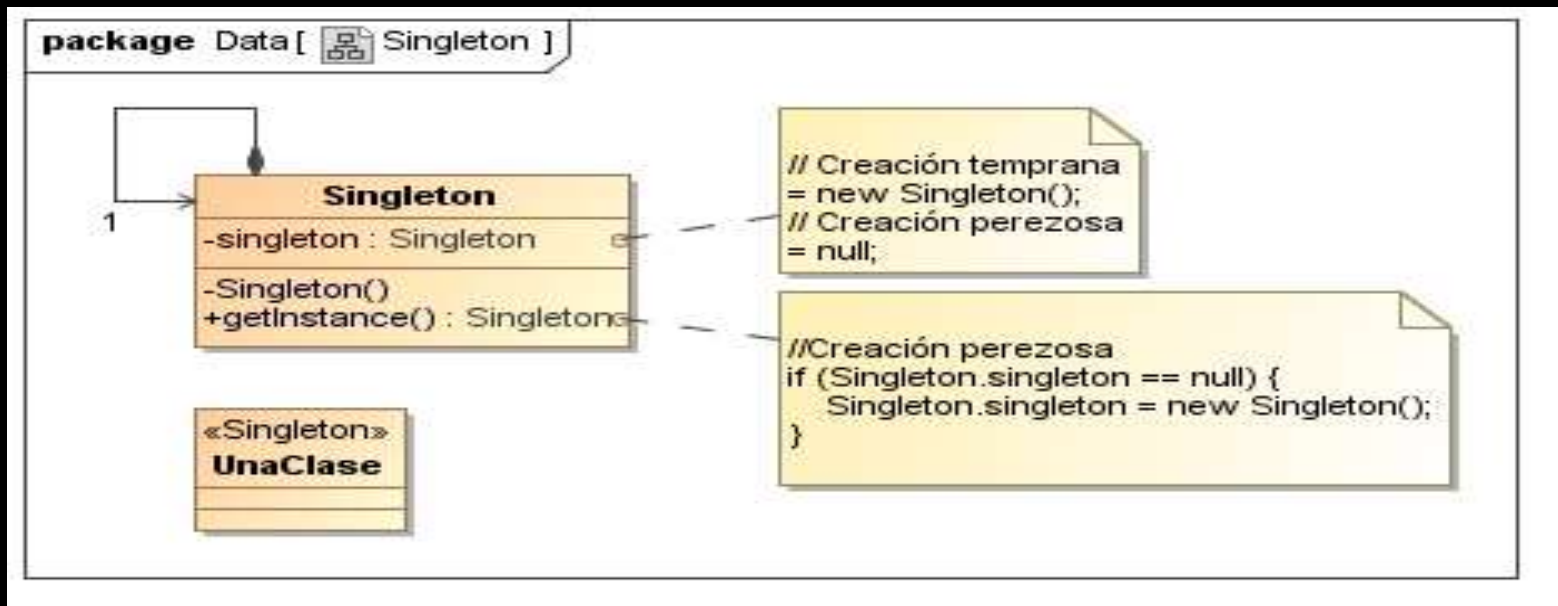
○ Motivación

- En una aplicación se necesita que exista un solo Gestor de Ficheros (Cola de Impresión, Gestor de Registros, Gestor de BD...), accesible desde muchas clases

○ Propósito

- Se garantiza que una clase sólo tenga una instancia y se proporciona un acceso desde cualquier otra clase

○ Implementación



Singleton . Implementación

```
public final class Singleton {  
    private static Singleton singleton = null;  
  
    private Singleton() { };  
  
    public static Singleton getInstance() {  
        //Creación perezosa  
        if (Singleton.singleton == null) {  
            Singleton.singleton = new Singleton();  
        }  
        return Singleton.singleton;  
    }  
}
```

Creación temprana: se crea en este punto.

No se permite crear instancias desde otras clases

Creación temprana. No hace falta realizar el if...

```
public class SingletonTest {  
    @Test public void testIsSingleton() {  
        assertEquals(Singleton.getSingleton(), Singleton.getSingleton());  
    }  
    @Test public void testSingletonNotNull() {  
        assertNotNull(Singleton.getSingleton());  
    }  
}
```

Singleton. Ejercicio dirigido Factory

- © Dada la clase Factory (gestión de objetos), aplicar el patrón Singleton
 - Realizar una creación perezosa

Clase Factory

Test de Factory

Singleton. Ejercicio dirigido Factory

- ⦿ Se debe respetar la funcionalidad original
- ⦿ Lista de roles
- ⦿ Añadir test de singleton, respetando test anterior
- ⦿ Probar

Singleton. Ejercicio. Logger

- ⦿ Dada la clase Logger (Registrador), aplicar el patrón Singleton
 - Realizar una creación temprana

```
public class Logger {  
    private String logs;  
    public Logger() {  
        this.clear();  
    }  
    public String getLogs() {  
        return logs;  
    }  
    public void addLog(String log) {  
        this.logs = this.logs + ">>> " + log + "\n";  
    }  
    public void clear() {  
        this.logs = new Date().toString() + "\n";  
    }  
    public void print() {  
        IO.out.print(this.logs);  
    }  
}
```

```
public class LoggerTest {  
    @Test public void testIsSingleton() {  
        assertSame("No es la misma referencia", Logger.getLogger(), Logger.getLogger());  
    }  
    @Test public void testSingletonNotNull() {  
        assertNotNull(Logger.getLogger());  
    }  
}
```

Singleton. Ejercicio. Printer

- Objetos Mock. Son objetos que imitan el comportamiento de objetos reales de una forma controlada. Se usan para probar a otros objetos en pruebas unitarias. Normalmente son objetos pertenecientes a otros paquetes o capas
- Se parte de la clase ***PrinterFactory***, es una factoría que puede contener un solo objeto de tipo ***Printer*** (interface)
- Se disponen de dos impresoras: ***PrinterAMock*** y ***PrinterBMock***, pero sólo una de ellas puede estar activa. Ambas ofrecen el método “*print(String msg)*”, imprime un mensaje
- Se aporta la clase ***PrinterTest*** para probar las clases
- Se pide, convertir la clase ***PrinterFactory*** en singleton

Printer. PrinterFactory y Printer

```
public class PrinterFactory {  
    private Printer printer = null;  
  
    public Printer getPrinter() {  
        return this.printer;  
    }  
  
    public void setPrinter(Printer printer) {  
        this.printer = printer;  
    }  
}
```

```
public interface Printer {  
    void print(String msg);  
}
```

Printer. Mocks

```
public class PrinterAMock implements Printer {  
    @Override  
    public void print(String msg) {  
        //System.out.println("A:" + msg);  
    }  
    @Override  
    public String toString() {  
        return "PrinterA";  
    }  
}
```

```
public class PrinterBMock implements Printer {  
    @Override  
    public void print(String msg) {  
        //System.out.println("B:" + msg);  
    }  
    @Override  
    public String toString() {  
        return "PrinterB";  
    }  
}
```

Printer. Test

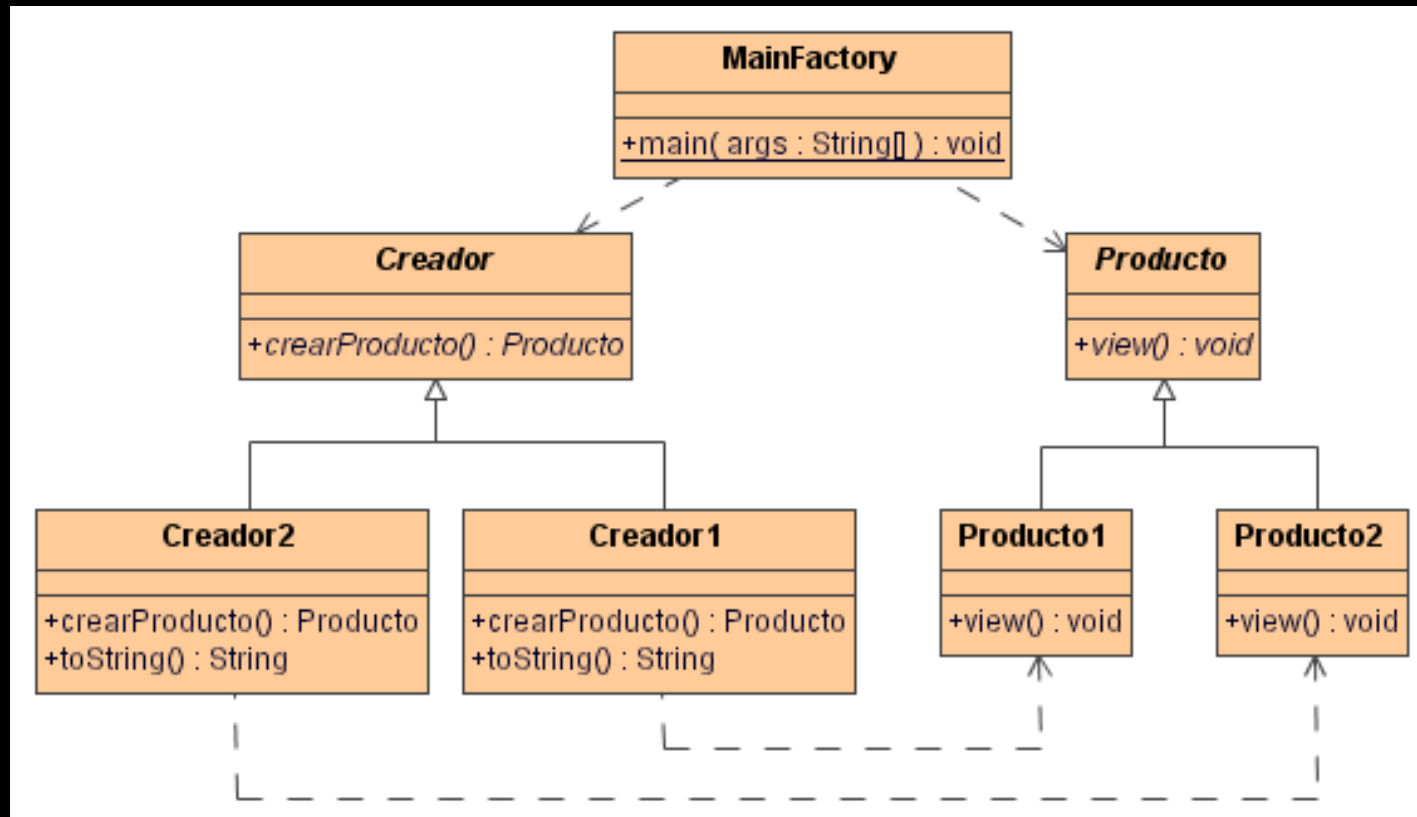
```
public class PrinterTest {  
    @Test  
    public void testIsSingleton() {  
        assertSame(PrinterFactory.getPrinterFactory(),  
                    PrinterFactory.getPrinterFactory());  
    }  
    @Test  
    public void testSingletonNotNull() {  
        assertNotNull(Logger.getLogger());  
    }  
    @Test  
    public void testIsFactory() {  
        PrinterAMock pam = new PrinterAMock();  
        PrinterBMock pbm = new PrinterBMock();  
        PrinterFactory.getPrinterFactory().setPrinter(pam);  
        assertSame(pam, PrinterFactory.getPrinterFactory().getPrinter());  
        assertSame(pam, PrinterFactory.getPrinterFactory().getPrinter());  
        PrinterFactory.getPrinterFactory().setPrinter(pbm);  
        assertSame(pbm, PrinterFactory.getPrinterFactory().getPrinter());  
    }  
}
```

Factory Method (Método factoría)

- ◎ También conocido
 - Virtual Builder (Constructor Virtual)
- ◎ Motivación
 - Dado un framework para presentar documentos (clase abstracta), se necesita crear documentos sin depender del tipo de documento
 - Dispone de dos clases abstractas: Aplicación y Documento. Cuando a la clase Aplicación le piden crear un nuevo documento, deja que la subclase decida el tipo de documento apropiado
- ◎ Propósito
 - Define una abstracción para crear objetos, y son las subclases que deciden la clase concreta a instanciar

Factory Method. Implementación

- Define una abstracción para crear objetos, y son las subclases que deciden la clase concreta a instanciar



Factory Method. Implementación

```
public abstract class Producto {
    public abstract void view();
}

public class Producto1 extends Producto {
    @Override public void view() {
        IO.out.println("FactoryMethod.Producto1");
    }
}

public class Producto2 extends Producto {
    @Override public void view() {
        IO.out.println("FactoryMethod.Producto2");
    }
}
```

```
public abstract class Creador {
    public abstract Producto crearProducto();
}

public class Creador1 extends Creador {
    @Override public Producto crearProducto() {
        return new Producto1();
    }
    @Override public String toString() { return "Creador de Producto1"; }
}

public class Creador2 extends Creador {
    @Override public Producto crearProducto() {
        return new Producto2();
    }
    @Override public String toString() { return "Creador de Producto2"; }
}
```

Factory Method. Implementación

```
public class FactoryMain {  
    private Creador[] creadores = {new Creador1(), new Creador2()};  
    private Creador creador = creadores[0];  
  
    public void tipoCreador() {  
        this.creador = (Creador) IO.in.select(creadores, "Elige un creador");  
    }  
  
    public void crearProducto() {  
        this.creador.crearProducto().view();  
    }  
  
    public static void main(String[] args) {  
        IO.in.addController(new MainFactory());  
    }  
}
```

Factory Method. Ejercicio. Numero

- Se parte de la clase NumeroEs, la cual convierte un valor numérico a formato texto (Español). Se pretende poder manejar varios idiomas: NumeroEN, NumeroFr... Además, se quiere abstraerse del idioma mediante la clase Numero
- Por otro lado, se necesita crear instancias de algún número concreto, pero queremos abstraernos del idioma
- Se presenta el contenido de la clase NumeroEs, la clase NumeroTest y una aplicación de prueba: MainNumero
- Implementar las clases que faltan

```
public class NumeroES {  
    private final static String[]  
        texto = {"cero", "uno", "dos", "tres", "cuatro", "cinco"};  
    public String convertir(int numero) {  
        if (numero >= 0 && numero < texto.length) {  
            return texto[numero];  
        } else {  
            return "???";  
        }  
    }  
}
```

Factory Method. Ejercicio. Numero

```
public class NumeroTest {  
    private CreadorNumero creadorEn = new CreadorNumeroEN();  
    private CreadorNumero creadorEs = new CreadorNumeroES();  
  
    @Test  
    public void testIsNew() {  
        Numero num = creadorEn.createNumero();  
        assertNotSame(creadorEn.createNumero(), num);  
    }  
  
    @Test  
    public void testConvertir() {  
        assertEquals("0 a ES", "cero",  
                     this.creadorEs.createNumero().convertir(0).toLowerCase());  
        assertEquals("5 a EN", "five",  
                     this.creadorEn.createNumero().convertir(5).toLowerCase());  
    }  
}
```

Factory Method. Ejercicio. Numero

```
public class NumeroMain {  
    private CreadorNumero[] creadores = {new CreadorNumeroEN(),  
                                           new CreadorNumeroES()};  
    private CreadorNumero creador = creadores[0];  
  
    public void tipoCreador() {  
        this.creador = (CreadorNumero) IO.in.select(creadores, "Tipo creador");  
    }  
  
    public void crearNumero() {  
        Numero num = creador.createNumero();  
        IO.out.println(num.convertir(IO.in.readInt("Numero")));  
    }  
  
    public static void main(String[] args) {  
        IO.in.addController(new MainNumero());  
    }  
}
```

State (Estado)

◎ También conocido

- Objects for States (Estados como Objetos)

◎ Motivación

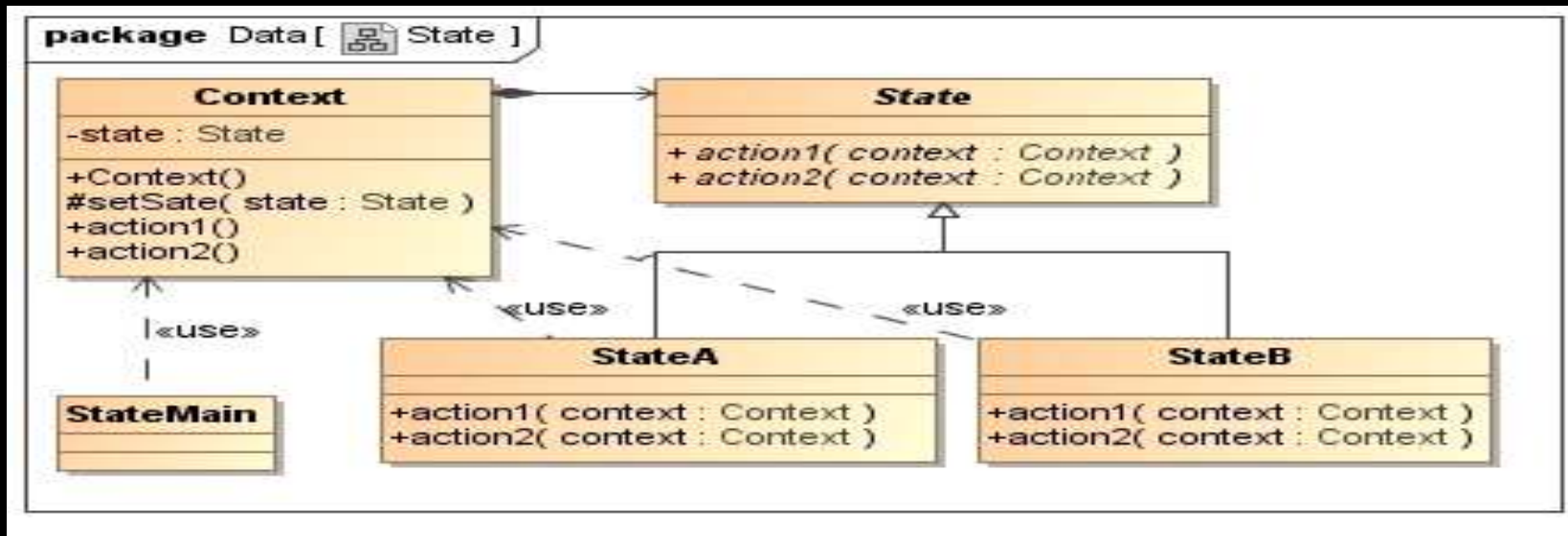
- Partiendo de una clase que representa una conexión TCP, la acción enviar debe tener diferentes respuestas dependiendo del estado de la conexión. En lugar de establecer una secuencia de if-else preguntando por el estado se codifica en varias clases

◎ Propósito

- Permite que un objeto cambie su comportamiento cada vez que cambie su estado interno

State. Implementación

- ◉ Permite que un objeto cambie su comportamiento cada vez que cambie su estado interno
- ◉ Creación de objetos:
 - Crearlos nuevos cada vez que se necesite
 - Crearlos al principio y no destruirlos



State. Implementación

```
public abstract class State {  
    public abstract void action1(Context context);  
    public abstract void action2(Context context);  
}
```

```
public class StateA extends State {  
    @Override public void action1(Context context) {  
        IO.out.println("Accion1 en estado A: se cambia a estado B");  
        context.setState(new StateB());  
    }  
    @Override public void action2(Context context) {  
        IO.out.println("Accion2 en estado A");  
    }  
    @Override public String toString() {  
        return "Estado A";  
    }  
}
```

```
public class StateB extends State {  
    @Override public void action1(Context context) {  
        IO.out.println("Accion1 en estado B");  
    }  
    @Override public void action2(Context context) {  
        IO.out.println("Accion2 en estado B: se cambia a estado A");  
        context.setState(new StateA());  
    }  
    @Override public String toString() {  
        return "Estado B";  
    }  
}
```

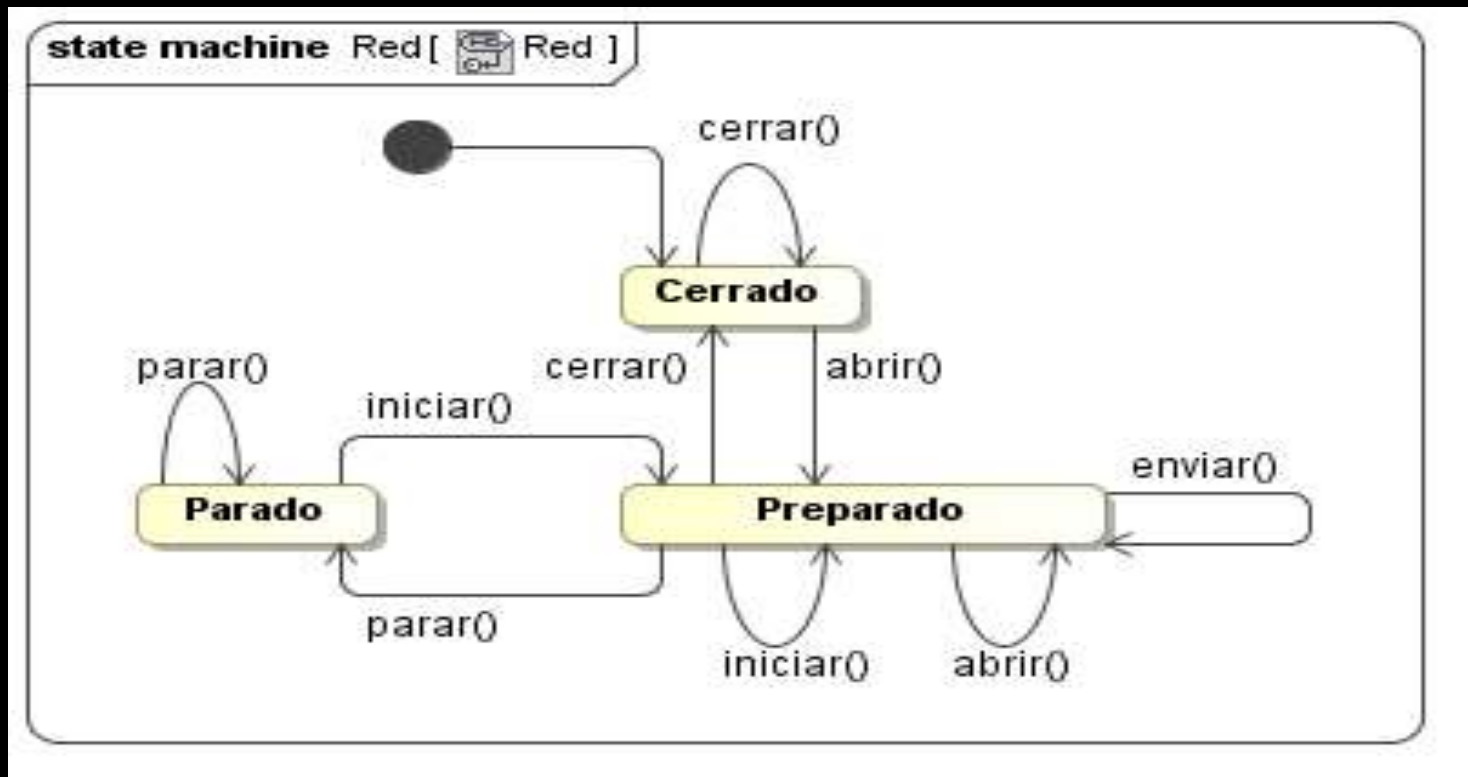

State. Implementación

```
public class Context {
    private State state;
    public Context() {
        this.state = new StateA();
    }
    protected void setState(State state) { this.state = state; }
    public void action1() {
        state.action1(this);
    }
    public void action2() {
        state.action2(this);
    }
    @Override public String toString() {
        return "Context[" + state.toString() + "]";
    }
}
```

```
public class StateMain {
    private Context c = new Context();
    public void state() {
        IO.out.println(this.c.toString());
    }
    public void accion1() {
        this.c.action1();
    }
    public void accion2() {
        this.c.action2();
    }
    public static void main(String[] args) {
        IO.in.addController(new StateMain());
    }
}
```

State. Ejercicio. Conexión de red

- ⦿ Crear nuevas instancias de estado, cada vez que se necesite
- ⦿ Sólo se permiten las acciones marcadas, el resto debe lanzar la excepción: *UnsupportedOperationException*



State. Ejercicio. Conexión de red

SRC: Ejercicio Red

```
public class ConexionSinPatron {
    private Estado estado;
    private Emisor emisor;
    public ConexionSinPatron() {this.estado = Estado.CERRADO;}
    public void setEmisor(Emisor emisor) {this.emisor = emisor;}
    public Estado estado() {return this.estado;}
    public void abrir() {
        if (this.estado == Estado.CERRADO) {
            this.estado = Estado.PREPARADO;
        } else if (this.estado == Estado.PARADO) {
            throw new UnsupportedOperationException("Acción no permitida... ");
        } else if (this.estado == Estado.PREPARADO) {
            else assert false : "estado imposible";
        }
    }
    public void cerrar() {
        if (this.estado == Estado.CERRADO) {
        } else if (this.estado == Estado.PARADO) {
            throw new UnsupportedOperationException("Acción no permitida... ");
        } else if (this.estado == Estado.PREPARADO) {
            this.estado = Estado.CERRADO;
        } else assert false : "estado imposible";
    }
    public void parar() {
        if (this.estado == Estado.CERRADO) {
            throw new UnsupportedOperationException("Acción no permitida... ");
        } else if (this.estado == Estado.PARADO) {
        } else if (this.estado == Estado.PREPARADO) {
            this.estado = Estado.PARADO;
        } else assert false : "estado imposible";
    }
    public void iniciar() {
        if (this.estado == Estado.CERRADO) {
            throw new UnsupportedOperationException("Acción no permitida... ");
        } else if (this.estado == Estado.PARADO) {
            this.estado = Estado.PREPARADO;
        } else if (this.estado == Estado.PREPARADO) {
        } else assert false : "estado imposible";
    }
    public void enviar(String msg) {
        if (this.estado == Estado.CERRADO) {
            throw new UnsupportedOperationException("Acción no permitida... ");
        } else if (this.estado == Estado.PARADO) {
            throw new UnsupportedOperationException("Acción no permitida... ");
        } else if (this.estado == Estado.PREPARADO) {
            this.emisor.enviar(msg);
        } else assert false : "estado imposible";
    }
}
```

State. Ejercicio. Conexión de red

SRC: Ejercicio Red

```
public class ConexionTest {
    private ConexionSinPatron conexion;
    private EmisorMock emisor;
    @Before public void ini() {
        this.conexion = new ConexionSinPatron();
        this.emisor = new EmisorMock();
        this.conexion.setEmisor(emisor); // Se inyecta el emisor en la conexion
    }
    @Test public void testEstadoInicial() {
        assertEquals(Estado.CERRADO, this.conexion.estado());
    }
    @Test public void testCerradoAbrir() {
        this.conexion.abrir();
        assertEquals(Estado.PREPARADO, this.conexion.estado());
    }
    @Test public void testCerradoCerrar() {
        this.conexion.cerrar();
        assertEquals(Estado.CERRADO, this.conexion.estado());
    }
    @Test public void testCerradoNoSoportadoParar() {
        try { this.conexion.parar(); fail(); }
        catch (UnsupportedOperationException ignored) { ignored.toString(); }
    }
    @Test public void testCerradoNoSoportadoIniciar() {
        try { this.conexion.iniciar(); fail(); }
        catch (UnsupportedOperationException ignored) { ignored.toString(); }
    }
    @Test public void testCerradoNoSoportadoEnviar() {
        try { this.conexion.enviar(""); fail(); }
        catch (UnsupportedOperationException ignored) { ignored.toString(); }
    }
}
```

State. Ejercicio. Conexión de red

SRC: Ejercicio Red

```
@Test public void testPreparadoAbrir() {
    this.conexion.abrir(); this.conexion.abrir();
    assertEquals(Estado.PREPARADO, this.conexion.estado());
}
@Test public void testPreparadoCerrar() {
    this.conexion.abrir(); this.conexion.cerrar();
    assertEquals(Estado.CERRADO, this.conexion.estado());
}
@Test public void testPreparadoParar() {
    this.conexion.abrir(); this.conexion.parar();
    assertEquals(Estado.PARADO, this.conexion.estado());
}
@Test public void testPreparadoIniciar() {
    this.conexion.abrir(); this.conexion.iniciar();
    assertEquals(Estado.PREPARADO, this.conexion.estado());
}
@Test public void testPreparadoEnviar() {
    this.conexion.abrir(); this.conexion.enviar("...");
    assertEquals(Estado.PREPARADO, this.conexion.estado());
    assertEquals("...", emisor.getMsg());
}
@Test public void testParadoNoSoportadoAbrir() {
    this.conexion.abrir(); this.conexion.parar();
    try {this.conexion.abrir();fail();}
    catch (UnsupportedOperationException ignored) {ignored.toString();}
}
@Test public void testParadoNoSoportadoCerrar() {
    this.conexion.abrir(); this.conexion.parar();
    try {this.conexion.cerrar();fail();}
    catch (UnsupportedOperationException ignored) {ignored.toString();}
}
@Test public void testParadoParar() {
    this.conexion.abrir(); this.conexion.parar(); this.conexion.parar();
    assertEquals(Estado.PARADO, this.conexion.estado());
}
@Test public void testParadoIniciar() {
    this.conexion.abrir(); this.conexion.parar(); this.conexion.iniciar();
    assertEquals(Estado.PREPARADO, this.conexion.estado());
}
@Test public void testParadoNoSoportadoEnviar() {
    this.conexion.abrir(); this.conexion.parar();
    try {this.conexion.enviar("");fail();}
    catch (UnsupportedOperationException ignored) {ignored.toString();}
}
}
```

Composite (Compuesto)

◎ Motivación

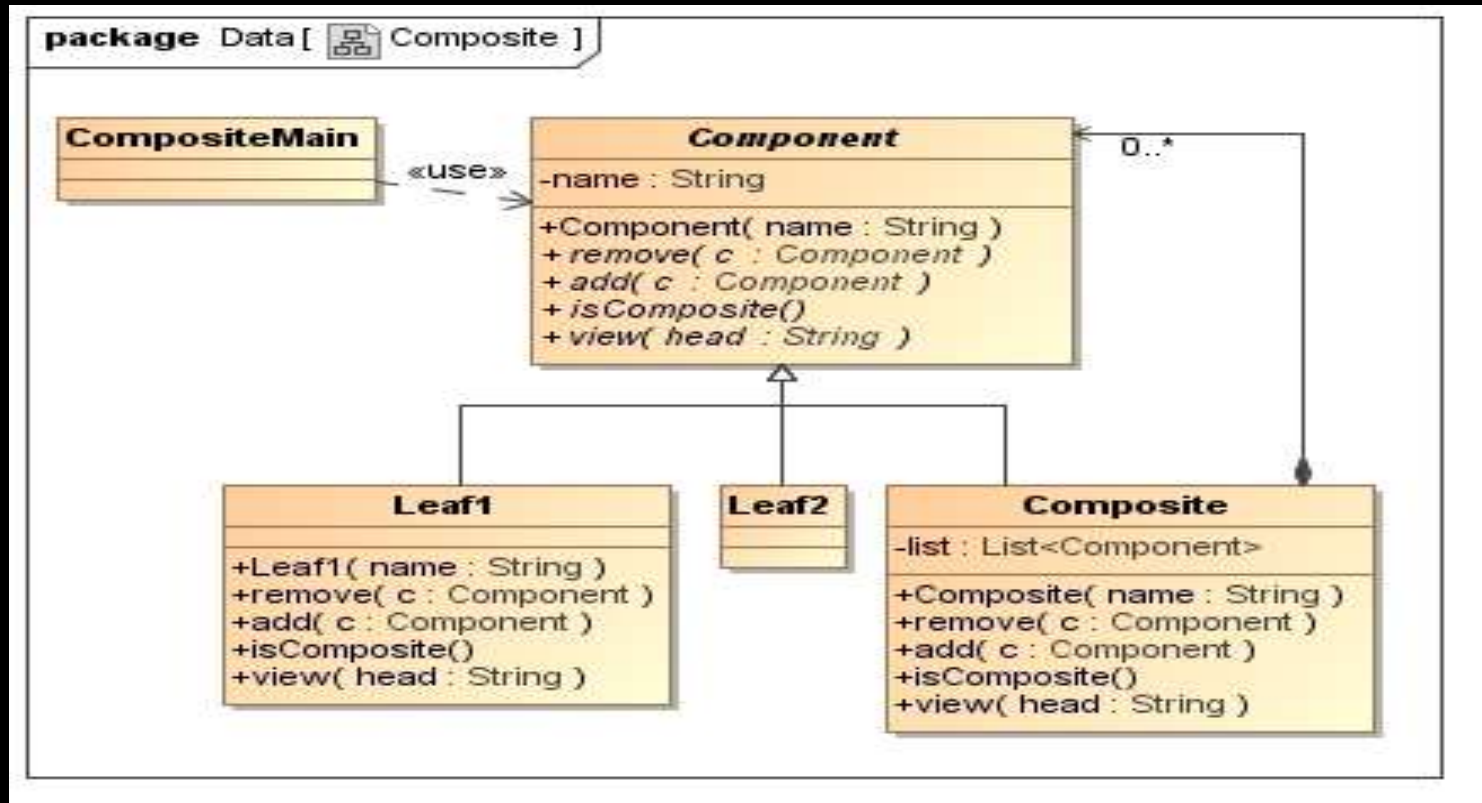
- En las aplicaciones gráficas, se permiten realizar dibujos por la agrupación de elementos simples y otros elementos agrupados

◎ Propósito

- Permite estructuras en árbol tratando por igual a las hojas que a los elementos compuestos

Composite. Implementación

- Permite estructuras en árbol tratando por igual a las hojas que a los elementos compuestos



Composite. Implementación

```
public abstract class Component {  
    private String name;  
  
    public Component(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
  
    public abstract void remove(Component cc);  
  
    public abstract void add(Component cc);  
  
    public abstract boolean isComposite();  
  
    public abstract void view(String head);  
}
```


Composite. Implementación

```
public class Leaf1 extends Component {
    public Leaf1(String nombre) {
        super(nombre);
    }
    @Override
    public void view(String cabecera) {
        IO.out.println(cabecera + "-" + this.toString());
    }
    @Override
    public void remove(Component cc) {
        throw new UnsupportedOperationException("Operación no soportada");
    }
    @Override
    public void add(Component cc) {
        throw new UnsupportedOperationException("Operación no soportada");
    }
    @Override
    public boolean isComposite() {
        return false;
    }
    @Override
    public String toString() {
        return "H1:" + this.getName().toLowerCase();
    }
}
```

Composite. Implementación

```
public class Composite extends Component {
    private java.util.List<Component> list;
    public Composite(String name) {
        super(name);
        this.list = new java.util.ArrayList<Component>();
    }
    @Override
    public void view(String head) {
        IO.out.println(head + "-" + this.getName() + ":");
        for (Component item : list) {
            item.view(head + " ");
        }
    }
    @Override public void add(Component cc) {
        list.add(cc);
    }
    @Override public void remove(Component cc) {
        list.remove(cc);
    }
    @Override public boolean isComposite() {
        return true;
    }
    @Override public String toString() {
        return "C:" + this.getName().toLowerCase();
    }
}
```

Composite. Implementación

```
public class CompositeMain {
    private List<Component> compuestos = new ArrayList<Component>();
    private Component raiz;

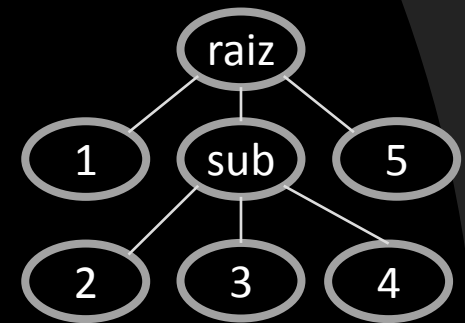
    public CompositeMain() {
        this.raiz = new Composite("raiz");
        this.compuestos.add(raiz);
    }
    public void addCompuesto() {
        Component com = (Component) IO.in.select(compuestos.toArray());
        Component nuevo = new Composite(IO.in.readString("Nombre"));
        com.add(nuevo);
        this.compuestos.add(nuevo);
    }
    public void addHoja1() {
        Component com = (Component) IO.in.select(compuestos.toArray());
        com.add(new Leaf1(IO.in.readString("Nombre")));
    }
    public void addHoja2() {
        Component com = (Component) IO.in.select(compuestos.toArray());
        com.add(new Leaf2(IO.in.readString("Nombre")));
    }
    public void view() {
        IO.out.clear();
        this.raiz.view("");
    }
    public static void main(String[] args) {
        IO.in.addController(new CompositeMain());
    }
}
```

Composite. Ejercicio. Arbol

- ⦿ Se quiere construir una estructura de árbol. Existen nodos con valores numéricos (NodoHoja) y nodos que contienen otros nodos (NodoComposite)
- ⦿ Si se intenta añadir a un nodo hoja, se debe lanzar la excepción: *UnsupportedOperationException*. Si se intenta borrar nodos a una hoja no se hace nada
- ⦿ Si se intenta añadir o borrar con parámetro null, no debe dar problemas. **OJO, no dar problemas significa que debemos asegurarnos que el parámetro no es null para añadirlo**
- ⦿ Se deben crear las operaciones numHijos(), suma() y mayor(), se opera sobre si mismo y todos los nodos que dependen de él
- ⦿ El test se realiza mediante la clase NodoTest

Composite. Ejercicio. Arbol

```
public class NodoTest {
    private NodoComponente raiz;
    private NodoComponente inter;
    private NodoHoja hoja;
    @Before public void ini() {
        this.raiz = new NodoCompuesto("raiz");
        this.hoja = new NodoHoja(1); this.raiz.add(hoja);
        this.inter = new NodoCompuesto("sub");
        this.inter.add(new NodoHoja(4)); this.inter.add(new NodoHoja(3));
        this.inter.add(new NodoHoja(2)); NodoComponente no = new NodoHoja(-1);
        this.inter.add(no); this.inter.remove(no);
        this.raiz.add(this.inter);
        this.raiz.add(new NodoHoja(5));
    }
    @Test public void testNumHijos() {
        assertEquals(3, this.raiz.numHijos());
        assertEquals(3, this.inter.numHijos());
        assertEquals(0, this.hoja.numHijos());
    }
    @Test public void testAddNull() {
        this.raiz.add(null);
        assertEquals(3, this.raiz.numHijos());
    }
    @Test public void testRemoveNull() {
        this.raiz.remove(null);
    }
    @Test public void testSuma() {
        assertEquals(15, this.raiz.suma());
        assertEquals(9, this.inter.suma());
    }
    @Test public void testMayor() {
        assertEquals(5, this.raiz.mayor());
        assertEquals(4, this.inter.mayor());
    }
    @Test public void testNoSoportado() {
        NodoComponente excp = new NodoHoja(0);
        try {excp.add(new NodoHoja(2)); fail();}
        catch (UnsupportedOperationException ignored) {ignored.toString();}
    }
}
```



Composite. Ejercicio. Expresiones

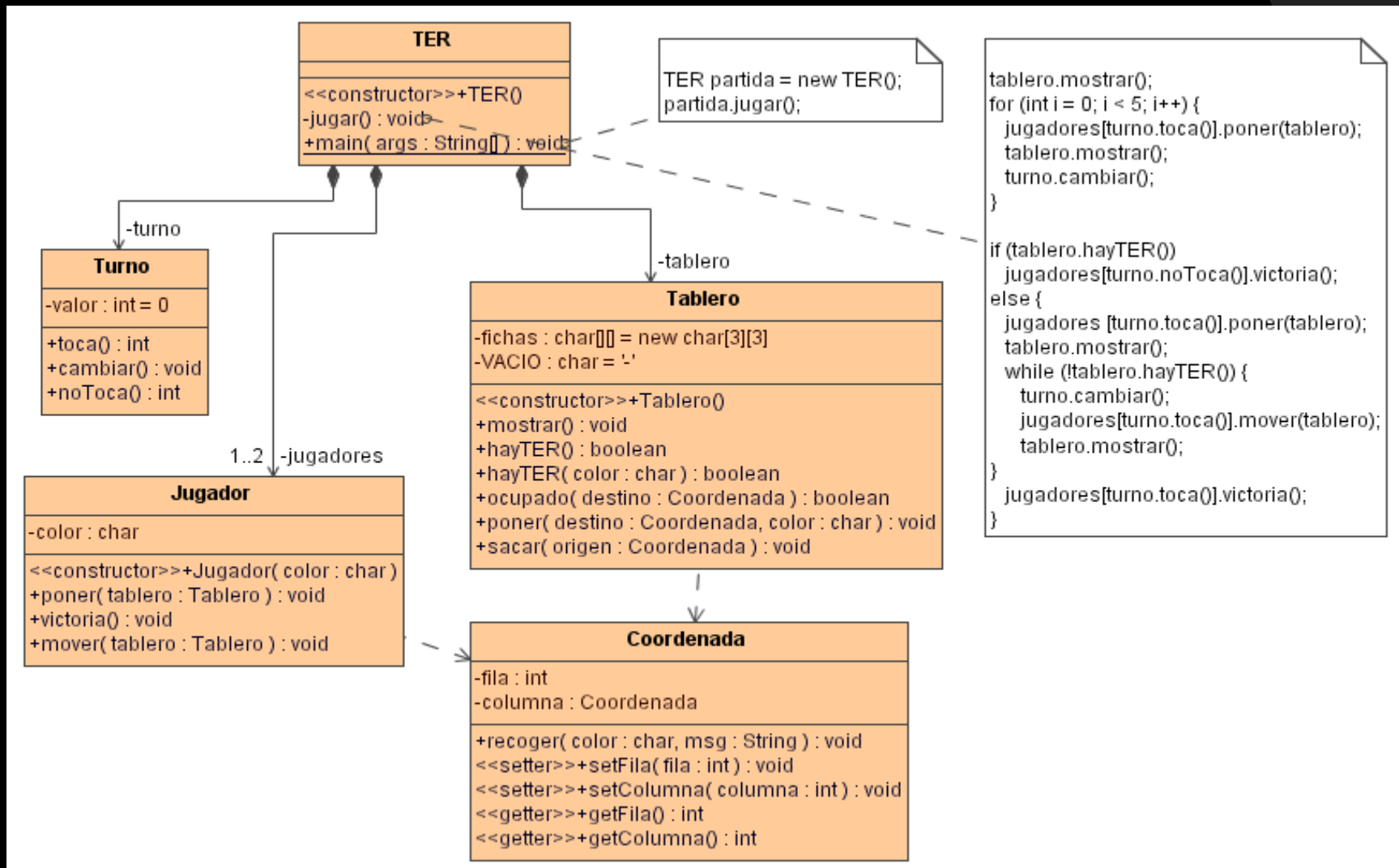
- ⦿ Se quiere construir un editor de expresiones matemáticas con valores enteros. Especificar el diagrama de clases que permita representar las expresiones
- ⦿ Una expresión válida estará formada o bien por un número, o bien por la suma, resta, división o multiplicación de dos expresiones
- ⦿ Ejemplos de expresiones válidas:
 - 5
 - $(1+(8*3))$
 - $((7+3) * (1+5))$
- ⦿ La clase de test es: ExpressionTest

Composite. Ejercicio. Expresiones.Test

Esquema UML...

```
public class ExpresionTest {
    private Expression exp1, exp2, exp3, exp4, exp5, exp6;
    @Before public void ini() {
        exp1 = new Numero(4); exp2 = new Sumar(exp1, new Numero(2));
        exp3 = new Restar(exp1, new Numero(3));
        exp4 = new Multiplicar(exp1, new Numero(2));
        exp5 = new Dividir(exp1, new Numero(3));
        exp6 = new Sumar(new Restar(new Numero(3), new Multiplicar(
            new Dividir(exp1, new Numero(2)), new Numero(3))), exp1);
    }
    @Test public void testValor() {assertEquals(4, exp1.operar());}
    @Test public void TestSuma() {assertEquals(6, exp2.operar());}
    @Test public void testResta() {assertEquals(1, exp3.operar());}
    @Test public void testMultiplicacion() {assertEquals(8, exp4.operar());}
    @Test public void testDivision() {assertEquals(1, exp5.operar());}
    @Test public void testCompuesto() {assertEquals(1, exp6.operar());}
    @Test public void testVer() {
        assertEquals("((3-((4/2)*3))+4)", exp6.ver());
    }
}
```

TER (Tres en Raya)



TER (Tres en Raya). Mejoras

- ⦿ Permitir manejar las coordenadas en formato letra o numérica
- ⦿ Permitir que un jugador sea manual o automático
- ⦿ Evitar el tener que pasar la referencia del tablero