Universidad ORT Uruguay

Facultad de Ingeniería

Primer obligatorio

Diseño de Aplicaciones II

Entregado como requisito para la obtención del título de

Licenciatura en Sistemas

Guillermo Diotti - 285578

Juan Peyrot – 275076

Nicolás Toscano - 220264

Profesores:

Daniel Acevedo

Pablo Geymonat

https://github.com/IngSoft-DA2/285578-220264-275076

2024

Índice

Descripción de la estrategia de TDD seguida

La estrategia seguida por el equipo fue la de TDD Outside-In, lo que significa que el desarrollo de la aplicación comenzó desde el nivel más externo (el que interactúa directamente con el usuario o cliente) y avanzó gradualmente hacia los componentes internos y más profundos del sistema. A diferencia de la estrategia clásica de TDD "Inside-Out", que comienza probando componentes internos y construyendo hacia afuera, el enfoque Outside-In permite diseñar el sistema en función de cómo se espera que interactúe con el mundo exterior.

Este enfoque nos permitió que el desarrollo esté impulsado por las necesidades del usuario final, comenzando con las pruebas de los controladores y otros puntos de entrada de alto nivel del sistema. Una vez teníamos claro el propósito del problema, pudimos pasar nuestras ideas y diagramas de la solución a código. Una vez definidas las expectativas externas del comportamiento del sistema, el equipo avanzó hacia los componentes internos como los repositorios y la capa de acceso a los datos correspondiente a la base de datos.

En otros términos, este enfoque nos ayudó a primero entender el problema y luego implementar una solución basado en las necesidades del cliente.

Los tests se almacenan en un proyecto llamado “Homify.Tests” el cual contiene las pruebas unitarias correspondientes a todas las partes de la aplicación, separadas por carpetas.

- *ControllerTests*: Pruebas de los controladores de la WebApi.

- *FilterTests*: Pruebas de los filtros de la WebApi.

- *RepositoryTests*: Pruebas de los repositorios ubicados en Homify.DataAccess.

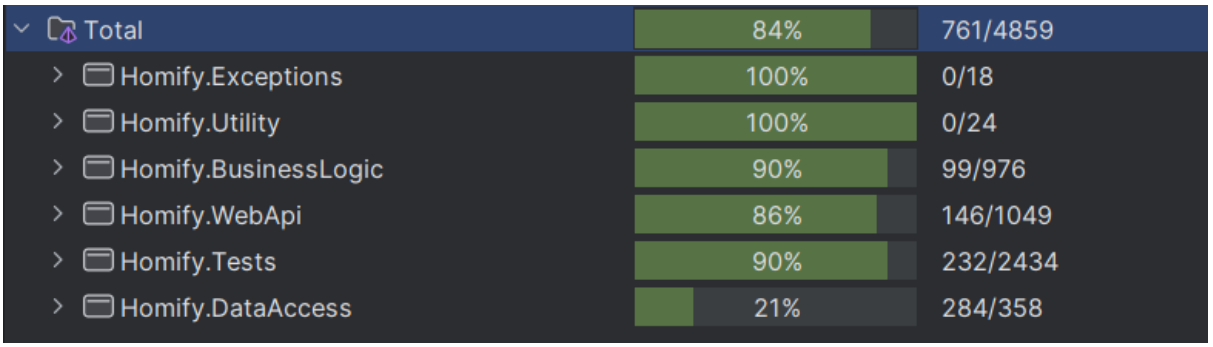
- *ServiceTests*: Pruebas de los servicios de lógica de negocio.

El proyecto usa *MSTest, Fluent Assertions y Mocks (Moq)* en las pruebas unitarias. Los Mocks son objetos simulados que se utilizan en lugar de dependencias reales del sistema durante la fase de pruebas. Este uso fue particularmente útil, ya que permitió que las pruebas fueran aún más unitarias y precisas al no depender de la implementación o del estado actual de otros componentes del sistema. Esto permitió que el equipo diseñara las interfaces necesarias sin preocuparse aún por la lógica interna.

Informe de cobertura para todas las pruebas desarrolladas

El uso de TDD ayudó a alcanzar una cobertura de código muy alta, lo que asegura que la mayoría de las rutas y funcionalidades de la API han sido probadas. Aunque el porcentaje de cobertura por sí solo no garantiza la calidad del código, es un buen indicador de que muchas rutas y comportamientos han sido verificados.

En nuestro proyecto, se alcanzó más del 90% de cobertura en todas las partes de la aplicación, lo cual es un excelente resultado. Esto fue posible gracias al enfoque disciplinado de TDD, que asegura que cada nueva pieza de código tenga pruebas que la validen al momento de ser creada.



Como podemos observar, la cobertura total es menor a 90%, al igual que **Homify.WebApi** (86%) y **Homify.DataAccess** (21%).

Sin embargo, estos valores no son absolutos, puesto que no se están excluyendo archivos que perjudican a los valores de cobertura y que no forman parte del dominio que se comprende probar mediante las pruebas unitarias.

En primer lugar, en Homify.WebApi presentamos un 86% de cobertura, sin embargo, en este paquete se encuentra el archivo Program.cs, el cual no tiene sentido testear.

Si hacemos los cálculos, al no tomar en cuenta este archivo, la cantidad de casos no cubiertos en el paquete decrecería de *146 a 94 (146-52)*, al igual que los casos totales bajarían de *1049 a 997*. Por ello, reformulando la operación, 94 líneas no cubiertas de un total de 997, nos da aproximadamente *9.4% de líneas sin cubrir*, por ende la cobertura real en el paquete es de aproximadamente **91%**.

Siguiendo con **Homify.DataAccess**, este paquete también tiene un índice elevado de cobertura, pero se ve muy perjudicado ya que contiene la carpeta de las migraciones correspondientes a la base de datos, por lo que hay una excesiva cantidad de datos que no deben ser testeados que están afectando los valores de cobertura obtenidos.

Aplicando las mismas operaciones que en el caso anterior, si quitamos las 282 líneas no cubiertas de la carpeta de Migrations, esto nos dejaría con un *balance de 2 líneas sin cubrir de un total de 76*, lo que representa aproximadamente un contundente **97%** de cobertura final.

Si trasladamos estas operaciones descritas anteriormente al balance total de cobertura, los números finales serían *427 líneas no cubiertas de 4525 totales*, lo que equivale a un aproximado **91% de cobertura total**.

Estos resultados obtenidos son en gran parte gracias a nuestro seguimiento de la metodología Test Driven Development, el cual nos permitió ir desarrollando las features necesarias, al mismo tiempo que íbamos generando cobertura y asegurándonos en gran parte que muchas de las características que estábamos desarrollando se comportasen de una manera controlada.

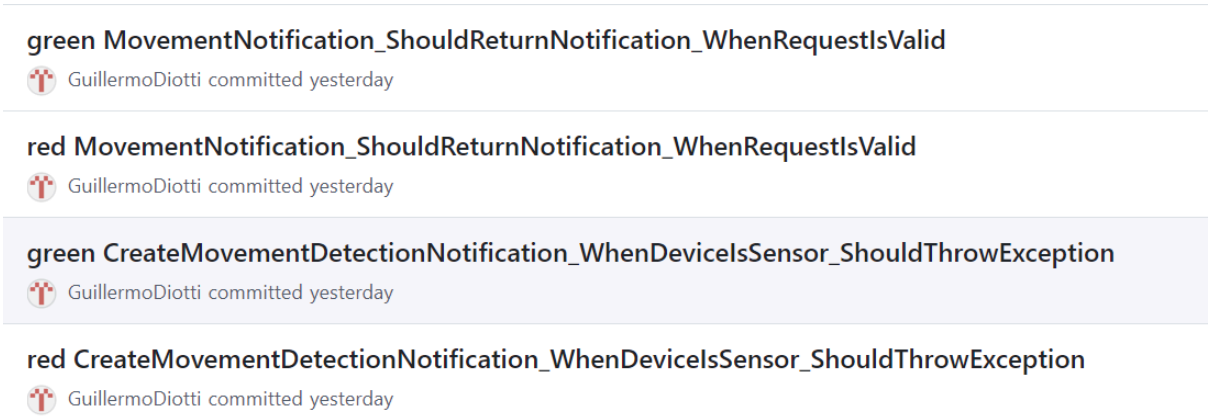
Si bien el término de cobertura no garantiza por sí solo que todo funcione a la perfección (o al menos como uno espera), si nos permite tener bajo control un gran abanico de casos cubiertos para nuestra tranquilidad, lo que nos indica que para muchos escenarios posibles, el sistema responde a lo que nosotros queremos.

Como pequeña aclaración, queremos notificar que sí en alguna parte se encuentra algún atributo **[ExcludeFromCodeCoverage]**, se deberá únicamente para poder realizar el Pull Request a nuestras ramas protegidas (develop y main) con el fin de realizar la entrega en tiempo y forma acordadas, ya que así se nos solicitó que debíamos cumplir con estos estándares. De igual forma, los datos desglosados previamente *no han excluído ningún archivo*, por lo que no afectan al argumento brindado.

Detalle de evolución de TDD para features marcadas con (*)

El desarrollo de la aplicación se desarrolló en su totalidad bajo el seguimiento de TDD, sin embargo, para algunas funcionalidades específicas, se requería ser más cauteloso y preciso, detallando las diferentes fases en commits independientes.

Detección de movimiento



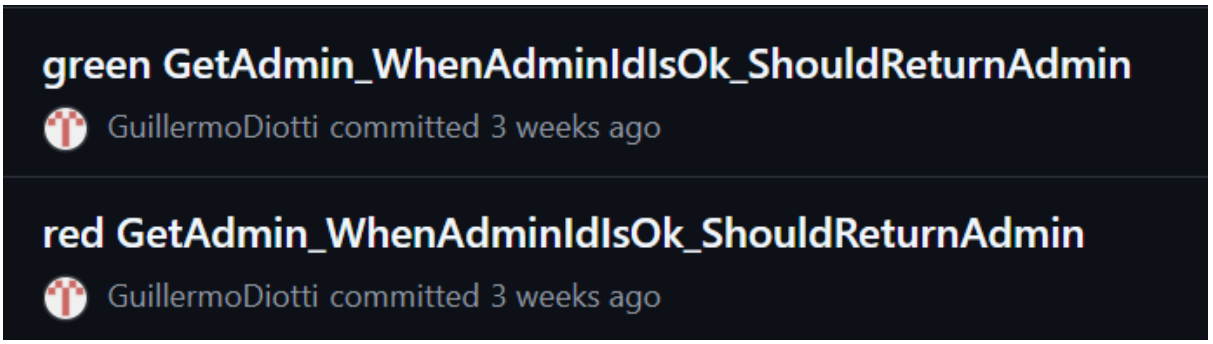
Creación de una empresa

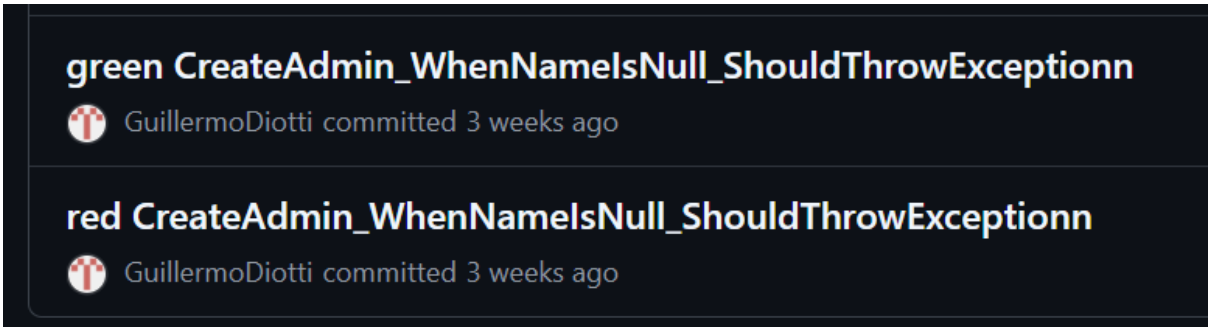
Si bien en este caso no aparece una fase “green”, se debe a que a la hora de commitearla hubo un desacierto con el comando *git commit*, en el cual faltaron las comillas (“”) para el parámetro *-m*, por lo que no se guardó correctamente, y tanto fase green como refactor quedaron guardadas en el último commit.

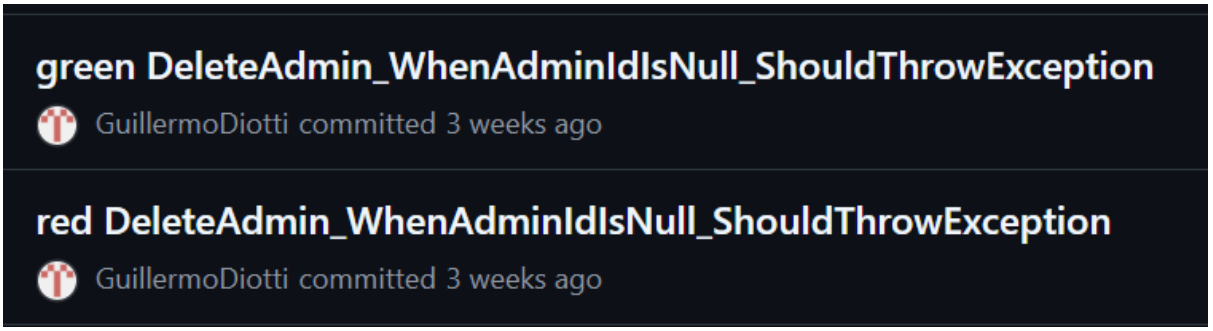
```
PS C:\Users\Juan\Desktop\285578-220264-275076> git add .
PS C:\Users\Juan\Desktop\285578-220264-275076> git commit -m create company green
```

Recibimos autorización por parte del docente para documentar y no tener que hacer toda la feature desde cero, lo cual agradecemos.

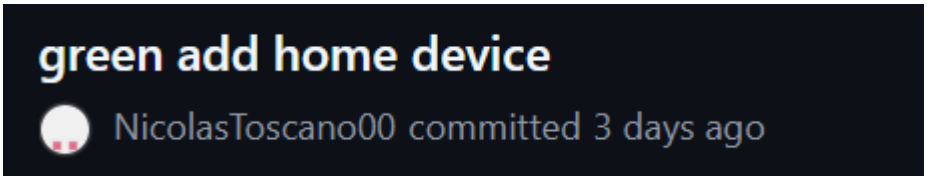
Mantenimiento de cuentas de administrador







Asociar dispositivos al hogar



En este caso particular, debido a la cantidad de requerimientos y algunas dificultades que tuvimos en un principio con las ramas, base de datos y demás, pasamos por alto realizar el primer commit en estado red, aunque la metodología fue aplicada de igual manera como a lo largo de todo el proyecto.

Principios de Clean Code aplicados

En el desarrollo de una API, es esencial garantizar que el código sea legible, mantenible y extensible a largo plazo. Para lograr esto, implementamos los principios de **Clean Code**, que aseguran que el código pueda ser comprendido por todos los miembros del equipo y se mantenga libre de deuda técnica. A continuación, se describe cómo se aplicaron estos principios en la creación de nuestra API, siguiendo buenas prácticas y respetando estándares de nomenclatura, estructura de métodos y organización de pruebas unitarias.

1. Estándares de nomenclatura

Uno de los pilares fundamentales de Clean Code es la **nomenclatura clara y coherente**. Para asegurar la legibilidad y mantener un estilo uniforme en todo el código, seguimos las siguientes reglas:

Métodos y propiedades públicas comienzan con mayúscula. Esto facilita identificar rápidamente los elementos accesibles desde fuera de la clase.

```
public class User
{
    public string Name { get; set; } = null!;
    public string Email { get; set; } = null!;
    public string Password { get; set; } = null!;
    public string LastName { get; set; } = null!;
    public string Id { get; init; }
    public string? ProfilePicture { get; set; } = null!;
    public Role Role { get; init; } = null!;
    public string RoleId { get; set; } = null!;
    public DateTimeOffset CreatedAt { get; init; }
```

Propiedades privadas comienzan con un guion bajo (_), diferenciándose visualmente de las variables públicas y reflejando su visibilidad de manera concisa.

```
[TestClass]
public class UserControllerTests
{
    private readonly AdminController _controller;
    private readonly Mock<IUserService> _userServiceMock;
    private readonly Mock<IRoleService> _roleServicemock;
```

Este estándar no solo hace que el código sea más legible, sino que también refleja por sí solo la visibilidad de cada elemento del sistema.

2. Nombres claros y concisos

Otro principio clave aplicado es la elección de nombres **claros y descriptivos** para los métodos y variables. En Homify, cada nombre de método y variable refleja directamente su propósito, lo que mejora la **autodescripción del código**. Esto significa que, sin necesidad de revisar comentarios o documentación adicional, un desarrollador puede entender qué hace una función solo con leer su nombre.

Por ejemplo:

Los métodos de prueba siguen el formato `[Method]When[Scenario]Should[ExpectedBehavior]`, como `CreateUser_WhenNameIsNull_ShouldThrowException`. Esto hace evidente qué se está probando y bajo qué condiciones.

En lugar de utilizar nombres ambiguos o genéricos como, se emplean nombres descriptivos. Por ejemplo, todas las clases encargadas de manejar lógica de negocio siguen el patron [Resource]Service (UserService, HomeService, etc), lo que mejora significativamente la comprensión del código. Lo mismo ocurre con la capa de acceso a datos y controladores.

3. Métodos con pocos parámetros

Otro principio clave fue **limitar el número de parámetros** que recibe cada método. Según Clean Code, los métodos deben recibir entre 0 y 3 parámetros como máximo. Esto reduce la complejidad de los métodos y facilita la comprensión del código, desacoplando a las invocadoras de procesar todos los parámetros correctamente antes de hacer un llamado.

El ejemplo más claro se puede apreciar con los DTO´s de respuestas de la WebApi, ya que estos reciben un objeto, y lo adecúan por si mismos al formato de respuesta requerido, para asegurarse de no brindar información sensible o innecesaria al cliente. Esto también permite que el mantenimiento sea más fácil, ya que si en el futuro se desean mostrar otras cosas, la responsabilidad será solo de esa clase, y no afectará a quienes la usen.

```
public class CreateHomeResponse
{
    public string Id { get; set; }
    public CreateHomeResponse(Home home)
    {
        Id = home.Id;
    }
}
```

4. Organización de las pruebas unitarias

Las pruebas unitarias son esenciales para asegurar la calidad del software, y su correcta organización es fundamental para mantener su claridad. Aplicamos la estructura **Arrange, Act y Assert (AAA)** en todas las pruebas unitarias de la API:

Arrange: Configuración del escenario de prueba, inicializando los objetos necesarios.

Act: Ejecución del método que se va a probar.

Assert: Verificación de que los resultados obtenidos son los esperados.

Esta separación clara entre las etapas de una prueba facilita su comprensión y depuración en caso de que falle. Además, para mantener las pruebas bien organizadas, se utilizaron **regiones** para dividirlas según su funcionalidad y comportamiento, lo que permite navegar más fácilmente por el código de pruebas.

```
[TestMethod]
public void GetByUserId_ShouldReturnCompany_WhenCompanyExists()
{
    var userId = "test-user-id";
    var expectedCompany = new Company
    {
        Id = Guid.NewGuid().ToString(),
        Name = "Test Company",
        OwnerId = userId
    };
    _companyRepositoryMock.Setup(repo => repo.Get(It.IsAny<Expression<Func<Company, bool>>>()))
        .Returns(expectedCompany);
    var result = _service.GetByUserId(userId);
    Assert.IsNotNull(result);
    Assert.AreEqual(expectedCompany, result);
}
```

También es de aclarar que en aquellas clases de testing donde había mucha cantidad de métodos, se las dividió por regiones para aumentar la legibilidad y acceso a las mismas.

5. Métodos reducidos y divididos por responsabilidad

Intentamos mantener siempre el *Principio Kiss*, por lo que separamos las responsabilidades a la hora de operar las distintas requests recibidas, intentando que cada función se encargara de lo más simple posible.

```
[HttpPost]
[AuthenticationFilter]
[AuthorizationFilter(PermissionsGenerator.CreateHome)]
public CreateHomeResponse Create(CreateHomeRequest request)
{
    if (request == null)
    {
        throw new NullRequestException("Request can not be null");
    }

    var owner = GetUserLogged() as HomeOwner;
    var arguments = new CreateHomeArgs(
       request.Street ?? string.Empty, request.Number ?? string.Empty, request.Latitude ?? string.Empty,
       request.Longitud ?? string.Empty, request.MaxMembers, owner);

    var homeSaved = _homeService.AddHome(arguments);
    return new CreateHomeResponse(homeSaved);
}
```

Por ejemplo, en este método encargado de crear un hogar, nos aseguramos de hacer todas las validaciones correspondientes, delegando las responsabilidades a quienes cuentan con la información necesaria para manejarlas. El método *Create* es el punto de entrada de las peticiones, se encarga de recibir una request, y brindar una respuesta. Las validaciones de formato (datos incorrectos) son delegadas a *CreateHomeArgs*, quien tirará excepciones que serán controladas por el ExceptionFilter si encuentra valores incorrectos.

La clase inyectada *HomeService* se encarga de llevar a cabo validaciones más cercanas a la lógica de negocio (datos repetidos, etc) y además delega la creación de la entidad a la base de datos al repositorio correspondiente.

Finalmente, se devuelve el objeto creado en base al modelo de respuesta *CreateHomeResponse* quien se encarga de recibir la entidad completa, y decidir qué datos debe mostrar al cliente y cuales preservar.

De esta manera, logramos respetar *SRP*, y dividir los métodos por la responsabilidad que ostentan, haciendo que mantener y escalar el proyecto a futuro, sea un proceso mucho más claro y sencillo, puesto que si se deben introducir modificaciones, solo debemos de acudir a la clase o función encargada de ella, sin preocuparnos por quienes las llaman o dependen de su resultado.