Universidad ORT Uruguay

Obligatorio 1
Diseño de Aplicaciones 2

Evidencia TDD y Clean Code

Martin Edelman - 263630

Tatiana Poznanski - 221056

Tomas Bañales - 239825

Profesores: Nicolás Fierro, Alexander Wieler, Marco Fiorito

2023

1. Link al repositorio

https://github.com/IngSoft-DA2-2023-2/263630_221056_239825.git

2. Aplicación TDD

Durante la mayor parte del desarrollo del sistema se siguió la metodología del Desarrollo Guiado por Pruebas (TDD), la cual significa que previo a la escritura de código de las funcionalidades se aplicaron pruebas automatizadas. Estas últimas se agruparon en diversos paquetes, según la distribución del proyecto. Asimismo, dichas pruebas aseguran que cada componente del sistema funciona como se espera y contribuyen a la prevención de errores en el desarrollo.

Además, el TDD brinda mayor confiabilidad y calidad en el código, dado que garantiza que todas las partes del proyecto se prueben correctamente y que cualquier cambio posterior pueda ser analizado rápidamente a través de las pruebas.

3. Estrategia

Nos basamos en la estrategia de TDD de "Outside-In" conocida como la Escuela de Londres. Para cada funcionalidad, seleccionamos el test que mejor describía el problema. Este enfoque, se centra en la experiencia del usuario final, empezando desde fuera del sistema y trabajando hacia adentro. Esto se traduce en una comprensión más profunda de las necesidades del usuario, interfaces de usuario amigables, retroalimentación continua, pruebas automatizadas y alineación con los objetivos.

4. Test Unitarios

A pesar de que teníamos un conjunto de pruebas implementadas en nuestro proyecto, las mismas no lograron proporcionar la cobertura adecuada debido a ciertos problemas inesperados. A medida que avanzamos en el desarrollo, descubrimos que las pruebas existentes no estaban capturando todos los posibles casos de uso o no estaban detectando algunos errores cruciales. Sin embargo, somos conscientes de la importancia de tener una buena cobertura de código para garantizar la calidad y la estabilidad a largo plazo de nuestro software.

Hierarchy	Covered (%Lines)	Not Covered (Lines)	Not Covered (%Lines)	Covered (Lines)
marti_NBK010105_2023-10-05.19_08_41.coverage	21,10%	2926	78,47%	787
dominio.dll	56,38%	41	43,62%	53
dataaccess.dll	0,37%	2398	99,63%	9
pruebas.dll	87,61%	58	12,18%	417
servicios.dll	73,06%	91	23,58%	282
api.dll	7,10%	338	92,35%	26

4.1 Data Access

La cobertura de código en el paquete DataAccess se ve reducida debido a la falta de pruebas en el contexto de Migrations. No pudimos realizar pruebas efectivas debido a problemas que enfrentamos con los mocks.

Hierarchy	Covered (%Lines)	Not Covered (Lines)	Not Covered (%Lines)	Covered (Lines)
marti_NBK010105_2023-10-05.19_08_41.coverage	21,10%	2926	78,47%	787
dominio.dll	56,38%	41	43,62%	53
dataaccess.dll	0,37%	2398	99,63%	9
{ } DataAccess	7,44%	112	92,56%	9
ECommerceContext	3,70%	26	96,30%	1
RepositorioCompra	0,00%	14	100,00%	0
RepositorioProducto	0,00%	56	100,00%	0
RepositorioUsuario	34,78%	15	65,22%	8
RepositorioProducto.<>c	0,00%	1	100,00%	0
{ } DataAccess.Migrations	0,00%	2286	100,00%	0

4.2 Dominio

No se realizaron pruebas explícitas en el paquete Dominio; en su lugar, evaluamos su funcionamiento a través de pruebas en Servicios y DataAccess

Hierarchy	Covered (%Lines)	Not Covered (Lines)	Not Covered (%Lines)	Covered (Lines)
marti_NBK010105_2023-10-05.19_08_41.coverage	21,10%	2926	78,47%	787
dominio.dll	56,38%	41	43,62%	53
{ } Dominio	46,67%	40	53,33%	35
Categoria	33,33%	4	66,67%	2
Color	66,67%	2	33,33%	4
Compra	47,06%	9	52,94%	8
Marca	0,00%	6	100,00%	0
Producto	65,63%	11	34,38%	21
QueryProducto	0,00%	8	100,00%	0
{ } Dominio.Usuario	94,74%	1	5,26%	18
Usuario	94,74%	1	5,26%	18

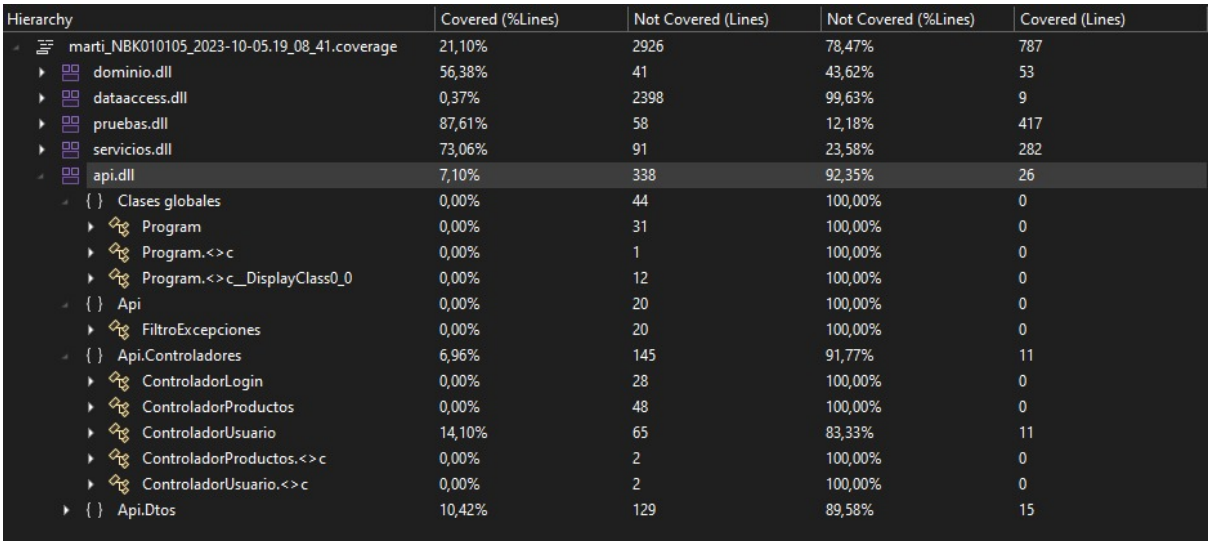
4.3 Servicios

En términos generales, la mayoría de las clases tienen una alta cobertura, a excepción de la clase de Productos, en la cual surgieron desafíos inesperados en las etapas finales del proyecto.

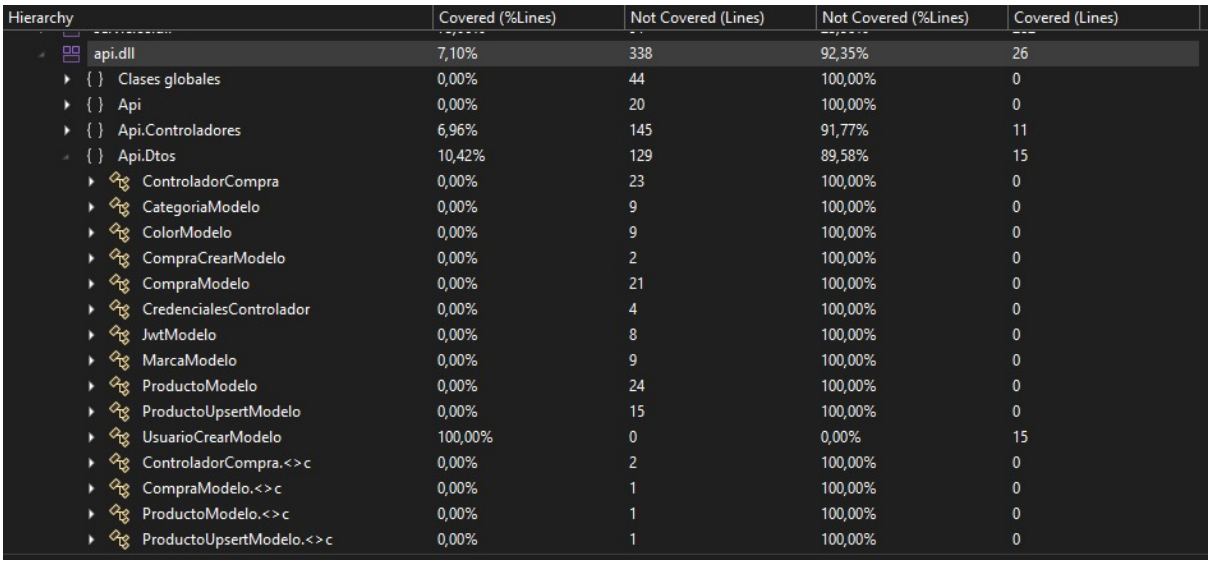
Hierarchy	Covered (%Lines)	Not Covered (Lines)	Not Covered (%Lines)	Covered (Lines)
marti_NBK010105_2023-10-05.19_08_41.coverage	21,10%	2926	78,47%	787
dominio.dll	56,38%	41	43,62%	53
dataaccess.dll	0,37%	2398	99,63%	9
pruebas.dll	87,61%	58	12,18%	417
servicios.dll	73,06%	91	23,58%	282
{ } Clases globales	0,00%	41	100,00%	0
ServicioProducto	0,00%	41	100,00%	0
{ } Servicios	73,43%	34	23,78%	105
ManejadorUsuario	73,77%	28	22,95%	90
PromocionContext	100,00%	0	0,00%	14
ServicioCompra	0,00%	6	100,00%	0
ManejadorUsuario.<>c__DisplayClass4_0	100,00%	0	0,00%	1
{ } Servicios.Promociones	87,62%	16	7,92%	177
Promocion20Off	96,77%	0	0,00%	30
Promocion3x1	91,30%	3	6,52%	42
Promocion3x2	89,74%	4	10,26%	35
PromocionTotalLook	80,25%	9	11,11%	65
Promocion20Off.<>c	100,00%	0	0,00%	2
Promocion3x1.<>c	100,00%	0	0,00%	1
Promocion3x2.<>c	100,00%	0	0,00%	2

4.4. Api

Enfrentamos dificultades al realizar las pruebas dado que ciertos resultados que estábamos tratando de evaluar eran nulos o carecían de información significativa, lo que dificultaba la validación de la funcionalidad.



Hierarchy	Covered (%Lines)	Not Covered (Lines)	Not Covered (%Lines)	Covered (Lines)
marti_NBK010105_2023-10-05.19_08_41.coverage	21,10%	2926	78,47%	787
dominio.dll	56,38%	41	43,62%	53
dataaccess.dll	0,37%	2398	99,63%	9
pruebas.dll	87,61%	58	12,18%	417
servicios.dll	73,06%	91	23,58%	282
api.dll	7,10%	338	92,35%	26
{ } Clases globales	0,00%	44	100,00%	0
Program	0,00%	31	100,00%	0
Program.<>c	0,00%	1	100,00%	0
Program.<>c__DisplayClass0_0	0,00%	12	100,00%	0
{ } Api	0,00%	20	100,00%	0
FiltroExcepciones	0,00%	20	100,00%	0
{ } Api.Controladores	6,96%	145	91,77%	11
ControladorLogin	0,00%	28	100,00%	0
ControladorProductos	0,00%	48	100,00%	0
ControladorUsuario	14,10%	65	83,33%	11
ControladorProductos.<>c	0,00%	2	100,00%	0
ControladorUsuario.<>c	0,00%	2	100,00%	0
{ } Api.Dtos	10,42%	129	89,58%	15



Hierarchy	Covered (%Lines)	Not Covered (Lines)	Not Covered (%Lines)	Covered (Lines)
api.dll	7,10%	338	92,35%	26
{ } Clases globales	0,00%	44	100,00%	0
{ } Api	0,00%	20	100,00%	0
{ } Api.Controladores	6,96%	145	91,77%	11
{ } Api.Dtos	10,42%	129	89,58%	15
ControladorCompra	0,00%	23	100,00%	0
CategoriaModelo	0,00%	9	100,00%	0
ColorModelo	0,00%	9	100,00%	0
CompraCrearModelo	0,00%	2	100,00%	0
CompraModelo	0,00%	21	100,00%	0
CredencialesControlador	0,00%	4	100,00%	0
JwtModelo	0,00%	8	100,00%	0
MarcaModelo	0,00%	9	100,00%	0
ProductoModelo	0,00%	24	100,00%	0
ProductoUpsertModelo	0,00%	15	100,00%	0
UsuarioCrearModelo	100,00%	0	0,00%	15
ControladorCompra.<>c	0,00%	2	100,00%	0
CompraModelo.<>c	0,00%	1	100,00%	0
ProductoModelo.<>c	0,00%	1	100,00%	0
ProductoUpsertModelo.<>c	0,00%	1	100,00%	0

5. Mocks

Al utilizar mocks y pruebas unitarias, para desarrollar el código nos centramos en la interacción con otras partes del sistema en lugar de depender de cómo están implementadas esas partes en particular. Esto conduce a un bajo acoplamiento entre las clases y sus dependencias, lo que significa que las clases no están fuertemente conectadas entre sí a nivel de código.

Este enfoque es útil cuando las clases dependen de recursos externos, como bibliotecas de terceros, archivos o bases de datos. Con la implementación mocks, es posible simular dichas dependencias externas en nuestras pruebas, lo que permite, al

mismo tiempo, verificar que la clase se comunica de manera adecuada con estos recursos sin necesidad de interactuar con las implementaciones reales. Esto hace que el código sea más flexible y menos propenso a problemas cuando las implementaciones externas cambian con el tiempo.

```
[TestMethod]
[ExpectedException(typeof(NullReferenceException))]
public void AplicarPromocionErrorNulo()
{
    //Act
    mock!.Setup(x => x.AplicarPromocion(It.IsAny<List<Producto>>())).Returns(3500);
    carrito!.Remove(producto2!);
    carrito!.Add(productoVacio!);
    int costoTotal = promocion3x1!.AplicarPromocion(carrito!);
}
```

6. Clean Code

A lo largo del desarrollo de nuestro ECommerce, optamos por emplear los principios de Clean Code por varias razones fundamentales muy beneficiosas.

En primer lugar, el Clean Code garantiza una fácil comprensión del código fuente, ya que tanto los nombres de las variables, como de las funciones y de las clases son nemotécnicos. A su vez, mantener el código limpio asegura su mantenibilidad y posteriores modificaciones y ello, junto con escribir funciones y métodos concretos y claros, evita errores inesperados.

Por otra parte, contribuye a poder probar el código fácilmente. Los métodos y clases bien organizados permiten la implementación de pruebas unitarias y de integración de manera más efectiva. Esto asegura que el software sea robusto y resistente a fallos. Asimismo, se promueve la reutilización del código,

A su vez, al adoptar la convención CamelCase para los nombres de variables y métodos, se logra una buena legibilidad del código. Esto facilita una distinción clara entre las palabras, lo que a su vez simplifica la comprensión del código. Además, el código se integra sin inconvenientes, permitiendo a otros entenderlo fácilmente y colaborar de manera eficaz. Al mismo tiempo, esta práctica posibilita una identificación rápida y sencilla de variables, métodos y propiedades, lo que agiliza el análisis del código y la comprensión de las diversas partes del sistema y sus funcionalidades.