ILERNA Online

Videotutoría 6: Refactorización

Módulo 05: Entornos de Desarrollo

RESUMEN

¿Qué habéis aprendido la VT anterior?

ILERNA Online

Objetivos VT 6

- Refactorización
- Junit
- Documentación

Clases de equivalencia

- Prueba de caja negra que divide los valores de los campos de entrada de un programa en clases de equivalencia.
- Se podrá definir dos tipos de clases de equivalencia:
  - **Clase válidas:** valores de entrada válidos.
  - **Clase no válidas:** valores de entrada no válidos.

Clases de equivalencia: Ejemplo

Considérese una aplicación bancaria, donde el usuario puede conectarse al banco por Internet y realizar una serie de operaciones bancarias. Una vez accedido al banco con las consiguientes medidas de seguridad (clave de acceso y demás), la información de entrada del procedimiento que gestiona las operaciones concretas a realizar por el usuario requiere la siguiente entrada:

-Código del banco. En blanco o número de tres dígitos. En este último caso, el primero de los tiene que ser mayor que 1. (Requisito funcional porque es algo que nosotros vamos a poder programar.)

-Código de sucursal. Un número de cuatro dígitos. El primero de ellos mayor de 0.

-Número de cuenta. Número de cinco dígitos.

-Clave personal. Valor alfanumérico de cinco posiciones.

-Orden. Este valor se introducirá según la orden que se desee realizar. Puede estar en blanco o ser una de las dos cadenas siguientes: o “Talonario” o “Movimientos”

Cuando hablamos de requisitos o aquellos requerimientos que nos surgen durante el análisis, hablamos sobretodo de requisitos funcionales y requisitos no funcionales.

Clases de equivalencia: Ejemplo

Condición de Entrada	Tipo	Clase Equivalencia Válida	Clase Equivalencia No Válida
Código banco	Lógica (puede estar o no) Si está es Rango	1: En blanco 2: 100<= Código banco <= 999	3: Un valor no numérico 4: Código banco < 100 5: Código banco > 999
Código sucursal	Rango	6: 1000 <= Código sucursal <= 9999	7: Código sucursal < 1000 8: Código sucursal >= 9999
Nº Cuenta	Valor	9: Cualquier número de cinco dígitos	10: Número de menos de cinco dígitos 11: Número de más de cinco dígitos
Clave	Valor	12: Cualquier cadena de caracteres alfanuméricos de 5 posiciones	13: Cadena de menos de cinco posiciones 14: Cadena de más de cinco posiciones
Orden	Conjunto, con comportamiento distinto	15: “” 16: “Talonario” 17: “Movimientos”	18: Cadena distinto de blanco y de las válidas 19: “Talonarios” 20: “Movimiento”

Análisis de valores límite

Prueba de caja negra donde se basa en la hipótesis de que suelen ocurrir más errores en los valores extremos de los campos de entrada.

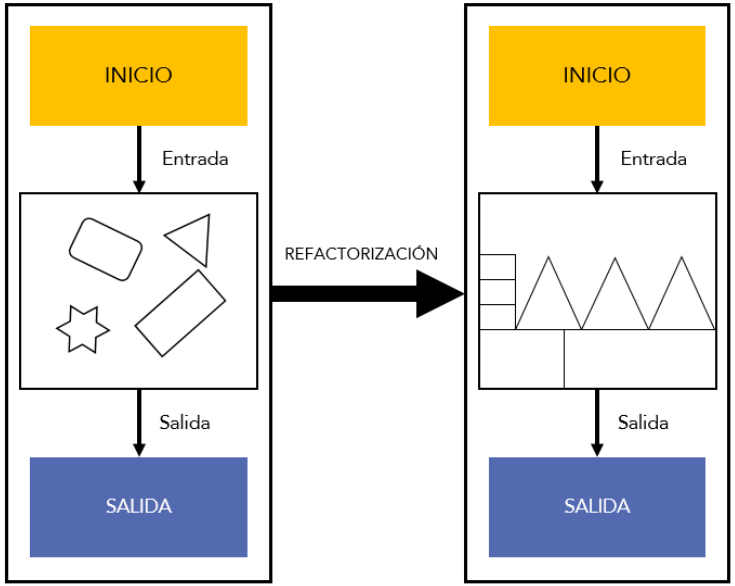
Complementa a la técnica anterior. Además, no solo estará centrado en las condiciones de entrada, sino que definen también las clases de salida.

Ejemplo: para un rango de valores enteros que estén comprendidos entre 5 y 15, tenemos que escribir casos de prueba para 5, 15, 4 y 16.

Refactorización

La refactorización nos va a permitir optimizar un código que se ha escrito previamente, realizando cambios en la estructura interna sin que afecten al comportamiento final del producto.

Funciona, pero esta desordenado, clases repetidas... Falta optimizarlo.



Bad Smells

- El ingeniero de software británico Martin Fowler y otros expertos diagnosticaron los ***bad smells*** (malos olores), es decir, pequeños indicios que indican que el sistema no funciona como es debido.(bibliografía *"Refactoring" de Kent Beck y Martin Fowler)* La salida puede tardar mucho más.
  - **Código duplicado**: código repetido
  - **Métodos muy largos** En una programación modular, cuando hay que dividirla en clases o módulos, no hacerlo más allá de 7 niveles. Dentro de un módulo no tener más de 50 líneas de código.
  - **Clases muy extensas**: muchas responsabilidades al tener demasiados métodos y atributos.
  - **Lista de parámetros extensa**: Las funciones deben tener el mínimo de parámetros posible
  - **Cambio divergente**: Una clase cambia sus métodos cada vez que hay un cambio sustancial en la estructura (p.ej: BBDD) No puede ser que por añadir una tabla tengamos que cambiar todo nuestro aplicativo.
  - **Cirugía a tiro de pistola**: cuando el programador tiene que hacer un montón de pequeños cambios sin ton ni son a un montón de clases diferentes porque el módulo no funciona. Parcheamos en lugar de realizar un código lo más óptimo posible.

Bad Smells

- El ingeniero de software británico Martin Fowler y otros expertos diagnosticaron los ***bad smells*** (malos olores), es decir, pequeños indicios que indican que el sistema no funciona como es debido.(bibliografía *“Refactoring” de Kent Beck y Martin Fowler)*
    - **Envidia de funcionalidad** : Ocurre cuando un método usa más elementos de otra clase que de la suya propia. Si tengo que estar llamando constantemente a otro método, a lo mejor es que me sobra.
    - **Clase de solo datos**: Clase que solo tiene atributos y métodos de acceso. No debería ser lo habitual. Tal vez podríamos refactorizarla y pasar sus datos a distintos módulos.
    - **Legado rechazado**: subclases que usan características de superclase, lo que pude inducir a un error en la jerarquía de clases. Tal vez esa subclase no deba tener la herencia de esa superclase.

JUNIT

MÉTODOS	MISIÓN
assertTrue(boolean expresión) assertTrue(String mensaje, boolean expression)	Comprueba que la expresión se evalúe *true*. Si no es *true* y se incluye el String, al producirse error se lanzará el *mensaje*.
assertFalse(Boolean expresión) assertFalse(String mensaje, Boolean expresión)	Comprueba que la expresión se evalúe *false*. Si no es *false* y se incluye el String, al producirse error se lanzará el *mensaje*.
assertEquals(valorEsperado, valorReal), assertEquals(String mensaje, valorEsperado, valorReal)	Comprueba que el *valorEsperado* sea igual al *valorReal*. Si no son iguales y se incluye el String, entonces se lanzará el *mensaje*. *ValorEsperado* y *ValorReal* pueden ser de diferentes tipos.
assertNull(Object objeto), assertNull(String mensaje, Object objeto)	Comprueba que el *objeto* sea *null*. Si no es *null* y se incluye el String, al producirse error se lanzará el mensaje.

Documentación

- **Documentación del diseño**: en este documento se describe toda la estructura interna del programa, formas de implementación, contenido de clases, métodos, objetos, etc. Documentación antes de la implementación.

- **Documentación del código fuente**: durante el desarrollo del proyecto se debe ir comentando en el código fuente cada parte, para tener una mayor claridad de lo que se quiere conseguir en cada sección (Javadoc).

- **Documentación de usuario final**: documentación que se entrega al cliente en la que se describe cómo usar las aplicaciones del proyecto.

Documentación

- La documentación del código del programa también es fundamental para que todo el equipo pueda realizar funciones de actualización y reparación de errores de manera mucho más sencilla.
- Hay 2 reglas que no se deben olvidar:
    - Todos los programas poseen errores y es cuestión de tiempo que se detecten.
    - Todos los programas sufren modificaciones a lo largo de su vida.
- **Testeo a pares** : Al realizar las modificaciones es necesario que el código esté bien documentado para que otro programador ajeno localice los cambios que quiere realizar.
- **Documentación útil**: Al documentarlo, habrá que explicar lo que realiza una clase o un método y por qué y para qué lo hace.

Javadoc

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

/**
 *
 * @param num1
 * Este es el parámetro 1
 * @param num2
 * Este es el parámetro 2
 * @return
 */
```

Suma

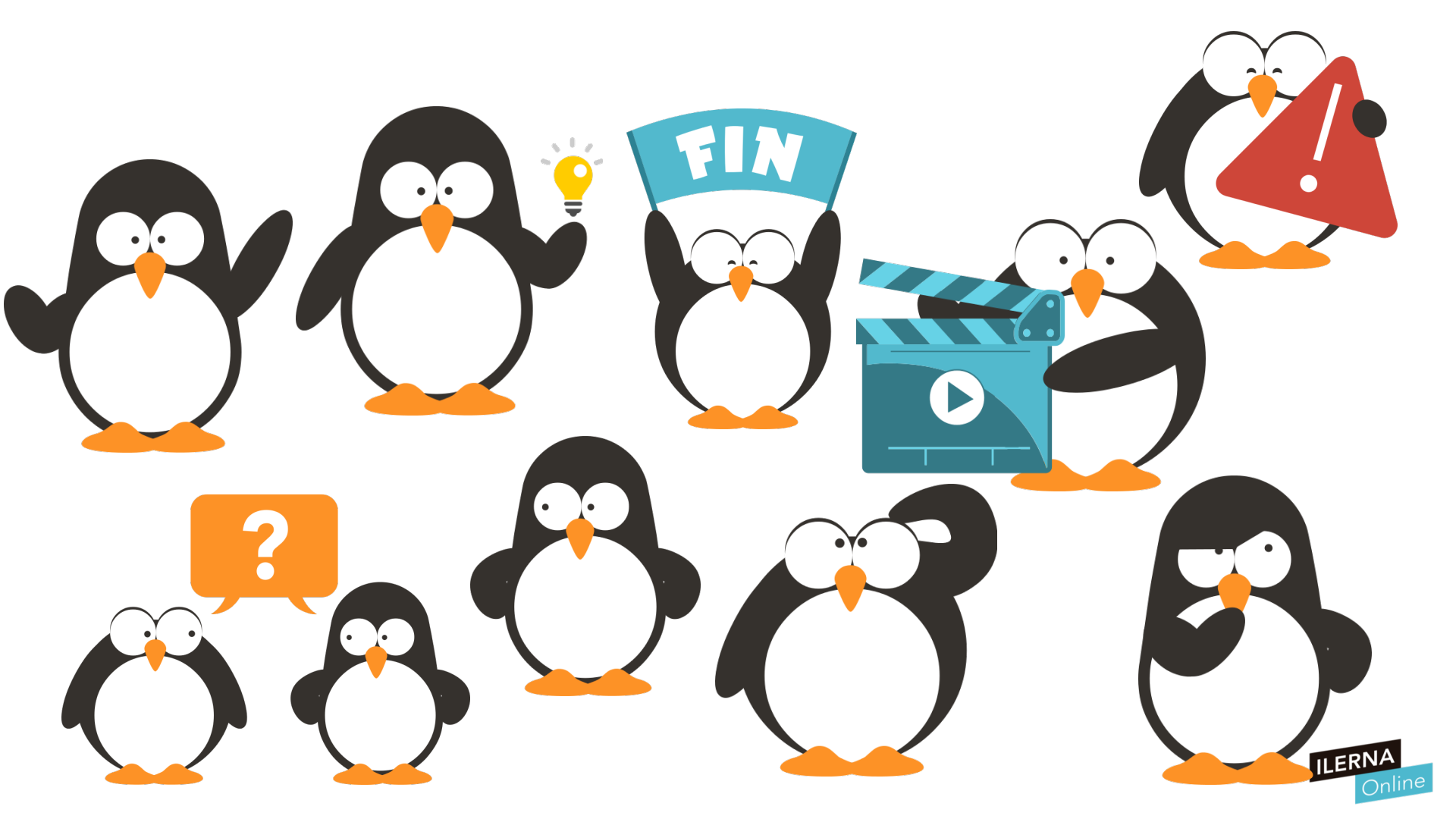
```
public int Suma(int num1,
                int num2)
```

Parameters:

num1 - Este es el parámetro 1

num2 - Este es el parámetro 2

Returns:


FIN
?
ILERNA
Online