# ENTORNOS DE DESARROLLO

ciclos formativos de grado superior de
desarrollo de aplicaciones multiplataforma

## CAPÍTULO 3

Carlos Casado Iglesias

Ra-Ma®

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 1. HERRAMIENTAS DE DEPURACIÓN

La **depuración** es uno de los procesos más importantes en el desarrollo de software, nos permite identificar y corregir errores de programación mediante la ejecución controlada del software.

Esta tarea la realiza el **depurador**, uno de los componentes básicos de un entorno de desarrollo, es, sin duda, una de las herramientas más importantes de un IDE.

Gracias al **depurador**, podemos ver el **proceso** de un programa paso a paso, observando los valores de nuestros métodos, variables y objetos, lo cual facilita sumamente la tarea, o podemos simplemente establecer puntos de control que **interrumpen** la ejecución del programa mostrándonos el código en donde pusimos el punto de **interrupción** con los valores actuales.

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 1. HERRAMIENTAS DE DEPURACIÓN

Podemos situar **puntos de ruptura** o puntos de interrupción en líneas concretas de nuestro código fuente para que el depurador detenga la ejecución del programa cuando alcance uno de ellos.

Se pueden colocar todos los puntos de **interrupción** que se desee, lo cual resulta muy útil cuando se está depurando un error que tiene una larga pila de llamadas.

También se pueden colocar puntos de interrupción de forma manual lo cual resulta extremadamente útil en situaciones donde la depuración asistida desde la ejecución del IDE no sea posible.

Los puntos de interrupción se pueden personalizar, añadiendo una condición o un filtro. Para añadir una condición a un punto de interrupción hay que hacer clic con el botón derecho del ratón en el punto de ruptura y seleccionar

**Condición**.

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 1. HERRAMIENTAS DE DEPURACIÓN

Las condiciones que podemos utilizar en el punto de interrupción condicional no tienen límite y son extremadamente útiles para establecer puntos de interrupción muy concretos, especialmente en estructuras de bucle.

También podemos utilizar **puntos de seguimiento** que tienen una funcionalidad semejante a los puntos de interrupción. La diferencia radica en que un punto de seguimiento no detiene el programa.

Resultan especialmente útiles para llevar un seguimiento de una operación concreta como **inspeccionar** diversos valores en los diferentes momentos de ejecución o establecer contadores de ejecución en la línea donde sitúas el punto de seguimiento.

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 1. HERRAMIENTAS DE DEPURACIÓN

Durante la **depuración** de código, es importante conocer el valor que tienen nuestras variables, las propiedades de nuestros objetos, los elementos y valores de nuestras colecciones o el resultado de un condicional.

Para ello, agregamos **inspecciones**, podemos agregar inspecciones existentes en nuestro código o personalizar los nuestros propios, por ejemplo, podríamos crear una inspección de un condicional que no existe en nuestro código o comprobar el valor de una propiedad de una instancia concreta de un objeto en un momento determinado.

Una vez agregadas las inspecciones, podemos ver los valores y cómo van cambiando mientras depuramos, lo que nos permitirá identificar dónde se encuentra un problema determinado mientras depuramos.

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 2. ANÁLISIS DE CÓDIGO

Una de las tareas más importantes que deberemos realizar mientras desarrollamos software es asegurarnos de que nuestro código funcionará correctamente.

Si nos tomamos el tiempo necesario para analizar lo que debemos hacer para conseguir el producto deseado, evitaremos que nos ocurran problemas que no parecen evidentes a simple vista, como organizar nuestra estructura de clases de una forma que aseguremos la coherencia del contexto de una petición *http* en una aplicación web.

Para ayudarnos a identificar en tiempo real errores en el código producto de despistes o de malas prácticas de desarrollo de software, contamos con el analizador estático de código de **Visual Studio** y la definición de reglas para el análisis.

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 2. ANÁLISIS DE CÓDIGO

El analizador de código tiene tres modos de hacernos saber que algo va mal o podría ir mal: los errores, las advertencias y los mensajes. Inicialmente, **Visual Studio** nos muestra en la ventana de errores los fallos de **compilación**, es decir, nos avisa de que algo en nuestro código no podrá ser compilado debido a los errores que contiene.

Por defecto, tenemos una serie de reglas o paquetes de reglas disponibles, pero podremos descargar más reglas para activarlas en nuestro código o incluso definir qué reglas queremos usar y cuáles no.

Podemos ver detalladamente el conjunto de reglas denominado "*Todas las reglas de Microsoft*", lo que sin duda nos dará una idea más clara del uso y potencial de esta herramienta. Para ello, iremos a las propiedades del proyecto, a la pestaña **Análisis de código** y haremos clic en el botón **Abrir**.

Podremos también definir un conjunto de reglas personalizado para evitar molestos mensajes que no nos interesen y asegurarnos de que los mensajes indiquen un fallo que queremos solucionar. Para ello nos dirigiremos a **Archivo > Nuevo > Archivo** y elegiremos **Conjunto de reglas de análisis de código** como la plantilla que hay que utilizar.

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 2. ANÁLISIS DE CÓDIGO

### ➢ CONTRATOS DE CÓDIGO

Los contratos de código son un modo de generar advertencias y mensajes para casos que determinemos. De este modo podemos estar atentos a los diferentes problemas que podrían surgir, siempre y cuando hayamos pensado en ellos antes.

Para ello se utiliza **System.Diagnostics.Contracts**, que nos permitirá utilizar los métodos **Ensure** y **Requires** para nuestros propósitos.

**Ensures:** Permite establecer un contrato de código para comprobar o asegurarnos de que el valor de retorno del método cumpla unos requisitos específicos. Comprueba valores de salida o retorno.

**Requires:** Mediante el uso de requires lanzamos una excepción cuando un parámetro no cumpla unos requisitos específicos. Comprueba valores de parámetros o de entrada.

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 3. CASOS DE PRUEBA

Los **casos de prueba** son una serie de condiciones que se establecen con el objetivo de determinar si la aplicación funciona correctamente según lo esperado. Para cada tarea, pueden surgir diversos casos de prueba, teniendo en cuenta todos los factores posibles para no dejar ningún cabo suelto sin probar y evitar así que ocurran errores no conocidos.

En principio los casos de prueba tienen un enfoque genérico y posteriormente se añaden más condiciones y variables que lo completen.

Existen diversos tipos de casos de prueba que se definen por la naturaleza de las pruebas y no por el tipo de operación que conlleva la prueba, por ejemplo, aunque las pruebas de **caja blanca** vayan sumamente ligadas al procedimiento en sí como una correcta evaluación de un condicional y las pruebas de **caja negra** se realicen desde el punto de vista del usuario final, podemos estar comprobando la misma operación y los mismos datos aunque sea desde un punto de vista diferente.

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 3. CASOS DE PRUEBA

- **CAJA BLANCA**

Las pruebas de caja blanca se centran en el funcionamiento interno del programa, observando y comprobando cómo se realiza una operación.

**Prueba del camino básico:** Este método se basa en el principio que establece que cualquier diseño procedimental se puede representar mediante un grafo de flujo. La complejidad *ciclomática* de dicho grafo establece el número de caminos independientes, cada uno de esos caminos se corresponde con un nuevo conjunto de sentencias o una nueva condición.

**Prueba de condiciones:** Evalúan los caminos posibles, en este caso de forma que solo provengan de condicionales. Para que las pruebas sean efectivas, no deben ser redundantes, por lo que a la hora de construir nuestra tabla de la verdad, debemos tener presentes las condiciones cortocircuitadas.

**Pruebas de bucles:** Evalúan todas las opciones posibles para un bucle de '*n*' iteraciones.

- ✓ El fujo del programa no entra ninguna vez al bucle.
- ✓ Pasa una única vez por el bucle.
- ✓ Pasa dos veces por el bucle.
- ✓ Pasa *m* veces por el bucle, donde $m<n$.
- ✓ Hace $n$-1 y $n$+2 iteraciones en el bucle.

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 3. CASOS DE PRUEBA

- **CAJA NEGRA**

Las pruebas de caja negra se enfocan en los métodos de entrada y salida de la aplicación, no en cuestión de formato, sino de validar y controlar los datos de entrada para evitar errores y, por supuesto, al igual que en todas las pruebas, obtener los resultados esperados.

**Partición equivalente:** La partición equivalente es un método de prueba consistente en dividir y separar los campos de entrada según el tipo de dato y las restricciones que conllevan.

- ✓ Si el campo debe de encontrarse en un rango, se especifica una clase de equivalencia válida y dos inválidas (los límites inferiores y superiores).
- ✓ Si el campo requiere de una entrada específica, se define una clase de equivalencia válida y dos inválidas.
- ✓ Si el campo especifica a un elemento de un conjunto, se define una clase de equivalencia válida y otra inválida.
- ✓ Si el campo especifica una condición de entrada lógica, se define una clase de equivalencia válida y otra inválida.

**Análisis de valores límite:** El AVL es una técnica complementaria a la partición equivalente, básicamente, nos indica que si especificamos un rango delimitado de valores o un número de valores específicos, también se deberá probar por el valor inmediatamente superior e inmediatamente inferior de dichas cotas.

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 3. CASOS DE PRUEBA

- **RENDIMIENTO**

Las pruebas de rendimiento miden el tiempo que le ha tomado a la aplicación realizar una acción específica. Si bien esto depende también de manera importante de la máquina en donde se esté ejecutando la aplicación, sigue teniendo validez, ya que nos permite probar y controlar tanto el tiempo para un equipo concreto como realizar operaciones de diferentes formas y ver cuál conlleva un mejor rendimiento.

Para realizar estas mediciones, usaremos un sencillo cronómetro llamado **StopWatch**, disponible en **.NET**. Gracias a este cronómetro, podremos ver lo que ha tardado nuestra aplicación en ejecutar nuestros comandos mediante milisegundos, aunque también podríamos usar los *ticks* o **ciclos de proceso**.

Si se ha detectado una operación que conlleva mucho tiempo, habrá que particionar la operación en diversos cronómetros para identificar dónde se encuentra el problema e intentar solucionarlo.

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 3. CASOS DE PRUEBA

- **COHERENCIA**

Las pruebas de coherencia son subjetivas, es decir, no están ligadas a la propagación ni a los datos en sí mismos, se enfocan, entre otras cosas, en el estudio del flujo de trabajo (*workflow*) de la aplicación de un modo coherente.

Con ellas comprobamos si la funcionalidad de la aplicación es correcta y no si la aplicación funciona correctamente. Es decir, que una aplicación no tenga ningún error en el código o la interfaz no quiere decir que funcione correctamente. Por ejemplo, si desarrollamos una aplicación que tiene entre otros objetivos el envío de emails pero dicha funcionalidad no está contemplada en la aplicación, el programa no está funcionando correctamente.

Además de comprobar o aseguramos de que la aplicación realice todas las funciones requeridas y además de forma satisfactoria, también hay que tener en cuenta el ciclo de vida y compartimiento de los elementos que la componen. Básicamente, se trata de controlar que las operaciones que realiza la aplicación se hagan (o se permitan hacer) en el momento (o estado) apropiado. Si nuestra aplicación no conlleva el control de etapas éstas últimas valoraciones no serían necesarias, como podría ser el caso por ejemplo de una calculadora.

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 4. PRUEBAS UNITARIAS

Las pruebas unitarias son pruebas individuales para un método o clase, realizadas de manera sistemática a modo de batería de pruebas donde conocemos los datos de entrada y sabemos cuál sería el resultado esperado. Las pruebas unitarias pertenecen a los casos de prueba de caja blanca.

Las pruebas deberían implementarse de manera sistemática con cada nueva funcionalidad que se añada, teniendo de este modo las pruebas actualizadas en cada momento. Es importante que las pruebas se realicen y se ejecuten de manera incremental.

# DEPURACIÓN Y REALIZACIÓN DE PRUEBAS

## 4. PRUEBAS UNITARIAS

➢ NUnit

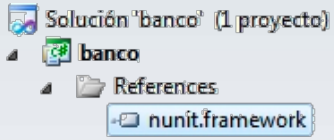
Para realizar estas pruebas en nuestras aplicaciones, vamos a utilizar **NUnit**. **NUnit** es una aplicación disponible como extensión que nos facilitará la integración con el entorno de desarrollo.

La instalación de **NUnit** nos registrará las *dlls* en nuestro sistema necesarias para utilizar **NUnit** en nuestros proyectos.

Para poder usar **NUnit** en nuestros proyectos, deberemos agregar una referencia a '*nunit.framework*' e incluir el espacio de nombres en nuestras clases *test*.

```
using System;
using NUnit.Framework;

namespace Banco
```




Mediante el uso de atributos de método, especificaremos que clases y que métodos serán nuestros métodos de prueba. Una vez especificados cuales son, implementaremos nuestras pruebas.

Las pruebas se realizan mediante **asertos**, básicamente consiste en establecer unos parámetros de entrada, salida o valor y la propiedad para la que queremos comprobar dichos valores.

Hay que tener presente que para que los asertos sean eficaces primero tendremos que desarrollar un algoritmo que nos lleve hasta el punto concreto en el que queremos hacer la prueba. Esto tiene un problema, ya que no siempre es viable que desde una clase se puedan acceder a todas las operaciones de los diferentes subsistemas.

Para solucionar éste problema se suele realizar un proyecto de pruebas o implementar un patrón fachada para la realización de las pruebas.