DigitalHouse> Coding School

Nº 1, AÑO 2022

MANUAL DEL DOCENTE

WWW.DIGITALHOUSE.COM

GUÍA PRÁCTICA

MATERIALES

BOOTCAMP MERCADO LIBRE

LO NUEVO

INFORMACIÓN DE INTERÉS

mercado libre

Índice

Objetivos

Al alcanzar estas pautas

// Estaremos preparando a personas que puedan afrontar cualquier desafío

• Interpretar especificaciones de diseño que le permitan construir el código en el contexto del desarrollo de software en el que participa el alumno.

• Dimensionar el trabajo en el contexto del proyecto de desarrollo de software.

• Realizar pruebas unitarias y de sistemas. Verificar el código desarrollado, utilizando revisiones técnicas.

• Analizar Errores de código

• Elaborar documentación técnica de acuerdo con los requerimientos funcionales y técnicos recibidos. Integrar un equipo en el contexto de un Proyecto de Desarrollo de Software.

• Interpretar las especificaciones formales o informales del Líder de proyecto

• Analizar el problema a resolver • Interpretar el material recibido y clarificar eventuales interpretaciones

• Determinar el alcance del problema y convalidar su interpretación a fin de identificar aspectos faltantes

JAVA

Repaso módulo JAVA

// Conceptos básicos

Clases y Objetos

Las clases son el centro del paradigma de Programación Orientada a Objetos (POO). Algunos conceptos importantes de la POO son los siguientes:

1. Encapsulamiento: Las clases pueden ser declaradas como públicas (public) y como paquete (package) (accesibles sólo para otras clases del mismo paquete). Las variables miembros y los métodos pueden ser public, private, protected y package. De esta forma se puede controlar el acceso entre objetos y evitar un uso inadecuado.

2. Herencia: Una clase puede derivar de otra (extends), y en ese caso hereda todas sus variables y métodos. Una clase derivada puede añadir nuevas variables y métodos y/o redefinir las variables y métodos heredados.

3. Polimorfismo: Los objetos de distintas clases pertenecientes a una misma jerarquía o que implementan una misma interface, pueden responder de forma indistinta a un mismo método. Esto, como se ha visto anteriormente, facilita la programación y el mantenimiento del código. A continuación,

veremos cómo se declaran tanto clases como interfaces, y cuál es el proceso para crear sus instancias.

Una clase es una agrupación de datos (variables o campos) y de funciones (métodos) que operan sobre esos datos.

Todos los métodos y variables deben ser definidos dentro del bloque {...} de la clase.

Un objeto (en inglés instance) es un ejemplar concreto de una clase. Las clases son como tipos de variables, mientras que los objetos son como variables concretas de un tipo determinado.

NombreDeLaClase unObjeto;

NombreDeLaClase otroObjeto;

4. Una clase sólo puede heredar de una única clase (en Java no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de Object.La

clase Object es la base de toda la jerarquía de clases de Java.

5. En un archivo de código fuente se pueden definir varias clases, pero en un mismo archivo no puede haber más que una clase definida como public. Este archivo se debe llamar como la clase public que debe tener extensión .java. Con algunas excepciones, lo habitual es escribir una sola clase por archivo.

6. Si una clase contenida en un fichero no es public, no es necesario que el fichero se llame como la clase.

7. Los métodos y variables de una clase pueden referirse de modo global a un objeto de esa clase a la que se aplican por medio de la referencia this.Al utilizar la palabra reservada 'this' para referirse tanto a métodos como atributos se restringe el ámbito al objeto que hace la declaración.

8. Las clases se pueden agrupar en packages que significa paquetes, introduciendo una línea al comienzo del fichero (package packageName;). Esta agrupación en packages está relacionada con la jerarquía de carpetas y archivos en la que se guardan las clases. En la práctica usamos paquetes para agrupar clases con un mismo propósito usando jerarquía de paquetes; esta decisión es muy importante a la hora de diseñar la estructura de nuestro programa.

En las clases:

1. Todas las variables y métodos de Java deben pertenecer a una clase. No hay variables y funciones globales.

2. Si una clase deriva de otra (extends), hereda todas sus variables y métodos.

3. Java tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los usuarios. Es decir que toda clase definida por el programador es heredada de la clase Object definida por el lenguaje de programación.

Los nombres de los packages se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan con mayúscula. El nombre de un package puede constar de varios nombres unidos por puntos (los propios packages de Java siguen esta norma, como por ejemplo java.awt.event)

. Es recomendable que los nombres de las clases de Java sean únicos, es el nombre del package lo que permite obtener esta característica.

En un programa de Java, una clase puede ser referida con su nombre completo (el nombre del package más el de la clase, separados por un punto). También se pueden referir con el nombre completo las variables y los métodos de las clases. Esto se puede hacer so y portable nuestra codificación.

Con import permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre del package importado.

Se importan por defecto el package java.lang y el package actual o por defecto (las clases del directorio actual).

Existen dos formas de utilizar import: para una clase y para todo un package:

import poo.cine.Actor;

import poo.cine.*;

El importar un package no hace que se carguen todas las clases del package: sólo se cargarán las clases public que se vayan a utilizar. Al importar un package no se importan los sub-packages. Éstos deben ser importados explícitamente, pues en realidad son packages distintos.

Por ejemplo, al importar java.awt no se importa java.awt.event.

La anotación

@Override simplemente se utiliza, para forzar al compilador a comprobar en tiempo de compilación que estás sobrescribiendo correctamente un método, y de este modo evitar errores en tiempo de ejecución, los cuales serían mucho más difíciles de detectar.

Por ejemplo, si fueras a sobrescribir el método toString() de la clase Object y lo haces de este modo:

```
public class MiClase {

	public String ToString() {
		return "Hola, esta es MiClase";
	}
}
```

realmente no estás sobrescribiendo el método, sino creando uno nuevo, ya que el nombre correcto comienza con minúscula y no con mayúscula. Por lo tanto, si luego en tu programa intentas obtener el String correspondiente de una instancia de MiClase, intentarías hacer esto, por ejemplo:

```
System.out.println(instanciaMiClase.toString());
```

obteniendo una salida como esta: MiClase@55b7a4e0, ya que se está llamando al método de la clase padre (que en este caso es Object). Pero si por el contrario le agregas la anotación:

```
public class MiClase {

	@Override
	public String ToString() {
		return "Hola, esta es MiClase";
	}
}
```

el compilador te va a avisar que tienes un problema con un mensaje de error como este: "Method does not override method from its superclass".

Por eso es que existe dicha anotación, para poder detectar en tiempo de compilación que no estás cumpliendo con los requisitos para sobrescribir un método.

Recomendación : usa esta anotación en los métodos que vayas a sobrescribir, evitará muchos problemas.

JAVA EXCEPCIONES

Repaso JAVA EXCEPCIONES

// Conceptos básicos

Una breve definición corta de una excepción podemos decir que es una excepción es un evento sí que se produce cuando se ejecuta el programa de forma que se interrumpe el flujo normal de instrucciones.

En el gráfico se observa las letras del flujo de un código fuente paralelamente que tiene programada una excepción y en el segundo el donde que está con el rato de error, es un código fuente que no tiene programado una excepción.

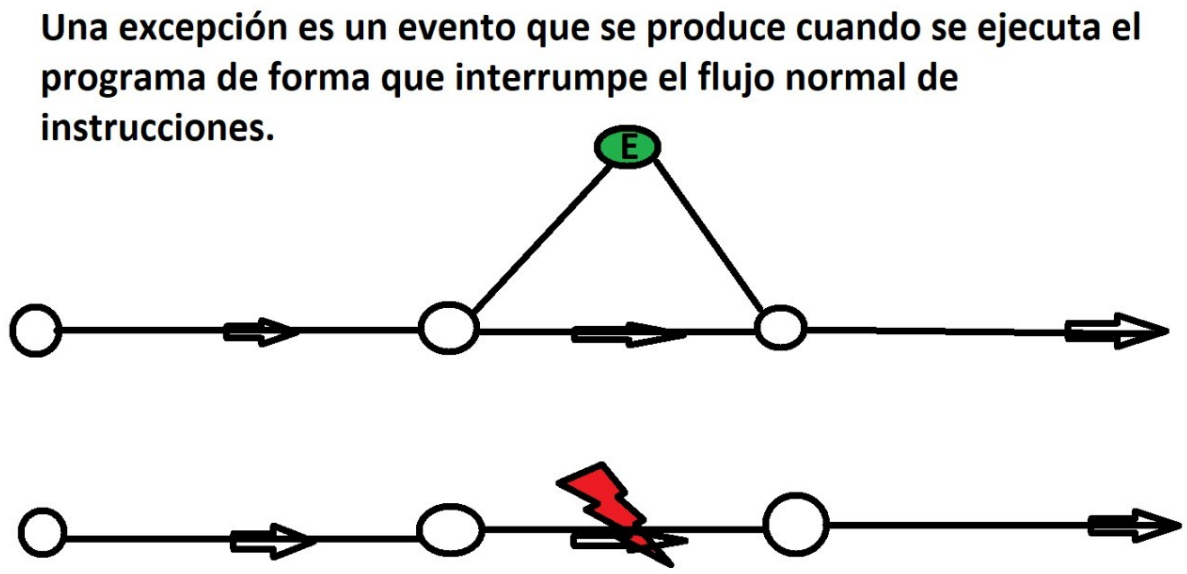
Si hay una interrupción del flujo normal genera el error.

Ejemplo

Una operación muy simple y muy usada es la división en cero no es un valor correcto, en una calculadora en el visor de la misma en vez de trabajar en infinito, lanza un mensaje de un error, con este mensaje es suficiente para decirte eso que no puedes dividir cero.

Distinto es sin no hay una excepción definida en el código , java tiene niveles y una librería muy extensa para modelar diferentes excepciones.

Fundamentales en lo que veremos en la unidades de pruebas unitarias.



Concluyendo: Una excepción es un error que ocurre en tiempo de ejecución. Utilizando el subsistema de manejo de excepciones de Java, puede, de una manera estructurada y controlada, manejar los errores de tiempo de ejecución.
Aunque la mayoría de los lenguajes de programación modernos ofrecen algún tipo de manejo de excepciones, el soporte de Java es fácil de usar y flexible.

Hay dos subclases directas de **Throwable**: **Exception** y **Error**:

1.Las excepciones de tipo Error :Están relacionadas con errores que ocurren en la Máquina Virtual de Java y no en tu programa. Este tipo de excepciones escapan a su control y, por lo general, tu programa no se ocupará de ellas. Por lo tanto, este tipo de excepciones no se describen aquí.

2.Los errores que resultan de la actividad del programa están representados por subclases de Exception. Por ejemplo, dividir por cero, límite de matriz y errores de archivo caen en esta categoría. En general, tu programa debe manejar excepciones de estos tipos. Una subclase importante de Exception es RuntimeException, que se usa para representar varios tipos comunes de errores en tiempo de ejecución.

Fundamentos de manejo de excepciones
El manejo de excepciones Java se gestiona a través de cinco palabras clave: **try**, **catch**, **throw**, **throws**,y **finally**. Forman un subsistema interrelacionado en el que el uso de uno implica el uso de otro. A lo largo de este curso, cada palabra clave se examina en detalle. Sin embargo, es útil desde el principio tener una comprensión general del papel que cada uno desempeña en el manejo de excepciones. En resumen, así es como funcionan.
Las declaraciones de programa que desea supervisar para excepciones están contenidas dentro de un bloque **try**. Si se produce una excepción dentro del bloque try, se lanza. Tu código puede atrapar esta excepción usando **catch**y manejarlo de una manera racional. Las excepciones generadas por el sistema son lanzadas automáticamente por el sistema de tiempo de ejecución de Java. Para lanzar manualmente una excepción, use la palabra

clavec**throw**. En algunos casos, una excepción arrojada por un método debe ser especificada como tal por una cláusula **throws**. Cualquier código que debe ejecutarse al salir de un bloque **try**se coloca en un bloquec**finally**.
3.cUso de try y catch
En el centro del manejo de excepciones están y . Estas palabras clave trabajan juntas; no puedes atrapar (**catch**) sin intentarlo (**try**). Aquí está la forma general de los bloques de manejo de excepciones try/catch:

```
try{
 //bloque de código para monitorear errores
}
catch (TipoExcepcion1 exOb){
  //Manejador para TipoExepción1
 }
catch (TipoExcepcion2 exOb){
 //Manejador para TipoExepción2
 }
```

Aquí, TipoExcepcion es el tipo de excepción que ha ocurrido. Cuando se lanza una excepción, es atrapada por su instrucción **catch** correspondiente, que luego procesa la excepción. Como muestra la forma general, puede haber más de una declaración **catch** asociada con un **try**. El tipo de la excepción determina qué declaración de captura se ejecuta. Es decir, si el tipo de excepción especificado por una instrucción **catch** coincide con el de la excepción, entonces se ejecuta esa instrucción de **catch** (y todos los demás se anulan). Cuando se detecta una excepción, exOb recibirá su valor.

✕Si no se lanza una excepción, entonces un bloque **try** finaliza normalmente, y todas sus declaraciones **catch** se pasan por alto. La ejecución se reanuda con la primera instrucción después del último **catch**. Por lo tanto, las declaraciones catch se ejecutan solo si se lanza una excepción.

Un ejemplo de excepción simple
Aquí hay un ejemplo simple que ilustra cómo observar y atrapar una excepción. Como saben, es un error intentar indexar una matriz más allá de sus límites. Cuando esto ocurre, la JVM lanza una **ArrayIndexOutOfBoundsException**. El siguiente programa genera a propósito tal excepción y luego la atrapa:

```
public class ExcDemo {
    public static void main(String[] args) {
        int nums[]=new int;
```

```
        try {
            System.out.println("Antes de que se genere la excepción.");
            //generar una excepción de índice fuera de límites
            nums=10;
        }catch (ArrayIndexOutOfBoundsException exc){
            //Capturando la excepción
            System.out.println("Índice fuera de los límites!");
        }
        System.out.println("Después de que se genere la excepción.");
    }
}
```

Salida:

```
Antes de que se genere la excepción.
Índice fuera de los límites!
Después de que se genere la excepción.
```

Aunque es bastante breve, el programa anterior ilustra varios puntos clave sobre el manejo de excepciones:

- Primero, el código que desea monitorear para detectar errores está dentro de un bloque **try**.
- En segundo lugar, cuando se produce una excepción (en este caso, debido al intento de indexar nums más allá de sus límites), la excepción se emite desde el bloque **try** y es atrapada por la instrucción **catch**. En este punto, el control pasa al catch, y el bloque try finaliza.
- Es decir, no se llama a catch. Por el contrario, la ejecución del programa se transfiere a él. Por lo tanto, la instrucción que sigue a nunca se ejecutará.
- Después de que se ejecuta la instrucción **catch**, el control del programa continúa con las declaraciones que siguen el **catch**. Por lo tanto, es el trabajo de tu controlador de excepción remediar el problema que causó la excepción para que la ejecución del programa pueda continuar normalmente.

Recuerde, si no se lanza una excepción por un bloque **try**, no se ejecutarán declaraciones **catch** y el control del programa se reanudará después de la instrucción catch. Para confirmar esto, en el programa anterior, cambie la línea

```
nums = 10;
```

por

```
nums = 10;
```

Ahora, no se genera ninguna excepción, y el bloque catch no se ejecuta.

3.2. Un ejemplo de excepción con método

Es importante comprender que todo código dentro de un bloque **try** se supervisa para detectar excepciones. Esto incluye excepciones que pueden ser generadas por un método llamado desde dentro del bloque try.

Una excepción lanzada por un método llamado desde dentro de un bloque try puede ser atrapada por las declaraciones catch asociadas con ese bloque try, asumiendo, por supuesto, que el método no captó la excepción en sí misma. Por ejemplo, este es un programa válido:

```
// Una excepción puede ser generada por un método

// y atrapada por otro

public class ExcEjemplo {

//Generando una exepción

    static void genExcepcion(){

        int nums[]= new int;


        System.out.println("Antes de que se genere la excepción.");


        //generar una excepción de índice fuera de límites

        nums=10;

        System.out.println("Esto no se mostrará.");

    }

}

public class ExcDemo {

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

```
    int nums[]=new int;


    try {
      ExcEjemplo.genExcepcion();
    }catch (ArrayIndexOutOfBoundsException exc){
            //Capturando la excepción
            System.out.println("Índice fuera de los límites!");
        }
        System.out.println("Después de que se genere la excepción.");
    }
}
```

Salida:

```
Antes de que se genere la excepción.

Índice fuera de los límites!

Después de que se genere la excepción.
```

Como se llama a genExcepcion() desde un bloque try, la excepción que genera es capturada por catch en main(). Entender, sin embargo, que si genExcepcion() había atrapado la excepción en sí misma, nunca se hubiera pasado a main().

3.3. Captura de excepciones de subclase

Hay un punto importante sobre declaraciones de múltiples **catch** que se relaciona con subclases. Una cláusula catch para una superclase también coincidirá con cualquiera de sus subclases.

Por ejemplo, dado que la superclase de todas las excepciones es Throwable, para atrapar todas las excepciones posibles, capture **Throwable**. Si desea capturar excepciones de un tipo de superclase y un tipo de subclase, coloque la subclase primero en la secuencia de **catch**. Si no lo hace, la captura de la superclase también atrapará todas las clases derivadas. Esta regla se autoejecuta porque poner primero la superclase hace que se cree un código inalcanzable, ya que la cláusula catch de la subclase nunca se puede ejecutar. En Java, el código inalcanzable es un error.

Por ejemplo, considere el siguiente programa:

```
// Las subclases deben preceder a las superclases

// en las declaraciones catch
```

```
public class ExcDemo {
    public static void main(String[] args) {


        //Aquí, num es más grande que denom
        int nums[]={4,8,16,32,64,128,256,512};
        int denom[]={2,0,4,4,0,8};


        for (int i=0;i< nums.length;i++){
            try {
                System.out.println(nums+" / "+
                                   denom+" es "+nums/denom);;
            }catch (ArrayIndexOutOfBoundsException exc){
                //Capturando la excepción (subclase)
                System.out.println("No se encontró ningún elemento.");
            }
            catch (Throwable exc){
                //Capturando la excepción (superclase)
            System.out.println("Alguna excepción ocurrió.");
        }
        }
    }
}
```

Salida:

```
4 / 2 es 2
Alguna excepción ocurrió.
16 / 4 es 4
```

```
32 / 4 es 8

Alguna excepción ocurrió.

128 / 8 es 16

No se encontró ningún elemento.

No se encontró ningún elemento.
```

En este caso, catch (Throwable) detecta todas las excepciones excepto ArrayIndexOutOfBoundsException. El problema de detectar excepciones de subclase se vuelve más importante cuando crea excepciones propias.
4. Los bloques try pueden ser anidados
Un bloque **try** se puede anidar dentro de otro. Una excepción generada dentro del bloque try interno que no está atrapada por un **catch** asociado con este **try**, se propaga al bloque try externo. Por ejemplo, aquí la ArrayIndexOutOfBoundsException no es capturada por el **catch** interno, sino por el **catch** externo:

```
// Uso de un bloque try anidado

public class TryAnidado{

    public static void main(String[] args) {

        //Aquí, num es más grande que denom

        int nums[]={4,8,16,32,64,128,256,512};

        int denom[]={2,0,4,4,0,8};


        try { //try externo

            for (int i = 0; i < nums.length; i++) {

                try { //try anidado

                    System.out.println(nums + " / " +

                            denom + " es " + nums / denom);

                } catch (ArithmeticException exc) {

                    //Capturando la excepción

                    System.out.println("No se puede dividir por cero!");

                            }
```

```
            }
        }
        catch (ArrayIndexOutOfBoundsException exc) {
                    //Capturando la excepción
                    System.out.println("Alguna excepción ocurrió.");
                    System.out.println("ERROR: Programa terminado.");
                }
            }
        }
```

Salida:

```
4 / 2 es 2
No se puede dividir por cero!
16 / 4 es 4
32 / 4 es 8
No se puede dividir por cero!
128 / 8 es 16
Alguna excepción ocurrió.
ERROR: Programa terminado.
```

En este ejemplo, una excepción que puede ser manejada por el try interno, en este caso, un error de división por cero, permite que el programa continúe. Sin embargo, un error de límite de matriz es capturado por la try externo, lo que hace que el programa finalice.
Aunque ciertamente no es la única razón para las instrucciones try anidadas, el programa anterior hace un punto importante que se puede generalizar. A menudo,l**os bloques try anidados se usan para permitir que las diferentes categorías de errores se manejen de diferentes maneras**. Algunos tipos de errores son catastróficos y no se pueden solucionar. Algunos son menores y pueden manejarse de inmediato.
Puede utilizar un bloque try externo para detectar los errores más graves, permitiendo que los bloques try internos manejen los menos serios.

Lanzar una excepción

Los ejemplos anteriores han estado capturando excepciones generadas automáticamente por la JVM. Sin embargo, es posible lanzar manualmente una excepción utilizando la instrucción **throw**. Su forma general se muestra aquí:

```
throw excepcOb;
```

Aquí, excepcOb debe ser un objeto de una clase de excepción derivada de Throwable. Aquí hay un ejemplo que ilustra la instrucción arrojando manualmente una ArithmeticException:

```
//Lanzar manualmente una excepción

public class ThrowDemo {

    public static void main(String[] args) {

        try{

            System.out.println("Antes de lanzar excepción.");

            throw new ArithmeticException(); //Lanzar una excepción

        }catch (ArithmeticException exc){

            //Capturando la excepción

            System.out.println("Excepción capturada.");

        }

        System.out.println("Después del bloque try/catch");

    }

}
```

Salida:

```
Antes de lanzar excepción.

Excepción capturada.

Después del bloque try/catch
```

Observe cómo se creó la ArithmeticException utilizando **new** en la instrucción **throw**. Recuerde, throw arroja un objeto. Por lo tanto, debe crear un objeto para “lanzar“. Es decir, no puedes simplemente lanzar un tipo.

Re-lanzar una excepción:

Una excepción capturada por una declaración **catch** se puede volver a lanzar para que pueda ser capturada por un **catch** externo. La razón más probable para volver a lanzar de esta manera es permitir el acceso de múltiples manejadores/controladores a la excepción.

Por ejemplo, quizás un manejador de excepciones maneja un aspecto de una excepción, y un segundo manejador se enfrenta a otro aspecto. Recuerde, cuando vuelve a lanzar una excepción, no se volverá a capturar por la misma declaración catch. Se propagará a la siguiente declaración de catch. El siguiente programa ilustra el relanzamiento de una excepción:

```
//Relanzando un excepción

public class Rethrow {

    public static void genExcepcion() {

        //Aquí, num es más largo que denom

        int nums[] = {4, 8, 16, 32, 64, 128, 256, 512};

        int denom[] = {2, 0, 4, 4, 0, 8};


        for (int i = 0; i < nums.length; i++) {

            try {

                System.out.println(nums + " / " +

                        denom + " es " + nums / denom);

            } catch (ArithmeticException exc){

                //Capturando la excepción

                System.out.println("No se puede dividir por cero!.");

            }

            catch (ArrayIndexOutOfBoundsException exc) {

                //Capturando la excepción

                System.out.println("No se encontró ningún elemento.");

                throw exc; //Relanzando la excepción

            }

        }

    }

}
```

```
public class RethrowDemo {
    public static void main(String[] args) {
        try{
            Rethrow.genExcepcion();
        }
        catch (ArrayIndexOutOfBoundsException exc){
            //Recapturando la excepción
            System.out.println("ERROR - Programa terminado");
        }
    }
}
```

Salida:

```
4 / 2 es 2
No se puede dividir por cero!.
16 / 4 es 4
32 / 4 es 8
No se puede dividir por cero!.
128 / 8 es 16
No se encontró ningún elemento.
ERROR - Programa terminado
```

En este programa, los errores de división por cero se manejan localmente, mediante genExcepcion(), pero se vuelve a generar un error de límite de matriz. En este caso, es capturado por main().

7. Una mirada más cercana a Throwable

Con cláusula **catch** especifica un tipo de excepción y un parámetro. El parámetro recibe el objeto de excepción. Como todas las excepciones son subclases de Throwable, todas las excepciones admiten los métodos definidos por Throwable.

Tabla de métodos Throwable.

Método	Sintaxis	Descripción
getMessage	String getMessage()	Devuelve una descripción de la excepción.
getLocalizedMessage	String getLocalizedMessage()	Devuelve una descripción localizada de la excepción.
toString	String toString()	Devuelve un objeto String que contiene una descripción completa de la excepción. Este método lo llama println() cuando se imprime un objeto Throwable.
printStackTrace()	void printStackTrace()	Muestra el flujo de error estándar.
printStackTrace	void printStackTrace(PrintStream s)	Envía la traza de errores a la secuencia especificada.
printStackTrace	void printStackTrace(PrintWriter s)	Envía la traza de errores a la secuencia especificada.
fillInStackTrace	Throwable fillInStackTrace()	Devuelve un objeto Throwable que contiene un seguimiento de pila completo. Este objeto se puede volver a lanzar.

De los métodos definidos por Throwable, dos de los más interesantes son printStackTrace() y toString().

- Puede visualizar el mensaje de error estándar más un registro de las llamadas a métodos que conducen a la excepción llamando a **printStackTrace()**.
- Puede usar **toString()** para recuperar el mensaje de error estándar. El método toString() también se invoca cuando se usa una excepción como argumento para println().

El siguiente programa demuestra estos métodos:

```
public class ExcDemo {

   static void genExcepcion(){

      int nums[]=new int;
```

```
        System.out.println("Antes de lanzar excepción.");


        nums=10;
        System.out.println("Esto no se mostrará.");
    }
}


class MetodosThrowable{
    public static void main(String[] args) {
        try{
            ExcDemo.genExcepcion();
        }
        catch (ArrayIndexOutOfBoundsException exc){
            System.out.println("Mensaje estándar: ");
            System.out.println(exc);
            System.out.println("\nTraza de errores: ");
            exc.printStackTrace();
        }
        System.out.println("Después del bloque catch.");
    }
}
```

Salida:

```
Antes de lanzar excepción.
Mensaje estándar:
java.lang.ArrayIndexOutOfBoundsException: 7
```

```
Traza de errores:

java.lang.ArrayIndexOutOfBoundsException: 7

at ExcDemo.genExcepcion(ExcDemo.java:8)

at MetodosThrowable.main(ExcDemo.java:16)

Después del bloque catch.
```

8. Uso de finally
Algunas veces querrá definir un bloque de código que se ejecutará cuando quede un bloque try/catch. Por ejemplo, una excepción puede causar un error que finaliza el método actual, causando su devolución prematura. Sin embargo, ese método puede haber abierto un archivo o una conexión de red que debe cerrarse.
Tales tipos de circunstancias son comunes en la programación, y Java proporciona una forma conveniente de manejarlos:
finally.
Para especificar un bloque de código a ejecutar cuando se sale de un bloque try/catch, incluya un bloque al final de una secuencia **try/catch**. Aquí se muestra la forma general de un try/catch que incluye **finally**.

```
try{

//bloque de código para monitorear errores

}

catch(TipoExcepcion1 exOb){

//manejador para TipoExcepcion1

}

catch(TipoExcepcion2 exOb){

//manejador para TipoExcepcion2

}

//...

finally{

//código final

}
```

El bloque finally se ejecutará siempre que la ejecución abandone un bloque **try/catch**, sin importar las condiciones que lo causen. Es decir, si el bloque try finaliza normalmente, o debido a una excepción, el último código ejecutado es el definido por finally. El bloque finally también se

ejecuta si algún código dentro del bloque try o cualquiera de sus declaraciones catch devuelve del método.
Aquí hay un ejemplo de finally:

```
//Uso de finally

public class UsoFinally {

    public static void genExcepecion(int rec) {

        int t;

        int nums[]=new int;


        System.out.println("Recibiendo "+rec);

        try {

            switch (rec){

                case 0:

                    t=10 /rec;

                    break;

                case 1:

                    nums=4; //Genera un error de indexación

                    break;

                case 2:

                    return; //Retorna desde el blorec try

            }

        }

        catch (ArithmeticException exc){

            //Capturando la excepción

            System.out.println("No se puede dividir por cero!");

            return; //retorna desde catch

        }

        catch (ArrayIndexOutOfBoundsException exc){

            //Capturando la excepción
```

```
            System.out.println("Elemento no encontrado");

        }

        finally {

            //esto se ejecuta al salir de los blorecs try/catch

            System.out.println("Saliendo de try.");

        }

    }

}
```

Salida:

```
class FinallyDemo{

    public static void main(String[] args) {

        for (int i=0;i<3;i++){

            UsoFinally.genExcepecion(i);

            System.out.println();

        }

    }

}
```

Como muestra la salida, no importa cómo se salga el bloque try, el bloque **finally** sí se ejecuta .
9. Uso de throws
En algunos casos, si un método genera una excepción que no maneja, debe declarar esa excepción en una cláusula . Aquí está la forma general de un método que incluye una cláusula **throws**:

```
tipo-retorno nombreMetodo(lista-param) throws lista-excepc {

// Cuerpo

}
```

Aquí, lista-excepc es una lista de excepciones separadas por comas que el método podría arrojar fuera de sí mismo.
Quizás se pregunte por qué no necesitó especificar una cláusula **throws** para algunos de los ejemplos anteriores, que arrojó excepciones fuera de los métodos. La respuesta es que las excepciones que son subclases de **Error** o **RuntimeException** no necesitan ser especificadas en una lista de **throws**. Java simplemente asume que un método puede

arrojar uno. Todos los otros tipos de excepciones deben ser declarados. De lo contrario, se produce un error en tiempo de compilación.
En realidad, usted vio un ejemplo de una cláusula **throws** anteriormente. Como recordará, al realizar la entrada del teclado, necesitaba agregar la cláusula a main(). Ahora puedes entender por qué. Una declaración de entrada podría generar una **IOException**, y en ese momento, no pudimos manejar esa excepción. Por lo tanto, tal excepción se descartaría de main() y necesitaría especificarse como tal. Ahora que conoce excepciones, puede manejar fácilmente IOException.

Ejemplo con throws

Un ejemplo que maneja **IOException**. Crea un método llamado prompt(), que muestra un mensaje de aviso y luego lee un carácter del teclado. Como la entrada se está realizando, puede ocurrir una IOException.
Sin embargo, el método prompt() no maneja IOException. En cambio, usa una cláusula throws, lo que significa que el método de llamada debe manejarlo. En este ejemplo, el método de llamada es main() y trata el error.

```
//Uso de throws

class ThrowsDemo {

    public static char prompt(String args)

        throws java.io.IOException {

            System.out.println(args + " :");

            return (char) System.in.read();

   }


        public static void main (String[]args){

            char ch;

            try {

                //dado que prompt() podría arrojar una excepción,

                // una llamada debe incluirse dentro de un bloque try

                ch = prompt("Ingresar una letra");

            } catch (java.io.IOException exc) {

                System.out.println("Ocurrió una excepción de E/S");

                ch = 'X';

            }
```

```
            System.out.println("Usted presionó: " + ch);

    }

}
```

En un punto relacionado, observe que IOException está totalmente calificado por su nombre de paquete java.io. Como aprenderá más adelante, el sistema de E/S de Java está contenido en el paquete **java.io**. Por lo tanto, **IOException** también está contenido allí. También habría sido posible importar java.io y luego referirme a IOException directamente.

Tres características adicionales de excepción

A partir de JDK 7, el mecanismo de **manejo de excepciones de Java** se ha ampliado con la adición de tres características.

1.**El primero es compatible con la gestión automática de recursos, que automatiza el proceso de liberación de un recurso, como un archivo, cuando ya no es necesario. Se basa en una forma expandida de , llamada declaración (try con recursos), y se describe más en Java Avanzado, cuando se discuten los archivos.**

2.**La segunda característica nueva se llama multi-catch.**

3.**Y la tercera a veces se llama final rethrow o more precise rethrow. Estas dos características se describen aquí.**

9.1. Característica multi-catch

El **multi-catch**permite capturar dos o más excepciones mediante la misma cláusula **catch.** Es posible (de hecho, común) que un intento sea seguido por dos o más cláusulas catch. Aunque cada cláusula catch a menudo proporciona su propia secuencia de código única, no es raro tener situaciones en las que dos o más cláusulas catch ejecutan la misma secuencia de código aunque atrapen diferentes excepciones.

En lugar de tener que capturar cada tipo de excepción individualmente, puede usar una única cláusula de catch para manejar las excepciones sin duplicación de código.

Para crear un **multi-catch**, especifique una lista de excepciones dentro de una sola cláusula **catch**. Para ello, separe cada tipo de excepción en la lista con el operador **OR**. Cada parámetro **multi-catch** es implícitamente **final**. (Puede especificar explícitamente final, si lo desea, pero no es necesario.) Debido a que cada parámetro **multi-catch** es implícitamente **final**, no se le puede asignar un nuevo valor.

Aquí se explica cómo puede usar la función **multi-catch** para capturar **ArithmeticException** y **ArrayIndexOutOfBoundsException** con una única cláusula catch:

```
catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
```

Aquí hay un programa simple que demuestra el uso de **multi-catch**:

```
// Uso de multi-catch

// Este código requiere JDK7 o superior
```

```
class MultiCatch {

    public static void main(String[] args) {

        int a=28, b=0;

        int resultado;

        char chars[]={'A','B','C'};


        for (int i=0; i<2;i++){

            try {

                if (i==0)

                    resultado=a/b; //genera un ArithmeticException

                else

                    chars ='X'; //genera un ArrayIndexOutOfBoundsException

            }catch (ArithmeticException | ArrayIndexOutOfBoundsException e){

                System.out.println("Excepción capturada: "+e);

            }

        }

        System.out.println("Después del multi-catch");

    }

}
```

Salida:

```
Excepción capturada: java.lang.ArithmeticException: / by zero

Excepción capturada: java.lang.ArrayIndexOutOfBoundsException: 5

Después del multi-catch
```

El programa generará una **ArithmeticException** cuando se intenta la división por cero. Generará una **ArrayIndexOutOfBoundsException** cuando se intente acceder fuera de los límites de chars. Ambas excepciones son capturadas por la declaración única de **catch**.
La función más precisa de **rethrow** restringe el tipo de excepciones que pueden volver a lanzarse solo a aquellas excepciones marcadas que arroja el bloque **try** asociado, que no son manejadas por una cláusula **catch** anterior, y que son un subtipo o supertipo del parámetro. Si bien esta capacidad puede no ser necesaria a menudo, ahora está disponible para su uso.

Para que la característica **rethrow** esté en vigor, el parámetro **catch** debe ser efectivamente **final**. Esto significa que no se le debe asignar un nuevo valor dentro del bloque **catch**. También se puede especificar explícitamente como **final**, pero esto no es necesario.

Patrones de diseño

Repaso Patrones de diseño

// Conceptos básicos

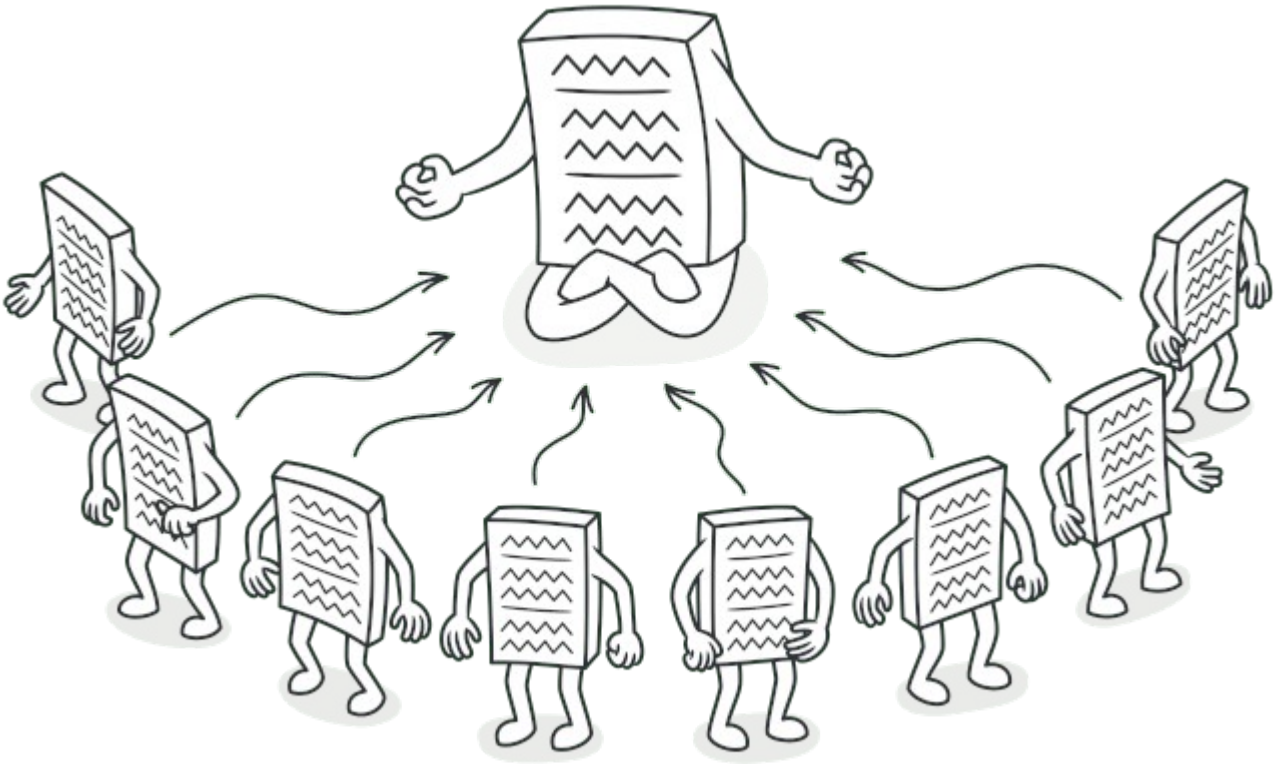
Singleton

También llamado: Instancia única

Propósito

Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.La clase **Singleton** declara el método estático `obtenerInstancia` que devuelve la misma instancia de su propia clase.

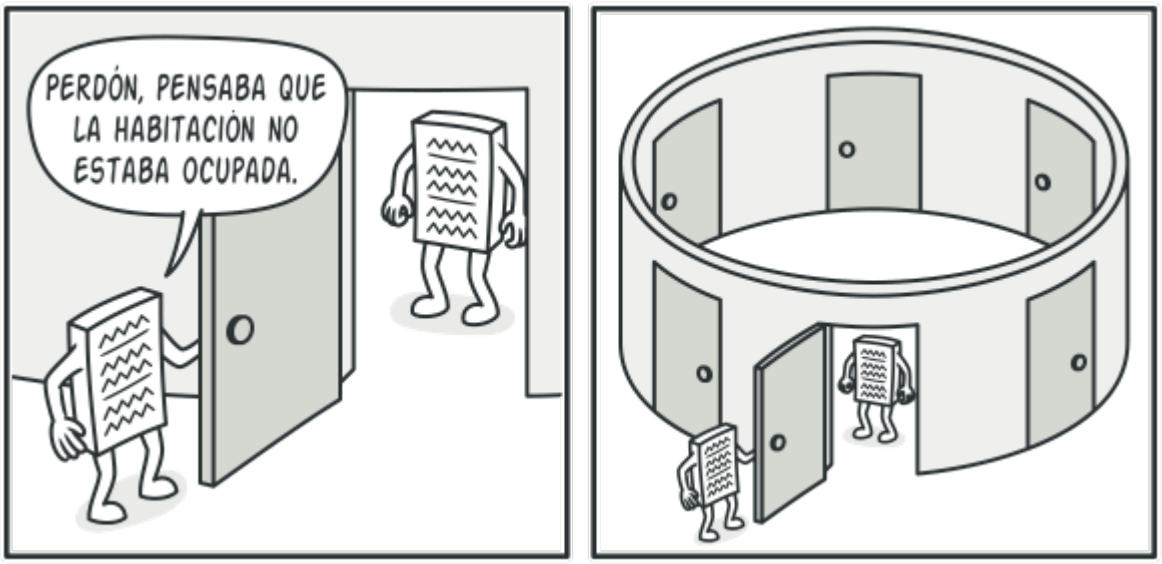
El constructor del Singleton debe ocultarse del código cliente. La llamada al método `obtenerInstancia` debe ser la única manera de obtener el objeto de Singleton.



Problema

El patrón Singleton resuelve dos problemas al mismo tiempo, vulnerando el Principio de responsabilidad única:

1. **Garantizar que una clase tenga una única instancia**. ¿Por qué querría alguien controlar cuántas instancias tiene una clase? El motivo más habitual es controlar el acceso a algún recurso compartido, por ejemplo, una base de datos o un archivo.
   Funciona así: en principio has creado un objeto y al cabo de un tiempo decides crear otro nuevo. En lugar de recibir un objeto nuevo, obtendrás el que ya habías creado.
   Ten en cuenta que este comportamiento es imposible de implementar con un constructor normal, ya que una llamada al constructor siempre **debe** devolver un nuevo objeto por diseño.



   Puede ser que los clientes ni siquiera se den cuenta de que trabajan con el mismo objeto todo el tiempo.
2. **Proporcionar un punto de acceso global a dicha instancia**. ¿Recuerdas esas variables globales que utilizaste (bueno, sí, fui yo) para almacenar objetos esenciales? Aunque son muy útiles, también son poco seguras, ya que cualquier código podría sobrescribir el contenido de esas variables y descomponer la aplicación.
   Al igual que una variable global, el patrón Singleton nos permite acceder a un objeto desde cualquier parte del programa. No obstante, también evita que otro código sobreescriba esa instancia.
   Este problema tiene otra cara: no queremos que el código que resuelve el primer problema se encuentre disperso por todo el programa. Es mucho más conveniente tenerlo dentro de una clase, sobre todo si el resto del código ya depende de ella.

Hoy en día el patrón Singleton se ha popularizado tanto que la gente suele llamar singleton a cualquier patrón, incluso si solo resuelve uno de los problemas antes mencionados.

Solución

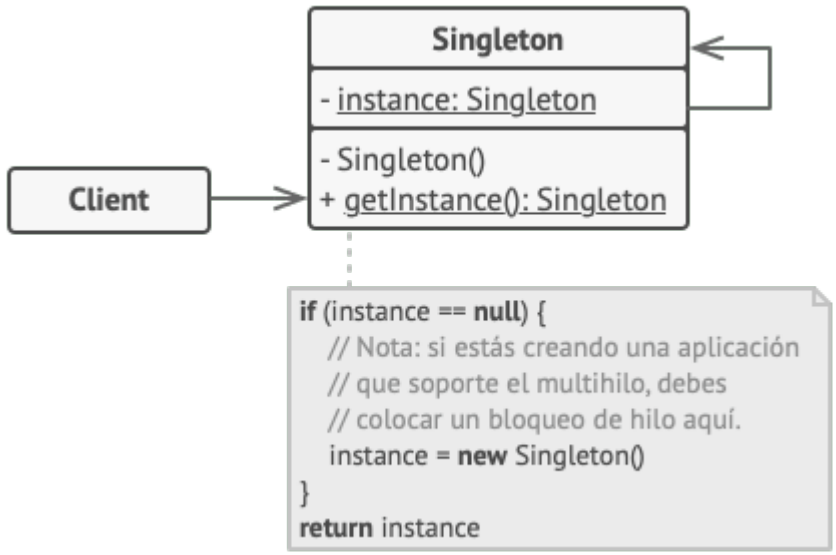
Todas las implementaciones del patrón Singleton tienen estos dos pasos en común:

- Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador `new` con la clase Singleton.
- Crear un método de creación estático que actúe como constructor. Tras bambalinas, este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.
  Si tu código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto.

Analogía en el mundo real

El gobierno es un ejemplo excelente del patrón Singleton. Un país sólo puede tener un gobierno oficial. Independientemente de las identidades personales de los individuos que forman el gobierno, el título “Gobierno de X” es un punto de acceso global que identifica al grupo de personas a cargo.

Estructura



1.

Pseudocódigo

En este ejemplo, la clase de conexión de la base de datos actúa como **Singleton**. Esta clase no tiene un constructor público, por lo que la única manera de obtener su objeto es invocando el método `obtenerInstancia`. Este método almacena en caché el primer objeto creado y lo devuelve en todas las llamadas siguientes.

```
// La clase Base de datos define el método `obtenerInstancia`
// que permite a los clientes acceder a la misma instancia de
// una conexión de la base de datos a través del programa.
class Database is
    // El campo para almacenar la instancia singleton debe
    // declararse estático.
    private static field instance: Database

    // El constructor del singleton siempre debe ser privado
    // para evitar llamadas de construcción directas con el
    // operador `new`.
    private constructor Database() is
        // Algún código de inicialización, como la propia
        // conexión al servidor de una base de datos.
        // ...

    // El método estático que controla el acceso a la instancia
    // singleton.
    public static method getInstance() is
        if (Database.instance == null) then
            acquireThreadLock() and then
                // Garantiza que la instancia aún no se ha
                // inicializado por otro hilo mientras ésta ha
                // estado esperando el desbloqueo.
                if (Database.instance == null) then
```

```
            Database.instance = new Database()
        return Database.instance


    // Por último, cualquier singleton debe definir cierta
    // lógica de negocio que pueda ejecutarse en su instancia.
    public method query(sql) is
        // Por ejemplo, todas las consultas a la base de datos
        // de una aplicación pasan por este método. Por lo
        // tanto, aquí puedes colocar lógica de regularización
        // (throttling) o de envío a la memoria caché.
        // ...


class Application is
    method main() is
        Database foo = Database.getInstance()
        foo.query("SELECT ...")
        // ...
        Database bar = Database.getInstance()
        bar.query("SELECT ...")
        // La variable `bar` contendrá el mismo objeto que la
        // variable `foo`.
```

Aplicabilidad

Utiliza el patrón Singleton cuando una clase de tu programa tan solo deba tener una instancia disponible para todos los clientes; por ejemplo, un único objeto de base de datos compartido por distintas partes del programa.
El patrón Singleton deshabilita el resto de las maneras de crear objetos de una clase, excepto el método especial de creación. Este método crea un nuevo objeto, o bien devuelve uno existente si ya ha sido creado.
Utiliza el patrón Singleton cuando necesites un control más estricto de las variables globales.
Al contrario que las variables globales, el patrón Singleton garantiza que haya una única instancia de una clase. A excepción de la propia clase Singleton, nada puede sustituir la instancia en caché.

Ten en cuenta que siempre podrás ajustar esta limitación y permitir la creación de cierto número de instancias Singleton. La única parte del código que requiere cambios es el cuerpo del método `getInstance`.

Cómo implementarlo

- Añade un campo estático privado a la clase para almacenar la instancia Singleton.
- Declara un método de creación estático público para obtener la instancia Singleton.
- Implementa una inicialización diferida dentro del método estático. Debe crear un nuevo objeto en su primera llamada y colocarlo dentro del campo estático. El método deberá devolver siempre esa instancia en todas las llamadas siguientes.
- Declara el constructor de clase como privado. El método estático de la clase seguirá siendo capaz de invocar al constructor, pero no a los otros objetos.
- Repasa el código cliente y sustituye todas las llamadas directas al constructor de la instancia Singleton por llamadas a su método de creación estático.

Pros y contras

- Puedes tener la certeza de que una clase tiene una única instancia.
- Obtienes un punto de acceso global a dicha instancia.
- El objeto Singleton solo se inicializa cuando se requiere por primera vez.
- Vulnera el Principio de responsabilidad única. El patrón resuelve dos problemas al mismo tiempo.
- El patrón Singleton puede enmascarar un mal diseño, por ejemplo, cuando los componentes del programa saben demasiado los unos sobre los otros.
- El patrón requiere de un tratamiento especial en un entorno con múltiples hilos de ejecución, para que varios hilos no creen un objeto Singleton varias veces.
- Puede resultar complicado realizar la prueba unitaria del código cliente del Singleton porque muchos frameworks de prueba dependen de la herencia a la hora de crear objetos simulados (mock objects). Debido a que la clase Singleton es privada y en la mayoría de los lenguajes resulta imposible sobrescribir métodos estáticos, tendrás que pensar en una manera original de simular el Singleton. O, simplemente, no escribas las pruebas. O no utilices el patrón Singleton.

Relaciones con otros patrones

- Una clase **fachada** a menudo puede transformarse en una **Singleton**, ya que un único objeto fachada es suficiente en la mayoría de los casos.
- **Flyweight** podría asemejarse a **Singleton** si de algún modo pudieras reducir todos los estados compartidos de los objetos a un único objeto flyweight. Pero existen dos diferencias fundamentales entre estos patrones:
    1. Solo debe haber una instancia Singleton, mientras que una clase Flyweight puede tener varias instancias con distintos estados intrínsecos.
    2. El objeto Singleton puede ser mutable. Los objetos flyweight son inmutables.
- Los patrones **Abstract Factory**, **Builder** y **Prototype** pueden todos ellos implementarse como **Singletons**.

Ejemplos de código

Singletones un patrón de diseño creacional que garantiza que tan solo exista un objeto de su tipo y proporciona un único punto de acceso a él para cualquier otro código.

El patrón tiene prácticamente los mismos pros y contras que las variables globales. Aunque son muy útiles, rompen la modularidad de tu código.

No se puede utilizar una clase que dependa del Singleton en otro contexto. Tendrás que llevar también la clase Singleton. La mayoría de las veces, esta limitación aparece durante la creación de pruebas de unidad.

Ejemplos de uso: Muchos desarrolladores consideran el patrón Singleton un antipatrón. Por este motivo, su uso está en declive en el código Java.
A pesar de ello, existen muchos ejemplos del patrón Singleton en las principales bibliotecas de Java:

- **java.lang.Runtime#getRuntime()**
- **java.awt.Desktop#getDesktop()**
- **java.lang.System#getSecurityManager()**

Identificación: El patrón Singleton se puede reconocer por un método de creación estático, que devuelve el mismo objeto guardado en caché.

Singleton ingenuo (hilo único)

Es muy fácil implementar un Singleton descuidado. Tan solo necesitas esconder el constructor e implementar un método de creación estático.

Singleton.java: Singleton

```
package refactoring_guru.singleton.example.non_thread_safe;


public final class Singleton {
    private static Singleton instance;
    public String value;


    private Singleton(String value) {
        // The following code emulates slow initialization.
        try {
            Thread.sleep(1000);
```

```
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        this.value = value;
    }

    public static Singleton getInstance(String value) {
        if (instance == null) {
            instance = new Singleton(value);
        }
        return instance;
    }
}
```

Factory method es un patrón de diseño creacional que resuelve el problema de crear objetos de producto sin especificar sus clases concretas.

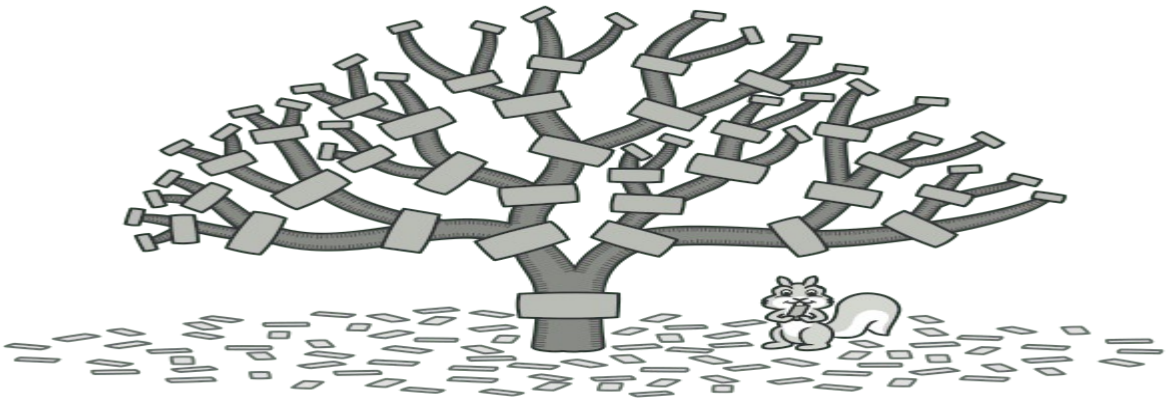
El patrón Factory Method define un método que debe utilizarse para crear objetos, en lugar de una llamada directa al constructor (operador `new`). Las subclases pueden sobrescribir este método para cambiar las clases de los objetos que se crearán.

Composite

También llamado: Objeto compuesto, Object Tree

Propósito

Composite es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

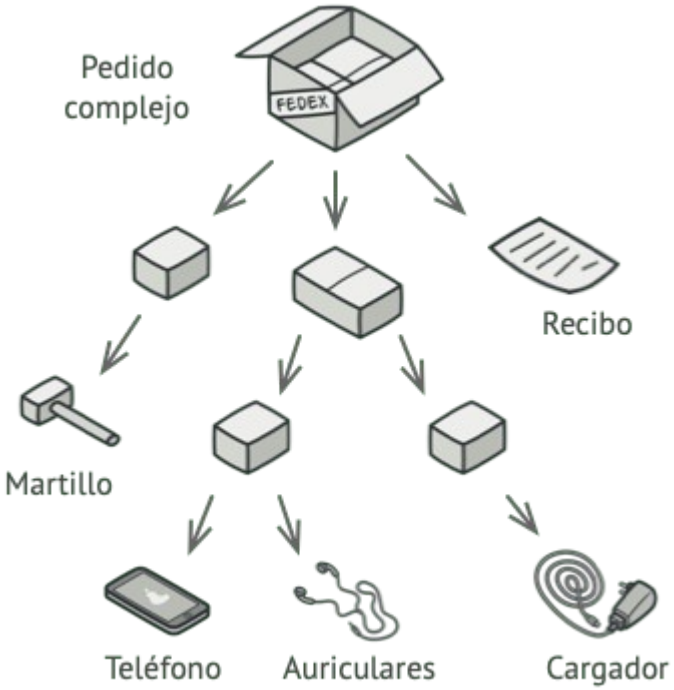


Problema

El uso del patrón Composite sólo tiene sentido cuando el modelo central de tu aplicación puede representarse en forma de árbol.

Por ejemplo, imagina que tienes dos tipos de objetos: `Productos` y `Cajas`.
Una `Caja` puede contener varios `Productos` así como cierto número de `Cajas` más pequeñas. Estas `Cajas` pequeñas también pueden contener algunos `Productos` o incluso `Cajas` más pequeñas, y así sucesivamente.

Digamos que decides crear un sistema de pedidos que utiliza estas clases. Los pedidos pueden contener productos sencillos sin envolver, así como cajas llenas de productos... y otras cajas. ¿Cómo determinarás el precio total de ese pedido?



Un pedido puede incluir varios productos empaquetados en cajas, que a su vez están empaquetados en cajas más grandes y así sucesivamente. La estructura se asemeja a un árbol boca abajo.

Puedes intentar la solución directa: desenvolver todas las cajas, repasar todos los productos y calcular el total. Esto sería viable en el mundo real; pero en un programa no es tan fácil como ejecutar un bucle. Tienes que conocer de antemano las clases de `Productos` y `Cajas` a iterar, el nivel de anidación de las cajas y otros detalles desagradables. Todo esto provoca que la solución directa sea demasiado complicada, o incluso imposible.

Solución

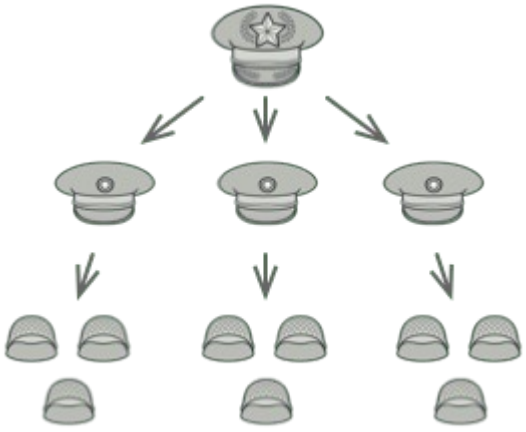
El patrón Composite sugiere que trabajes con `Productos` y `Cajas` a través de una interfaz común que declara un método para calcular el precio total.

¿Cómo funcionaría este método? Para un producto, sencillamente devuelve el precio del producto. Para una caja, recorre cada artículo que contiene la caja, pregunta su precio y devuelve un total por la caja. Si uno de esos artículos fuera una caja más pequeña, esa caja también comenzaría a repasar su contenido y así sucesivamente, hasta que se calcule el precio de todos los componentes internos. Una caja podría incluso añadir costos adicionales al precio final, como costos de empaquetado.

El patrón Composite te permite ejecutar un comportamiento de forma recursiva sobre todos los componentes de un árbol de objetos.

La gran ventaja de esta solución es que no tienes que preocuparte por las clases concretas de los objetos que componen el árbol. No tienes que saber si un objeto es un producto simple o una sofisticada caja. Puedes tratarlos a todos por igual a través de la interfaz común. Cuando invocas un método, los propios objetos pasan la solicitud a lo largo del árbol.

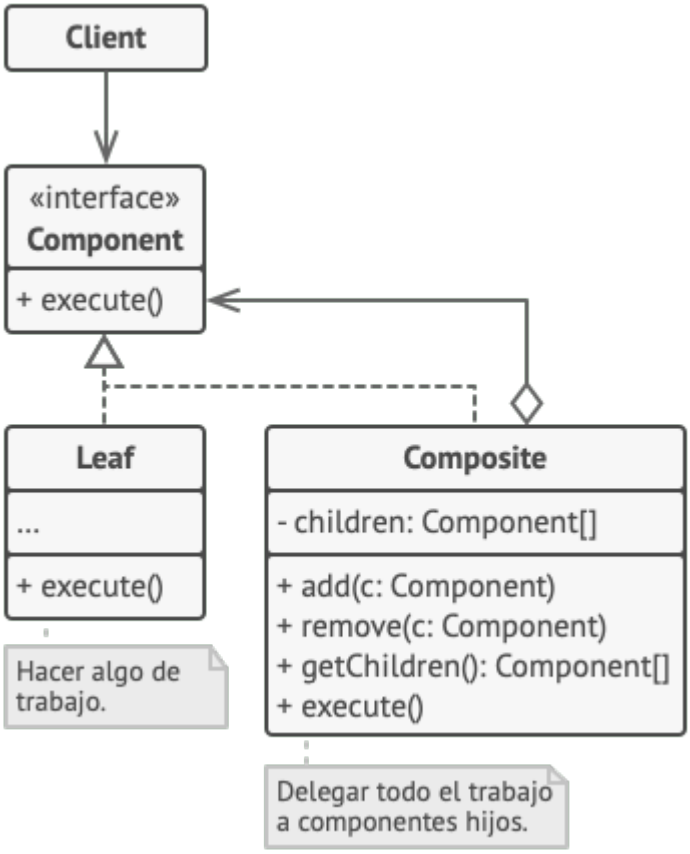
Analogía en el mundo real



Un ejemplo de estructura militar.

Los ejércitos de la mayoría de países se estructuran como jerarquías. Un ejército está formado por varias divisiones; una división es un grupo de brigadas y una brigada está formada por pelotones, que pueden dividirse en escuadrones. Por último, un escuadrón es un pequeño grupo de soldados reales. Las órdenes se dan en la parte superior de la jerarquía y se pasan hacia abajo por cada nivel hasta que todos los soldados saben lo que hay que hacer.

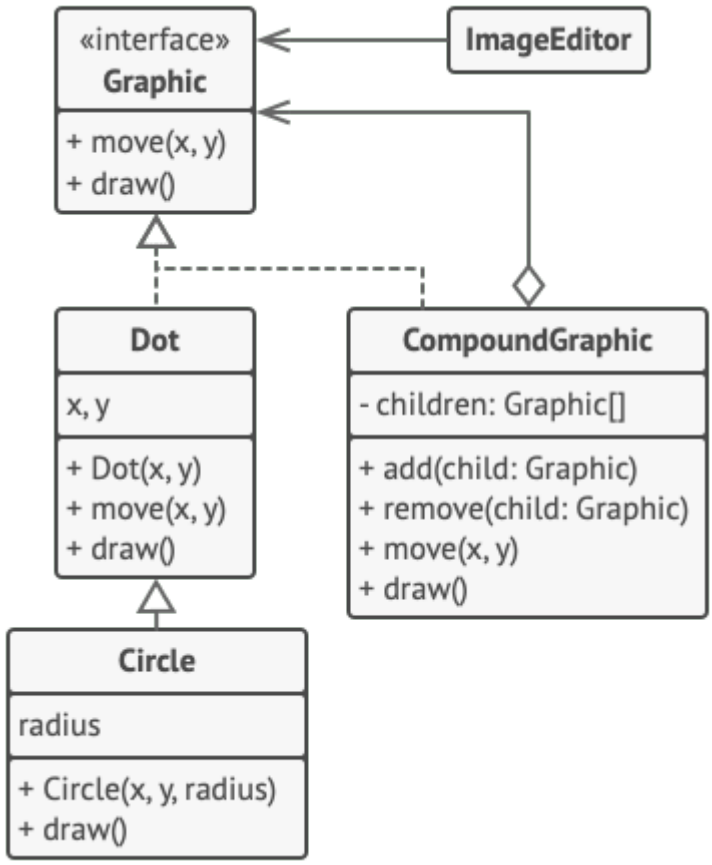
Estructura



1.

Pseudocódigo

En este ejemplo, el patrón **Composite** te permite implementar el apilamiento (stacking) de formas geométricas en un editor gráfico.



Ejemplo del editor de formas geométricas.

La clase `GráficoCompuesto` es un contenedor que puede incluir cualquier cantidad de subformas, incluyendo otras formas compuestas. Una forma compuesta tiene los mismos métodos que una forma simple. Sin embargo, en lugar de hacer algo por su cuenta, una forma compuesta pasa la solicitud de forma recursiva a todos sus hijos y “suma” el resultado.

El código cliente trabaja con todas las formas a través de la interfaz común a todas las clases de forma. De este modo, el cliente no sabe si está trabajando con una forma simple o una compuesta. El cliente puede trabajar con estructuras de objetos muy complejas sin acoplarse a las clases concretas que forman esa estructura.

```
// La interfaz componente declara operaciones comunes para

// objetos simples y complejos de una composición.

interface Graphic is

    method move(x, y)

    method draw()


// La clase hoja representa objetos finales de una composición.
```

```
// Un objeto hoja no puede tener ningún subobjeto. Normalmente,
// son los objetos hoja los que hacen el trabajo real, mientras
// que los objetos compuestos se limitan a delegar a sus
// subcomponentes.
class Dot implements Graphic is
    field x, y

    constructor Dot(x, y) { ... }

    method move(x, y) is
        this.x += x, this.y += y

    method draw() is
        // Dibuja un punto en X e Y.

// Todas las clases de componente pueden extender otros
// componentes.
class Circle extends Dot is
    field radius

    constructor Circle(x, y, radius) { ... }

    method draw() is
        // Dibuja un círculo en X y Y con radio R.

// La clase compuesta representa componentes complejos que
// pueden tener hijos. Normalmente los objetos compuestos
// delegan el trabajo real a sus hijos y después "recapitulan"
// el resultado.
class CompoundGraphic implements Graphic is
```

```
    field children: array of Graphic

    // Un objeto compuesto puede añadir o eliminar otros
    // componentes (tanto simples como complejos) a o desde su
    // lista hija.
    method add(child: Graphic) is
        // Añade un hijo a la matriz de hijos.

    method remove(child: Graphic) is
        // Elimina un hijo de la matriz de hijos.

    method move(x, y) is
        foreach (child in children) do
            child.move(x, y)

    // Un compuesto ejecuta su lógica primaria de una forma
    // particular. Recorre recursivamente todos sus hijos,
    // recopilando y recapitulando sus resultados. Debido a que
    // los hijos del compuesto pasan esas llamadas a sus propios
    // hijos y así sucesivamente, se recorre todo el árbol de
    // objetos como resultado.
    method draw() is
        // 1. Para cada componente hijo:
        //     - Dibuja el componente.
      //     - Actualiza el rectángulo delimitador.
                // 2. Dibuja un rectángulo de línea punteada utilizando
        // las coordenadas de delimitación.



// El código cliente trabaja con todos los componentes a través
```

```
// de su interfaz base. De esta forma el código cliente puede
// soportar componentes de hoja simples así como compuestos
// complejos.
class ImageEditor is
    field all: CompoundGraphic

    method load() is
        all = new CompoundGraphic()
        all.add(new Dot(1, 2))
        all.add(new Circle(5, 3, 10))
        // ...

    // Combina componentes seleccionados para formar un
    // componente compuesto complejo.
    method groupSelected(components: array of Graphic) is
        group = new CompoundGraphic()
        foreach (component in components) do
            group.add(component)
            all.remove(component)
        all.add(group)
            // Se dibujarán todos los componentes.
        all.draw()
```

Aplicabilidad

Utiliza el patrón Composite cuando tengas que implementar una estructura de objetos con forma de árbol.

El patrón Composite te proporciona dos tipos de elementos básicos que comparten una interfaz común: hojas simples y contenedores complejos. Un contenedor puede estar compuesto por hojas y por otros contenedores. Esto te permite construir una estructura de objetos recursivos anidados parecida a un árbol.

Utiliza el patrón cuando quieras que el código cliente trate elementos simples y complejos de la misma forma.

Todos los elementos definidos por el patrón Composite comparten una interfaz común. Utilizando esta interfaz, el cliente no tiene que preocuparse por la clase concreta de los objetos con los que funciona.
Cómo implementarlo

- Asegúrate de que el modelo central de tu aplicación pueda representarse como una estructura de árbol. Intenta dividirlo en elementos simples y contenedores. Recuerda que los contenedores deben ser capaces de contener tanto elementos simples como otros contenedores.
- Declara la interfaz componente con una lista de métodos que tengan sentido para componentes simples y complejos.
- Crea una clase hoja para representar elementos simples. Un programa puede tener varias clases hoja diferentes.
- Crea una clase contenedora para representar elementos complejos. Incluye un campo matriz en esta clase para almacenar referencias a subelementos. La matriz debe poder almacenar hojas y contenedores, así que asegúrate de declararla con el tipo de la interfaz componente.
- Al implementar los métodos de la interfaz componente, recuerda que un contenedor debe delegar la mayor parte del trabajo a los subelementos.
- Por último, define los métodos para añadir y eliminar elementos hijos dentro del contenedor.
- Ten en cuenta que estas operaciones se pueden declarar en la interfaz componente. Esto violaría el Principio de segregación de la interfaz porque los métodos de la clase hoja estarían vacíos. No obstante, el cliente podrá tratar a todos los elementos de la misma manera, incluso al componer el árbol.

Pros y contras

- Puedes trabajar con estructuras de árbol complejas con mayor comodidad: utiliza el polimorfismo y la recursión en tu favor.
- Principio de abierto/cerrado. Puedes introducir nuevos tipos de elemento en la aplicación sin descomponer el código existente, que ahora funciona con el árbol de objetos.
- Puede resultar difícil proporcionar una interfaz común para clases cuya funcionalidad difiere demasiado. En algunos casos, tendrás que generalizar en exceso la interfaz componente, provocando que sea más difícil de comprender.

Relaciones con otros patrones

- Puedes utilizar **Builder** al crear árboles **Composite** complejos porque puedes programar sus pasos de construcción para que funcionen de forma recursiva.
- **Chain of Responsibility** se utiliza a menudo junto a Composite. En este caso, cuando un componente hoja recibe una solicitud, puede pasarla a lo largo de la cadena de todos los componentes padre hasta la raíz del árbol de objetos.
- Puedes utilizar **Iteradores** para recorrer árboles Composite.
- Puedes utilizar el patrón **Visitor** para ejecutar una operación sobre un árbol **Composite** entero.
- Puedes implementar nodos de hoja compartidos del árbol **Composite** como **Flyweights** para ahorrar memoria RAM.

- **Composite** y **Decorator** tienen diagramas de estructura similares ya que ambos se basan en la composición recursiva para organizar un número indefinido de objetos.
  Un Decorator es como un Composite pero sólo tiene un componente hijo. Hay otra diferencia importante: Decorator añade responsabilidades adicionales al objeto envuelto, mientras que Composite se limita a “recapitular” los resultados de sus hijos.
  No obstante, los patrones también pueden colaborar: puedes utilizar el Decorator para extender el comportamiento de un objeto específico del árbol Composite.
- Los diseños que hacen un uso amplio de **Composite** y **Decorator** a menudo pueden beneficiarse del uso de **Prototype**. Aplicar el patrón te permite clonar estructuras complejas en lugar de reconstruirlas desde cero.

Patron State

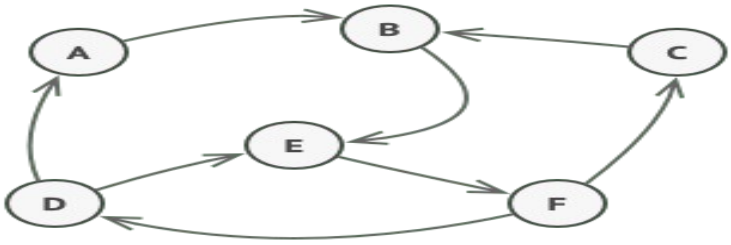
También llamado:Estado

Propósito

States un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara suclase.

Problema

El patrón State está estrechamente relacionado con el concepto de la **Máquina de estados finitos**.



La interfaz **Componente** describe operaciones que son comunes a elementos simples y complejos del árbol.

Máquina de estados finitos.
La idea principal es que, en cualquier momento dado, un programa puede encontrarse en un número finito de estados. Dentro de cada estado único, el programa se comporta de forma diferente y puede cambiar de un estado a otro instantáneamente. Sin embargo, dependiendo de un estado actual, el programa puede cambiar o no a otros estados. Estas normas de cambio llamadas transiciones también son finitas y predeterminadas.

También puedes aplicar esta solución a los objetos. Imagina que tienes una clase `Documento`. Un documento puede encontrarse en uno de estos tres estados: `Borrador`, `Moderación` y `Publicado`.

El método `publicar` del documento funciona de forma ligeramente distinta en cada estado:

- En `Borrador`, mueve el documento a moderación.
- En `Moderación`, hace público el documento, pero sólo si el usuario actual es un administrador.
- En `Publicado`, no hace nada en absoluto.
-

Posibles estados y transiciones de un objeto de documento.

Las máquinas de estado se implementan normalmente con muchos operadores condicionales (`if` o `switch`) que seleccionan el comportamiento adecuado dependiendo del estado actual del objeto. Normalmente, este “estado” es tan solo un grupo de valores de los campos del objeto. Aunque nunca hayas oído hablar de máquinas de estados finitos, probablemente hayas implementado un estado al menos alguna vez. ¿Te suena esta estructura de código?

```
class Document is
  field state: string
  // ...
  method publish() is
    switch (state)
      "draft":
        state = "moderation"
        break
```

```
        "moderation":
            if (currentUser.role == 'admin')
                state = "published"
            break
        "published":
            // No hacer nada.
            break
    // ...
```

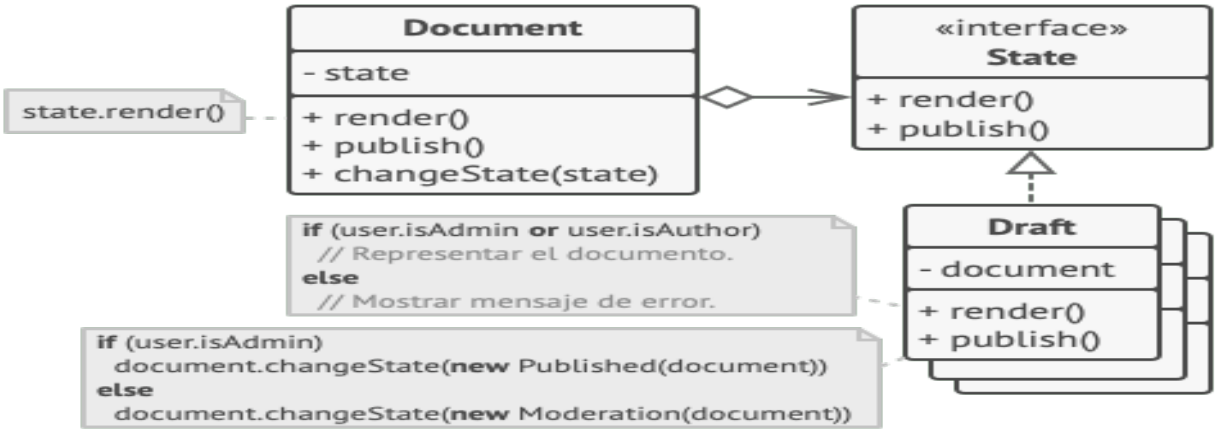
La mayor debilidad de una máquina de estado basada en condicionales se revela una vez que empezamos a añadir más y más estados y comportamientos dependientes de estados a la clase `Documento`. La mayoría de los métodos contendrán condicionales monstruosos que eligen el comportamiento adecuado de un método de acuerdo con el estado actual. Un código así es muy difícil de mantener, porque cualquier cambio en la lógica de transición puede requerir cambiar los condicionales de estado de cada método.

El problema tiende a empeorar con la evolución del proyecto. Es bastante difícil predecir todos los estados y transiciones posibles en la etapa de diseño. Por ello, una máquina de estados esbelta, creada con un grupo limitado de condicionales, puede crecer hasta convertirse en un abotargado desastre con el tiempo.

Solución

El patrón State sugiere que crees nuevas clases para todos los estados posibles de un objeto y extraigas todos los comportamientos específicos del estado para colocarlos dentro de esas clases.

En lugar de implementar todos los comportamientos por su cuenta, el objeto original, llamado contexto, almacena una referencia a uno de los objetos de estado que representa su estado actual y delega todo el trabajo relacionado con el estado a ese objeto.



Documento delega el trabajo a un objeto de estado.

Para la transición del contexto a otro estado, sustituye el objeto de estado activo por otro objeto que represente ese nuevo estado. Esto sólo es posible si todas las clases de estado siguen la misma interfaz y el propio contexto funciona con esos objetos a través de esa interfaz.

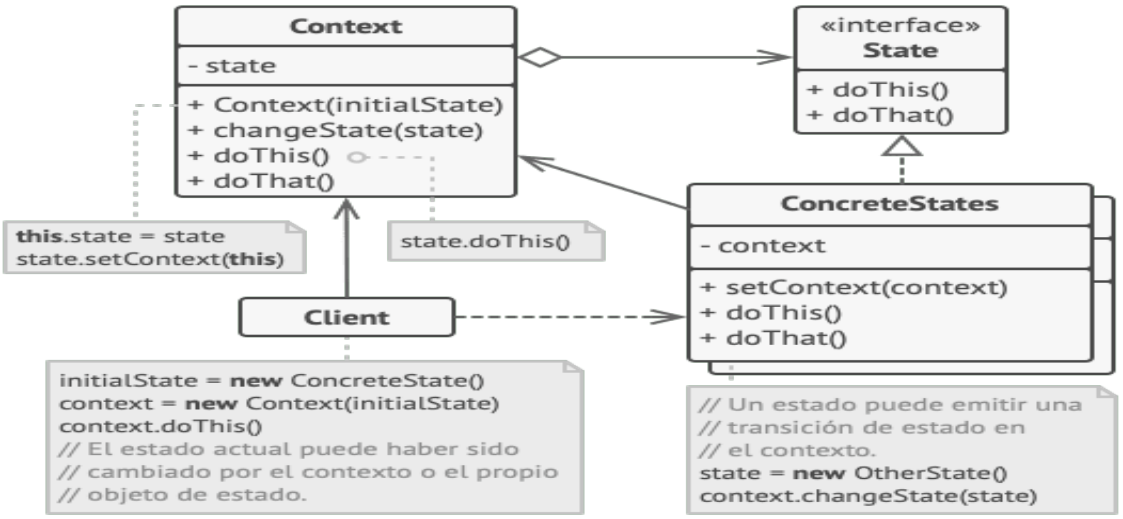
Esta estructura puede resultar similar al patrón **Strategy**, pero hay una diferencia clave. En el patrón State, los estados particulares pueden conocerse entre sí e iniciar transiciones de un estado a otro, mientras que las estrategias casi nunca se conocen.

Analogía en el mundo real

Los botones e interruptores de tu smartphone se comportan de forma diferente dependiendo del estado actual del dispositivo:

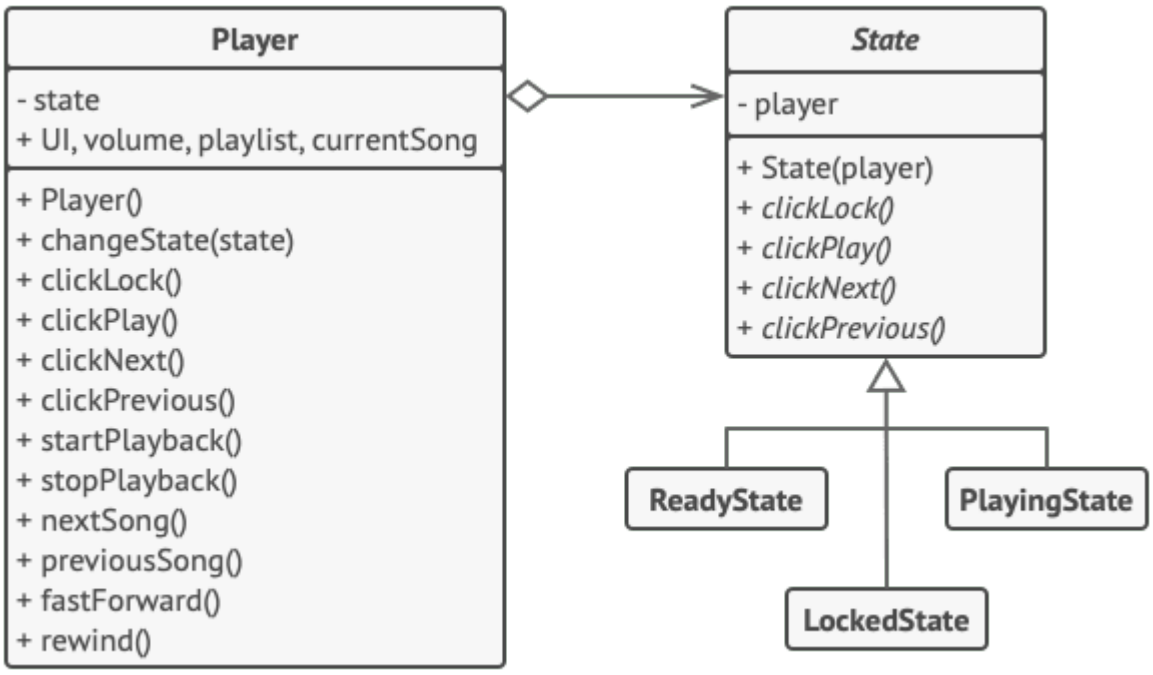
- Cuando el teléfono está desbloqueado, al pulsar botones se ejecutan varias funciones.
- Cuando el teléfono está bloqueado, pulsar un botón desbloquea la pantalla.
- Cuando la batería del teléfono está baja, pulsar un botón muestra la pantalla de carga.

Estructura



1-Pseudocódigo

En este ejemplo, el patrón **State** permite a los mismos controles del reproductor de medios comportarse de forma diferente, dependiendo del estado actual de reproducción.



Ejemplo de cambio del comportamiento de un objeto con objetos de estado.

El objeto principal del reproductor siempre está vinculado a un objeto de estado que realiza la mayor parte del trabajo del reproductor. Algunas acciones sustituyen el objeto de estado actual del reproductor por otro, lo cual cambia la forma en la que el reproductor reacciona a las interacciones del usuario.

```
// La clase ReproductordeAudio actúa como un contexto. También

// mantiene una referencia a una instancia de una de las clases

// estado que representa el estado actual del reproductor de

// audio.

class AudioPlayer is

   field state: State

   field UI, volume, playlist, currentSong


   constructor AudioPlayer() is

      this.state = new ReadyState(this)
```

```
    // El contexto delega la gestión de entradas del usuario
    // a un objeto de estado. Naturalmente, el resultado
    // depende del estado que esté activo ahora, ya que cada
    // estado puede gestionar las entradas de manera
    // diferente.
    UI = new UserInterface()
    UI.lockButton.onClick(this.clickLock)
    UI.playButton.onClick(this.clickPlay)
    UI.nextButton.onClick(this.clickNext)
    UI.prevButton.onClick(this.clickPrevious)

// Otros objetos deben ser capaces de cambiar el estado
// activo del reproductor.
method changeState(state: State) is
    this.state = state

// Los métodos UI delegan la ejecución al estado activo.
method clickLock() is
    state.clickLock()
method clickPlay() is
    state.clickPlay()
method clickNext() is
    state.clickNext()
method clickPrevious() is
    state.clickPrevious()

// Un estado puede invocar algunos métodos del servicio en
// el contexto.
method startPlayback() is
    // ...
```

```
    method stopPlayback() is
        // ...
    method nextSong() is
        // ...
    method previousSong() is
        // ...
    method fastForward(time) is
            // ...
    method rewind(time) is
        // ...



// La clase estado base declara métodos que todos los estados
// concretos deben implementar, y también proporciona una
// referencia inversa al objeto de contexto asociado con el
// estado. Los estados pueden utilizar la referencia inversa
// para dirigir el contexto a otro estado.
abstract class State is
    protected field player: AudioPlayer

    // El contexto se pasa a sí mismo a través del constructor
    // del estado. Esto puede ayudar al estado a extraer
    // información de contexto útil si la necesita.
    constructor State(player) is
        this.player = player

    abstract method clickLock()
    abstract method clickPlay()
    abstract method clickNext()
    abstract method clickPrevious()
```

```
// Los estados concretos implementan varios comportamientos
// asociados a un estado del contexto.
class LockedState extends State is

    // Cuando desbloqueas a un jugador bloqueado, puede asumir
    // uno de dos estados.
    method clickLock() is
        if (player.playing)
            player.changeState(new PlayingState(player))
        else
            player.changeState(new ReadyState(player))

    method clickPlay() is
        // Bloqueado, no hace nada.

    method clickNext() is
        // Bloqueado, no hace nada.

    method clickPrevious() is
        // Bloqueado, no hace nada.

// También pueden disparar transiciones de estado en el
// contexto.
class ReadyState extends State is
    method clickLock() is
        player.changeState(new LockedState(player))

    method clickPlay() is
```

```
        player.startPlayback()
        player.changeState(new PlayingState(player))

    method clickNext() is
        player.nextSong()

    method clickPrevious() is
        player.previousSong()



class PlayingState extends State is
    method clickLock() is
        player.changeState(new LockedState(player))

    method clickPlay() is
        player.stopPlayback()
        player.changeState(new ReadyState(player))

    method clickNext() is
        if (event.doubleclick)
            player.nextSong()
        else
            player.fastForward(5)

    method clickPrevious() is
        if (event.doubleclick)
            player.previous()
        else
            player.rewind(5)
```

Aplicabilidad

Utiliza el patrón State cuando tengas un objeto que se comporta de forma diferente dependiendo de su estado actual, el número de estados sea enorme y el código específico del estado cambie con frecuencia.

El patrón sugiere que extraigas todo el código específico del estado y lo metas dentro de un grupo de clases específicas. Como resultado, puedes añadir nuevos estados o cambiar los existentes independientemente entre sí, reduciendo el costo de mantenimiento.

Utiliza el patrón cuando tengas una clase contaminada con enormes condicionales que alteran el modo en que se comporta la clase de acuerdo con los valores actuales de los campos de la clase.

El patrón State te permite extraer ramas de esos condicionales a métodos de las clases estado correspondientes. Al hacerlo, también puedes limpiar campos temporales y métodos de ayuda implicados en código específico del estado de fuera de tu clase principal.

Utiliza el patrón State cuando tengas mucho código duplicado por estados similares y transiciones de una máquina de estados basada en condiciones.

El patrón State te permite componer jerarquías de clases de estado y reducir la duplicación, extrayendo el código común y metiéndolo en clases abstractas base.

Cómo implementarlo

➔ Decide qué clase actuará como contexto. Puede ser una clase existente que ya tiene el código dependiente del estado, o una nueva clase, si el código específico del estado está distribuido a lo largo de varias clases.

➔ Declara la interfaz de estado. Aunque puede replicar todos los métodos declarados en el contexto, céntrate en los que pueden contener comportamientos específicos del estado.

➔ Para cada estado actual, crea una clase derivada de la interfaz de estado. Después repasa los métodos del contexto y extrae todo el código relacionado con ese estado para meterlo en tu clase recién creada.

Al mover el código a la clase estado, puede que descubras que depende de miembros privados del contexto. Hay varias soluciones alternativas:

- Haz públicos esos campos o métodos.
- Convierte el comportamiento que estás extrayendo para ponerlo en un método público en el contexto e invócalo desde la clase de estado. Esta forma es desagradable pero rápida y siempre podrás arreglarlo más adelante.
- Anida las clases de estado en la clase contexto, pero sólo si tu lenguaje de programación soporta clases anidadas.

➔ En la clase contexto, añade un campo de referencia del tipo de interfaz de estado y un modificador (setter) público que permita sobrescribir el valor de ese campo.

➔ Vuelve a repasar el método del contexto y sustituye los condicionales de estado vacíos por llamadas a métodos correspondientes del objeto de estado.

➔ Para cambiar el estado del contexto, crea una instancia de una de las clases de estado y pásala a la clase contexto. Puedes hacer esto dentro de la propia clase contexto, en distintos estados, o en el cliente. Se haga de una forma u otra, la clase se vuelve dependiente de la clase de estado concreto que instancia.

Pros y contras

- Principio de responsabilidad única. Organiza el código relacionado con estados particulares en clases separadas.
- Principio de abierto/cerrado. Introduce nuevos estados sin cambiar clases de estado existentes o la clase contexto.
- Simplifica el código del contexto eliminando voluminosos condicionales de máquina de estados.
- Aplicar el patrón puede resultar excesivo si una máquina de estados sólo tiene unos pocos estados o raramente cambia.

Relaciones con otros patrones

•**Bridge**, **State**, **Strategy** (y, hasta cierto punto,**Adapter**) tienen estructuras muy similares. De hecho, todos estos patrones se basan en la composición, que consiste en delegar trabajo a otros objetos. Sin embargo, todos ellos solucionan problemas diferentes. Un patrón no es simplemente una receta para estructurar tu código de una forma específica. También puede comunicar a otros desarrolladores el problema que resuelve.
•**State** puede considerarse una extensión de **Strategy**. Ambos patrones se basan en la composición: cambian el comportamiento del contexto delegando parte del trabajo a objetos ayudantes. Strategy hace que estos objetos sean completamente independientes y no se conozcan entre sí. Sin embargo, State no restringe las dependencias entre estados concretos, permitiéndoles alterar el estado del contexto a voluntad.

Spring Boot

› Repaso módulo Spring Boot

// Conceptos básicos

Arquitectura Multicapa

Cuando creamos nuestro proyecto, la estructura debe representar la arquitectura multicapa elegida. Cada capa debe incluirse en un paquete específico, con el nombre de la misma.

Ojo: No existe un modelo único y dependiendo del proyecto tendremos más o menos capas.

Podemos incluir las siguientes capas:

- **Controller**: Se encarga de atender una request desde el momento en que es solicitada hasta la generación de la response y su transmisión.

  Lo que hace un controlador es llamar a una o más funciones de la capa de servicio. También gestiona la deserialización de una solicitud y la serialización de la respuesta, a través de la capa DTO.

  Cada clase de controlador debe estar marcada con la anotación **@RestController**.

  En cada Controller se especifican los **endpoint** a exponer al cliente.

- **DTO (Data Transfer Object)**: Se usan para transferir información. En Spring dentro del @RestController podemos retornar directamente un DTO y el framework se encarga de transformarlo a formato JSON.

  Ojo: Los DTOs son objetos planos, no deben tener lógica, solo se usan para enviar y recibir datos. No debemos olvidarnos los getters, setters y el constructor (eso es mucho muy importante)

- **Service**: La capa de servicio es el lugar donde ubicamos cualquier operación de lógica de negocio que uno o más controladores necesiten.

  Cada clase de servicio debe marcarse con la anotación **@Service**.

- **Repository**: Es la capa de persistencia de datos. Cada clase del repositorio debe estar marcada con la anotación **@Repository**.

  Esta capa se encarga de buscar en los datos guardados información para el service. Contiene funciones SQL, trabaja con colecciones de objetos. Por debajo podemos tener la capa DAO, que ofrece el crud jpa y trabaja con un objeto (a fines prácticos y por simplicidad para nosotrxs repository y dao serán sinónimos).

Estructura del proyecto

La estructura de un proyecto debería verse similar a la que se puede apreciar en la Imagen 1, donde en cada uno de esos paquetes crearemos las clases correspondientes.

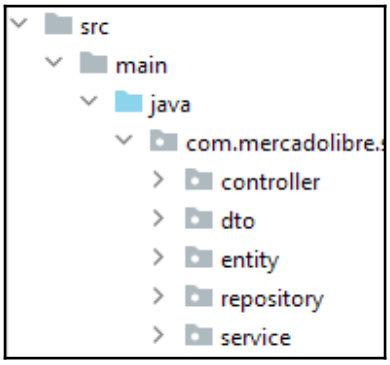


Imagen 1: Estructura del proyecto.

// Nociones a tener en cuenta

- Cada **endpoint** puede ser principalmente de los métodos GET, POST, PUT y DELETE (ver especificaciones mas detalladas de cuando utilizar cada uno). Dentro de Spring Boot se da el método especificado a través de las notaciones **@GetMapping, @PostMapping, @PutMapping y @DeleteMapping**.

- Para modelar recursos y asignaciones para diferentes verbos HTTP con Spring Boot, usamos la anotación **@RequestMapping**. Es aplicable a nivel de clase y a nivel de método, por lo que podemos construir nuestras API's de una manera sencilla. Para hacerlo aún más simple, el framework proporciona variantes como **@PostMapping**, **@GetMapping**, etc..

- Si queremos pasar el cuerpo de una solicitud (request) a un método dentro del controlador, usamos la anotación **@RequestBody**. Si usamos una clase personalizada, Spring Boot intentará deserializarla, utilizando el tipo pasado al método.

- La anotación **@RestController** le dice a Spring que este es un componente especializado modelado de un controlador REST. Es una combinación de @Controller y @ResponseBody, que le indica a Spring que coloque el resultado de los métodos como cuerpo de una respuesta HTTP.

- Por defecto en Spring Boot (y si no se indica lo contrario), la respuesta se serializará como **JSON** y se incluirá en el cuerpo de la respuesta (response).

- Spring **@Autowired** es una de las anotaciones más habituales cuando trabajamos con Spring ya que se trata de la anotación que permite inyectar unas dependencias con otras. Una vez que los objetos están creados, la anotación Spring **@Autowired** se encarga de construir las relaciones entre los distintos elementos.
- Cuando usamos la inyección de dependencias (por ejemplo con @Autowired) el tipo del atributo que vamos a inyectar va a ser una interfaz y no un tipo concreto que se haya implementado.

// Manejo de errores

- **@ControllerAdvice**: La anotación @ControllerAdvice nos permite consolidar múltiples @ExceptionHandlers dispersos de antes, en un solo componente global de manejo de errores.

- **@ExceptionHandler**: Esta anotación, como su nombre lo indica, proporciona una forma de definir métodos en un controlador en particular para manejar específicamente las excepciones lanzadas por dichos métodos **@ResponseStatus:** vincula una excepción particular a un estado de respuesta HTTP específico. Entonces, cuando Spring detecta esa excepción en particular, genera una respuesta HTTP con la configuración definida allí. Marca un método o clase de excepción con el **código de estado** y el **mensaje** de motivo que debe devolverse. El código de estado se aplica a la respuesta HTTP cuando se invoca el método del controlador o siempre que se lanza la excepción especificada.

- Testing

- Repaso Testing

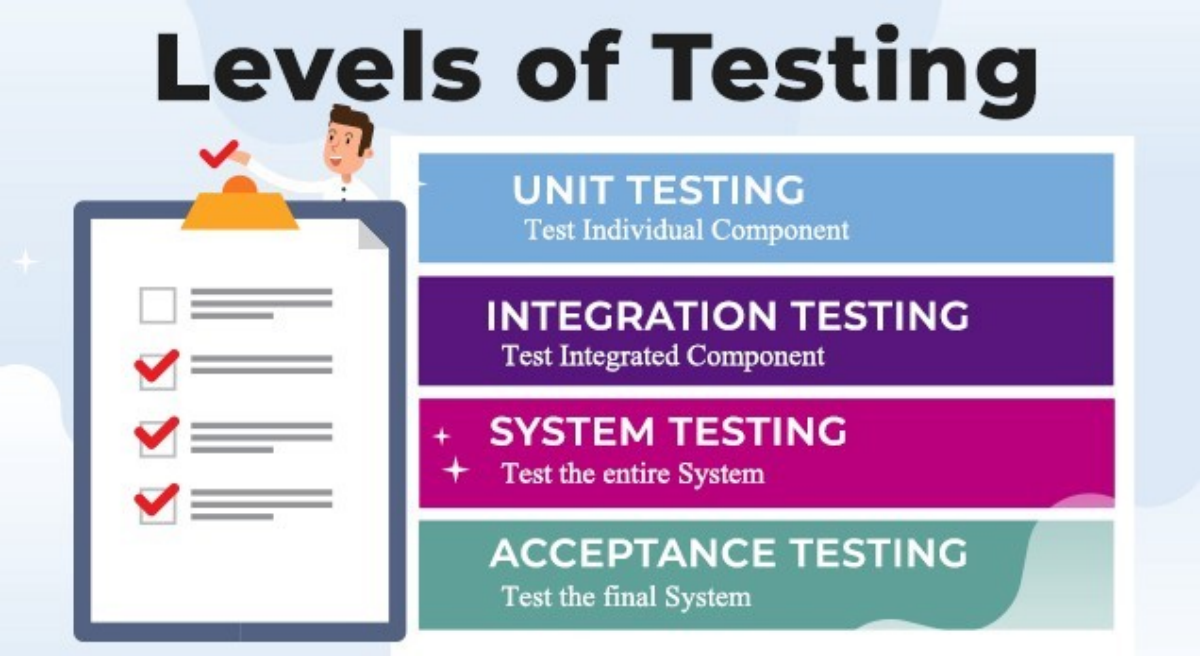
// Conceptos básicos

- Tipos de pruebas de software

Existen multitud de pruebas. En este artículo, nos vamos a centrar en explicar el explicar el siguiente mapa. Conocer las diferentes pruebas de testing que existen, nos permitirá saber en qué punto estamos situados en cada momento.

Pruebas no funcionales (Non-Functional Testing): Son pruebas no funcionales, prueban los atributos de un componente o de un sistema que no se refieren a la funcionalidad. Algunos ejemplos pueden ser: seguridad, fiabilidad, eficiencia, escalabilidad, usabilidad, mantenibilidad, instalación, portabilidad, etc.

Tipos de pruebas funciones (Nivel 2)



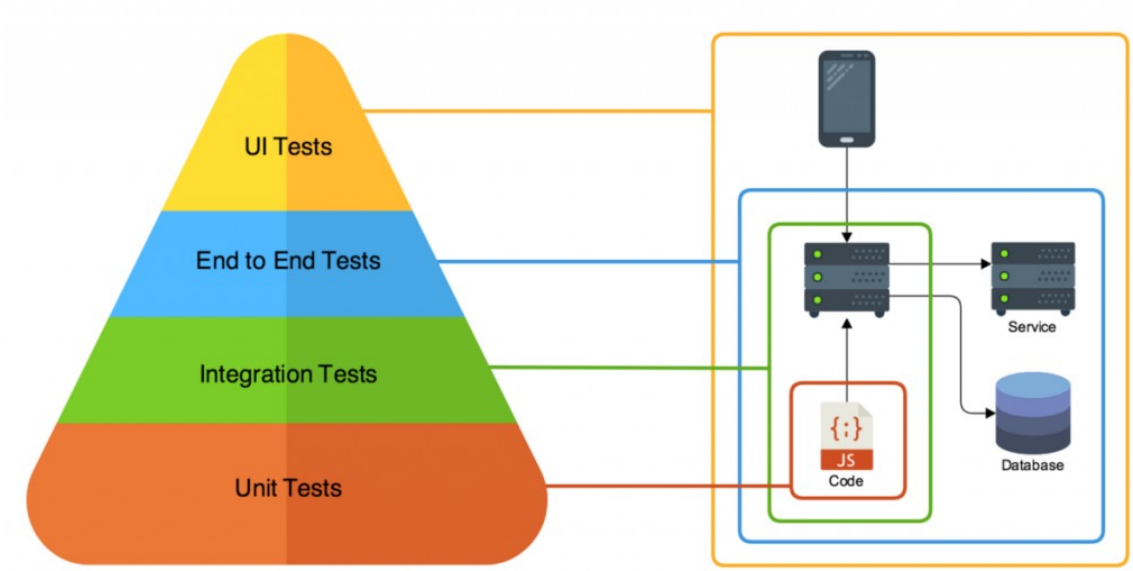
Las principales pruebas funcionales son:

•Pruebas unitarias (Unit Testing): el objetivo de estas pruebas es probar los componentes de un programa individualmente (un ejemplo podría ser el testing de una función o de una clase).

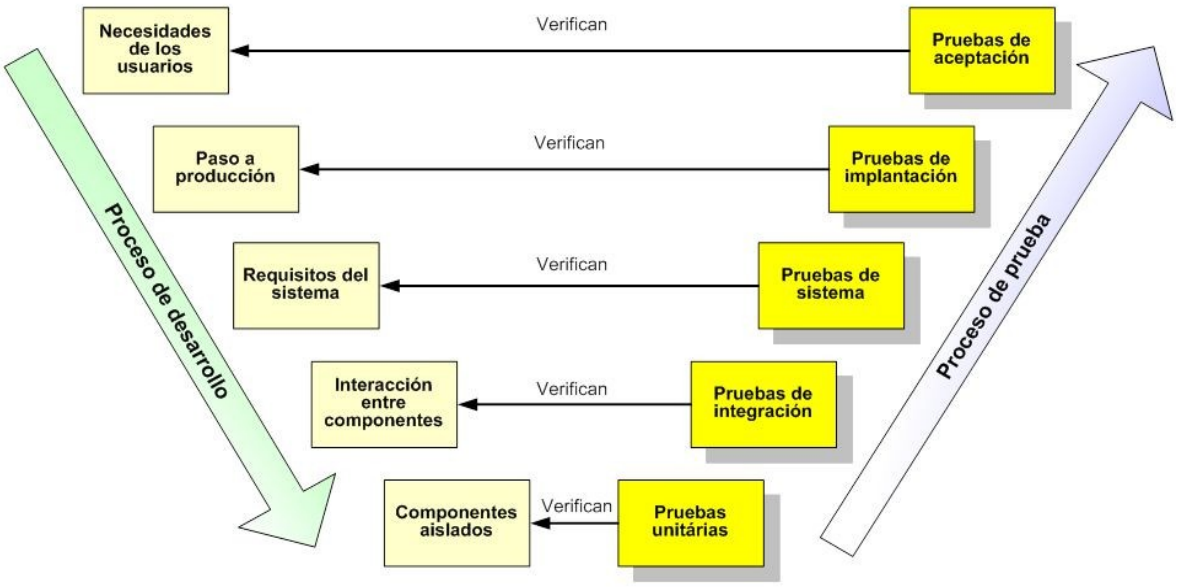
•Pruebas de integración (Integration Testing): el objetivo de estas pruebas es verificar el correcto ensamblaje de un conjunto de componentes en el entorno. Los cuales, ya han sido probados anteriormente de forma individual mediante a las pruebas unitarias.

•Pruebas de regresión (End to End Testing): el objetivo de estas pruebas es verificar el correcto funcionamiento de un componente modificado, y a su vez, verificar que el componente modificado no a afectado al correcto funcionamiento del resto de componentes. Se realiza con la finalidad de detectar posibles bugs (fallos) tanto en dicho componente, como en otros componentes. Se ejecutan cuando se implementa un nuevo componente dentro del entorno.

•Pruebas de aceptación o (User Acceptance Testing): el objetivo de estas pruebas es que clientes testeen el software para verificar que cumple las expectativas.


UI Tests
End to End Tests
Integration Tests
Unit Tests
Service
Database
JS
Code

Niveles de prueba



Tipos de pruebas no funcionales (Nivel 2)

Las principales pruebas no funcionales son:

- Pruebas de Rendimiento (Performance): el objetivo de estas pruebas es garantizar el funcionamiento bajo una carga de trabajo esperada.

•Pruebas de carga (Load Testing): el objetivo de estas pruebas es testear la capacidad de la aplicación bajo cargas previstas por el cliente.

•Pruebas de Estrés (Stress Testing): el objetivo de estas pruebas es testear cómo reacciona la aplicación a cargas de trabajo extremas, como son: un tráfico elevado o una gran cantidad de procesamiento de datos.

•Pruebas de Escalabilidad (Scalability Testing): el objetivo de estas pruebas es determinar la capacidad de adaptación de la aplicación al aumento de carga de de usuarios (escalabilidad).

•Pruebas de Volumen (Volume Testing): el objetivo de estas pruebas es determinar el rendimiento de la aplicación según el volumen de datos que contiene la BBDD.

Más información: https://javadesde0.com/introduccion-a-los-test-unitarios-unit-testing/

-

-

ANEXO 1 Preguntas del aula y videos

Qué es un Factory? seguro les va ser útil para entenderlo.

FACTORY

Es una interfaz, se crean varias implementaciones y una clase get que dependiendo el string que se use, devuelve la implementación que corresponde)
El patrón de diseño “Factory” se enfoca en la creación de una interfaz que facilite la creación de objetos que se organizan por diferentes subclases. Esto ocurre con frecuencia cuando se hace uso de la herencia. Una clase abstracta se genera conteniendo los atributos generales, y después se crean clases para objetos específicos. Para evitar llamar constructores específicos se deben crear interfaces que nos ayuden en estas tareas.
En este patrón de diseño hay dos formas de generar las interfaces para crear objetos: El método Factory (Factory method) y el “Abstract Factory”.
El Factory se trata de un simple método agregado dentro de una clase que, por medio de un parámetro o varios parámetros, se decide qué tipo de objeto crear para entonces retornarlo y poder utilizarlo. Esto hace que no tengamos que importar numerosas clases y mucho menos instanciar numerosos objetos de esas mismas clases.

https://www.oscarblancarteblog.com/2014/07/18/patron-de-diseno-factory/#:~:text=El%20patr%C3%B3n%20de%20dise%C3%B1o%20Factory,detalles%20de%20creaci%C3%B3n%20del%20objeto.

Patrón de Diseño Factory - Oscar Blancarte - Software Architecture (Oscar Blancarte - Software Architecture)

El patrón de diseño Factory Method es uno de los patrones de diseño más utilizados en el desarrollo de software.

Videos

https://www.youtube.com/c/ByteCodeHN

https://www.oscarblancarteblog.com/2018/11/30/data-transfer-object-dto-patron-diseno/

Data Transfer Object (DTO) – Patrón de diseño

El patrón Data Transfer Object (DTO) son clases planas que nos permite transmitir información de múltiples fuentes de datos o tablas desde el servidor a un cliente en una sola invocación.

Ejemplo tenes una clase que adentro tiene un atributo que es una instancia de otra clase
tipo que consultorio tiene dentro un dentista, en la tabla se almacena
el id del dentista y vos podes consultar la query de esta manera
Dentro del repo de "consultorio" quiero la que tiene el nombre
de consultorio y el dentista con "id"

Pregunta:

JPA: How to save two entities that have one-to-many relation?

Respuesta: Recuerden que cuando creamos relaciones entre entidades, esas relaciones a nivel de base de datos se establecen por medio de atributos que fungen como FK.A nivel de código Java nosotros especificamos esas relaciones a través de atributos de clases cuyo tipo sería el modelo con el cuál queremos relacionar

.**Ejemplo:**

```
public class Profile {
        private String firstname;
        private String lastname;
}
```

ByteCode:https://www.youtube.com/c/ByteCodeHN
Web:https://www.youtube.com/channel/UCRUSZxZURhuBCPqRjTF9PUA/about ►
Magtimus:https://www.youtube.com/channel/UCgY6raSf9MhsMjXwmG83-
CodyTron:https://www.youtube.com/channel/UC3bi2lyXjvM4yvz8QkSboyA

Antipatrones

https://www.youtube.com/playlist?list=PLZVwXPbHD1KN1JRaNzcU8-QA-oUkgH79C
https://www.youtube.com/watch?v=HrK1kijS82g

Spring Boot

YouTube Techstack I/O **Unit and Integration Testing in Spring Boot**
https://www.youtube.com/watch?v=N3uAMuC-bxo
El ModelMapper, video que enseña a evitar crear el mapper a mano

YouTube Java Guides **Entity To DTO Conversion using ModelMapper in Spring Boot Application**

Agradecer al grupo de Bootcamp Mercado Libre

A Miguel Arjona

Gabriela Monzón por sus aportes y todo el resto del equipo.


DigitalHouse >
Coding School
Buenos Aires , Argentina
www.digitalhouse.com