

Universidad ORT Uruguay

Facultad de Ingeniería

Documentación Obligatorio 1 Evidencia de la aplicación de TDD y Clean Code

Diseño de Aplicaciones 2

Link al Repositorio: IngSoft-DA2/301178 231810 280070 (github.com)

Integrantes:

- Angelina Maverino 280070
- Yliana Otero 301178
- María Belén Rywaczuk 231810

Tutores:

- Juan Ignacio Irabedra De Maio
- Juan Pablo Barrios García
- Ignacio Valle Dubé

ÍNDICE

Descripción de la estrategia de TDD seguida (inside – out o outside - in).	
Informe de cobertura para todas las pruebas desarrolladas.	5
Funcionalidades especificadas como prioritarias (*) evidencia mediante los commits de se aplicó TDD mostrando la evolución del ciclo de TDD	que 6
Mantenimiento de cuentas de administrador	6
Creación de una empresa	8
Evidencia de TDD en service:	10
Evidencia de TDD en controller:	10
Detección de movimiento	10
Aquí se incorpora la funcionalidad inicial. Sin embargo, posteriormente realizamos una	
refactorización al identificar que se añadirían otras funcionalidades.	11
Asociar dispositivos al hogar	12

Descripción de la estrategia de TDD seguida (inside – out o outside - in).

En el desarrollo guiado por pruebas (TDD), hay dos enfoques principales: inside-out y outside-in:

Inside-Out:

Comienzas desarrollando las partes más internas o fundamentales del sistema, como los modelos o los servicios.

Luego, vas construyendo el resto de la aplicación, conectando estas capas internas hacia el exterior (controladores, interfaces de usuario).

Ejemplo: En una API, primero escribirías pruebas para los métodos de servicio o lógica de negocio, y después para los controladores.

Outside-In:

Empiezas con los componentes más externos (como la interfaz de usuario o los controladores), escribiendo pruebas que describen cómo se espera que funcione la aplicación desde una perspectiva de usuario.

A partir de ahí, vas construyendo las capas internas necesarias para que la funcionalidad pase las pruebas externas.

Ejemplo: Primero creamos una prueba para verificar si una llamada a un endpoint de la API devuelve la respuesta esperada, y luego implementar la lógica del servicio para pasar esa prueba.

Nuestra estrategia

Para el desarrollo de la API, comenzamos creando el **domain**, lo cual nos permitió definir claramente la lógica de negocio y las entidades fundamentales sin depender de la infraestructura o la interfaz externa. Esto facilitó la evolución y reutilización del código en futuras implementaciones.

Nos reunimos para elaborar un Excel con la lista de endpoints, lo cual nos proporcionó una visión clara de lo que íbamos a implementar y de los recursos necesarios para lograrlo. Aunque los endpoints se desarrollaron al final del proceso, esta planificación anticipada nos permitió estructurar mejor el proyecto y priorizar las tareas.

Primero nos enfocamos en implementar los **services** y **models**, lo que permitió una base sólida antes de crear los endpoints. Por lo tanto, seguimos un enfoque **inside-out**, donde la lógica interna se desarrolla primero y luego se conecta con los controladores y puntos de entrada.

Uso de mocks

En nuestro proyecto utilizamos **mocks** para simular el comportamiento de dependencias externas durante las pruebas unitarias, lo que nos permite centrarnos en probar el código en sí, sin depender de la implementación real de esas dependencias. Usamos la biblioteca **Moq** para crear estos mocks de manera sencilla.

Los mocks se generan definiendo el comportamiento esperado de ciertos métodos o propiedades de una clase o interfaz que queremos simular. Por ejemplo, si un servicio depende de una base de datos o una API externa, en lugar de acceder a la implementación real (que podría ser lenta o no estar disponible), creamos un mock que "imita" ese comportamiento con respuestas predefinidas. Esto nos permite aislar y probar el código bajo condiciones controladas.

Usamos **Moq** en nuestro proyecto principalmente para simular servicios como el acceso a datos o la lógica de negocio. Esto nos asegura que las pruebas sean consistentes y rápidas, y que se puedan ejecutar en cualquier entorno sin depender de conexiones reales o datos externos.

Además, los mocks son útiles para **test-driven development (TDD)**, ya que permiten validar los comportamientos esperados de las dependencias desde el inicio del desarrollo, antes de que las implementaciones reales estén completas.

Informe de cobertura para todas las pruebas desarrolladas.

Dado que todo el desarrollo se realizó siguiendo la metodología TDD (Desarrollo Guiado por Pruebas), garantizamos que la cobertura de las pruebas unitarias alcanzara entre el 90% y el 100%. A continuación, se muestra una imagen que detalla el nivel de cobertura logrado.

√ Total	79%	894/4	
> IDataAccess	100%	0/7	
> CustomExceptions	100%	0/15	
> TestDataAccess	99%	1/131	
> TestWebApi	99%	8/647	
> Domain	97%	10/392	
>	93%	36/539	
> TestDomain	92%	37/476	
> TestService	91%	48/547	
> BusinessLogic	90%	32/335	
>	60%	156/389	
> DataAccess	23%	557/724	
> ServiceFactory	0%	9/9	
✓ ■ WebApi	60%	156/389	
∨ ⟨⟩ WebApi	64%	131/364	
> () Controllers	99%	1/234	
> () Filters	0%	121/121	
> () Attributes	0%	9/9	
> 🔩 Program	0%	25/25	
∨ □ DataAccess	23%	557/724	
∨ ⟨⟩ DataAccess	23%	557/724	
> 🔩 SqlRepository <t></t>	100%	0/84	
> 🔩 SmartHomeContext	83%	17/100	
> () Migrations	0%	540/540	

Como se puede observar, todos los puntos alcanzan entre un 90% y 100% de cobertura. Es importante señalar que no se realizaron pruebas para las migraciones en la capa de acceso a datos (Data Access), ni para los filtros y atributos (Filter y Attributes).

Funcionalidades especificadas como prioritarias (*) evidencia mediante los commits de que se aplicó TDD mostrando la evolución del ciclo de TDD

Nuestro flujo de trabajo comenzó desarrollando las clases del dominio, seguido de los servicios que considerábamos necesarios para el endpoint a implementar, y finalmente procedimos con el desarrollo del propio endpoint. Este enfoque nos permitió ir de lo más específico a lo más general o abstracto, brindándonos una visión más clara de las necesidades del proyecto y permitiéndonos trabajar de manera organizada.

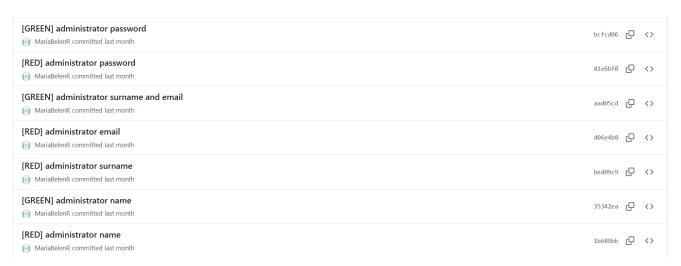
Todo este proceso se realizó utilizando Test-Driven Development (TDD). Prestamos especial atención a las funcionalidades marcadas con un asterisco, aunque todas las funcionalidades fueron implementadas bajo TDD. El proceso comenzaba escribiendo las pruebas unitarias utilizando mocks, lo que correspondía a la etapa RED del ciclo TDD. Luego, implementamos la funcionalidad esperada en la etapa GREEN, asegurándonos de cumplir con los requisitos. En muchos casos, las etapas GREEN y de refactorización se realizaron en el mismo commit, salvo algunas excepciones.

A lo largo del proyecto, realizamos commits separados para distinguir entre la etapa RED y la etapa GREEN/refactorización, evidenciando el uso de TDD. A continuación, se adjuntan ejemplos de commits con imágenes y descripciones que detallan el flujo de trabajo seguido.

Mantenimiento de cuentas de administrador

Pruebas de uso de TDD en dominio Administrator:

DOMAIN

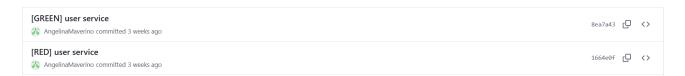


Ejemplo de commit [RED] (se realiza el Test que va estar en rojo ya que la funcionalidad todavía no está realizada)

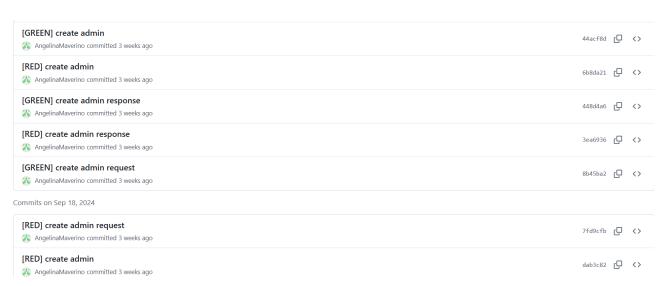
```
✓ SmartHome/DomainTest/AdministratorTest.cs □
                                                                                                              +10 000000 ...
••• @@ -0,0 +1,19 @@
    1 + namespace DomainTest;
    2 +
    3 + [TestClass]
    4 + public class AdministratorTest
    5 + {
    6 + [TestMethod]
    7 + public void TestAddNameToAdministrator()
    8 + {
    9 +
               // Arrange
    10 +
               Administrator administrator = new Administrator();
    11 +
    12 +
              // Act
    13 +
              administrator.Surname = "Juan";
    14 +
    15 +
               // Assert
    16 +
                Assert.AreEqual("Juan", administrator.Surname);
    17 +
    18 +
    19 + }
```

Ejemplo de commit [GREEN] (se implementa la funcionalidad mínima para que el test pase, luego se refactoriza)

Se creó el UserService, el cual se conecta con el repositorio de usuarios. Para agregar o eliminar un administrador, se deben seguir procedimientos específicos dentro de este servicio.



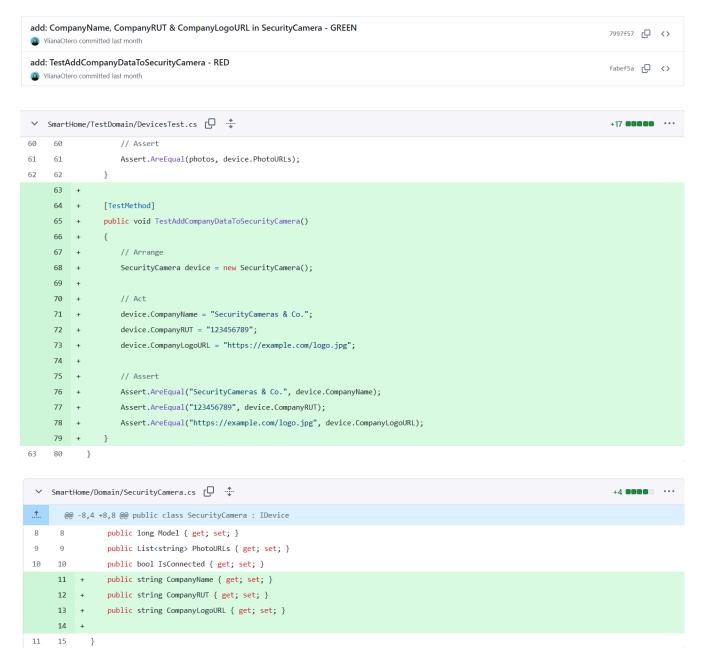
Se realizó el endpoint para agregar un administrador



Creación de una empresa

DOMAIN

Aquí hay un ejemplo claro en donde realizamos REFACTOR, ya que mientras hacíamos device, nos dimos cuenta que sería mejor tener una clase Company, en vez de tener los atributos como nombre de company en la clase de Device.

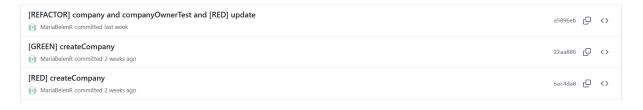


Aquí es donde se hace el REFACTOR:

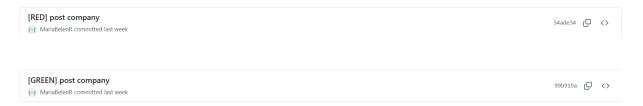


```
∨ SmartHome/Domain/Company.cs □
                                                                                                                          +9 00000 ...
... @@ -0,0 +1,9 @@
   1 + namespace Domain;
     3 + public class Company
     4 + {
     5  + public string Name { get; set; }
6  + public string RUT { get; set; }
              public string LogoURL { get; set; }
     9 + }
                                                                                                                         +1 -3 •0000 •••
∨ SmartHome/Domain/SecurityCamera.cs 📮 💠
     @@ -8,8 +8,6 @@ public class SecurityCamera : IDevice
public List<string> PhotoURLs { get; set; }
10 10
             public bool IsConnected { get; set; }
    - public string CompanyName { get; set; }
11
    public string CompanyRUT { get; set; }public string CompanyLogoURL { get; set
12
               public string CompanyLogoURL { get; set; }
    11 + public Company Company { get; set; }
```

Evidencia de TDD en service:

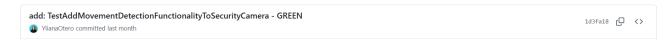


Evidencia de TDD en controller:

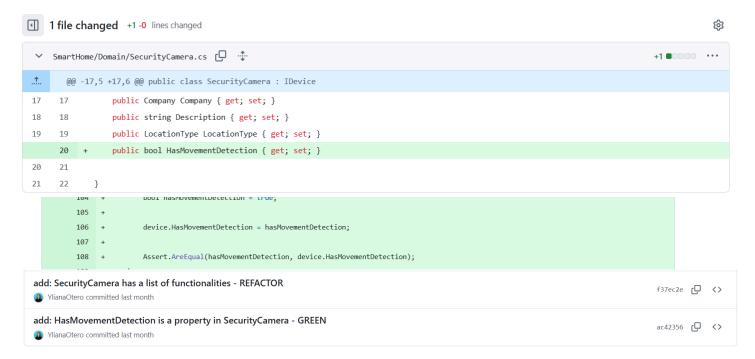


Detección de movimiento

DOMAIN



Nota: Aquí hay un leve error de tipeo, este commit se refiere a [RED]

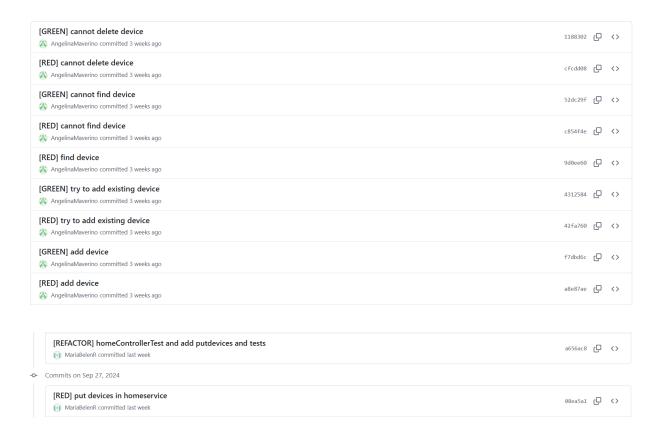


Aquí se incorpora la funcionalidad inicial. Sin embargo, posteriormente realizamos una refactorización al identificar que se añadirían otras funcionalidades.

```
SmartHome/TestDomain/DevicesTest.cs
                                                                                                                                       +3 -3 
        @@ -101,11 +101,11 @@ public void TestAddOutdoorLocationTypeToSecurityCamera()
....
      101
                   [TestMethod]
101
102
      102
                   public void TestAddMovementDetectionFunctionalityToSecurityCamera()
103
      103
                       bool hasMovementDetection = true;
104
                       List<string> functionalities = new List<string> { "Movement Detection" };
      104
105
      105
                       device.HasMovementDetection = hasMovementDetection;
106
                       device.Functionalities = functionalities;
      106
107
                       Assert.AreEqual(hasMovementDetection, device.HasMovementDetection);
108
      108
                       Assert.AreEqual(functionalities, device.Functionalities);
109
      109
                   }
110
      110
111
```

Asociar dispositivos al hogar

Aquí se muestra cómo se crea un test para cada flujo, incluyendo los casos en los que se lanzan excepciones.



Aquí se adjunta la parte [RED] con la [GREEN] en el service.

```
✓ SmartHome/TestService/HomeServiceTest.cs ☐ 
                                                                                                                          +20 •••• •••
    211 + [TestMethod]
    212 +
                public void TestPutDevicesInHome()
    213 +
    214 +
                   HomeService homeService = new HomeService(_mockHomeRepository.Object);
                                                                                                                                      +
    215 +
                 Home home = new Home(homeOwnerId,Street, DoorNumber, Latitude, Longitude);
    216 +
                   _mockHomeRepository.Setup(x=>x.GetById(1)).Returns(home);
    217 +
    218 +
                   List<Device> homeDevices = new List<Device>
    219 +
    220 +
                       new WindowSensor
    221 +
    222 +
                          Id = 1, Name = "Sensor de ventana", Model = 456, Description = "Sensor para ventanas", IsConnected = false ,
    223 +
                       new SecurityCamera { Id = 1, Name = "Cámara de seguridad", Model = 123, Description = "Cámara para exteriores",
    224 +
           IsConnected = true }
    225 +
    226 +
    227 +
                   homeService.PutDevicesInHome(1, homeDevices);
                   Assert.AreEqual(home.Devices,homeDevices);
    229 + }
    230 +
```

```
+14 00000 ...
.... @@ -75,4 +75,18 @@ public Home GetHomeById(long id)
75
76 76
               return home;
    78 +
    79 +
           public Home PutDevicesInHome(long homeId, List<Device> homeDevices)
    80 +
    81 +
               Home home = homeRepository.GetById(homeId);
    83 +
    84 +
                  throw new ElementNotFound(HomeNotFoundMessage);
    85 +
    87 +
                home.Devices = homeDevices;
    88 +
    89 +
               homeRepository.Update(home);
    91 +
78 92 }
```

Aquí se adjunta la parte [RED] con la [GREEN] en el endpoint.

```
✓ SmartHome/TestWebApi/Controllers/HomesControllerTest.cs ☐ 
                                                                                                                                  +15 00000 ...
      @@ -169,6 +169,21 @@ public void TestGetMembersFromHomeNotFoundStatusCode()
169 169
170 170
                     Assert.IsInstanceOfType(result, typeof(NotFoundObjectResult));
171 171
     172 +
                 [TestMethod]
     173 +
      174 +
                 public void TestPutDevicesInHomeOkStatusCode()
      175 +
      176 +
                      \label{tomeDevicesRequest} \mbox{HomeDevicesRequest request = } \mbox{${\sf new}$ HomeDevicesRequest()$}
      177 +
     178 +
                      Devices = new List<DeviceRequest>(),
     179 +
                       WindowSensors = new List<WindowSensorRequest>(),
      180 +
                        SecurityCameras = new List<SecurityCameraRequest>()
      181 +
      183 +
                     ObjectResult? result = _homeController.PutDevicesInHome(request) as OkObjectResult;
      184 +
      185 +
                      Assert.AreEqual(200, result!.StatusCode);
172
      187
173
     188
                  private HomeResponse DefaultHomeResponse()
```

```
+9 -1 •••• •••
.<u>†</u>..
     @@ -74,5 +74,13 @@ public IActionResult GetHomeById([FromRoute] long id)
74
    74
                                                                                                                   +
75
    75
                return Ok(homeResponse);
76
    76
77
    77 +
    78 + [HttpPut]
    79 + [Route("{id}/devices")]
    80 + [RolesWithPermissions(RoleWithPermissions)]
    81 + public IActionResult PutDevicesInHome([FromRoute] long id, [FromBody] HomeDevicesRequest request)
    82 +
    83 +
                _homeService.PutDevicesInHome(id, request.ToEntity());
    84 +
              return Ok();
78 86 }
```