

Universidad ORT Uruguay

# Diseño de Aplicaciones 2

## Obligatorio 1

Link al repositorio:

[https://github.com/IngSoft-DA2/278311\\_227286\\_266889.git](https://github.com/IngSoft-DA2/278311_227286_266889.git)

### **Docentes:**

Nicolás Fierro  
Alexander Wieler  
Máximo Halty

### **Alumnos:**

Nicolás Russo 227286  
Matias Hirschfeld 266889  
Victoria Chappuis 278311

## DOCUMENTO: Aplicación de TDD y Clean Code

<b>Pruebas unitarias.....</b>	<b>3</b>
Proyectos.....	3
Clean Code.....	3
Nombres de variables y funciones claros y descriptivos.....	3
Funciones cortas y con una sola responsabilidad.....	3
Evitar la duplicación de código (DRY - Don't Repeat Yourself).....	3
Manejo de excepciones claro y consistente.....	3
TDD.....	4
Uso de Mocks en los tests.....	4
Descripción de la estrategia de tdd seguida: Inside-out.....	4
TDD Documentation.....	4
Asociar dispositivos al hogar:.....	4
Creación de una empresa:.....	6
Detección de movimiento:.....	9
Mantenimiento de cuentas de administrador.....	11
Cobertura de pruebas unitarias.....	14

# Pruebas unitarias

## Proyectos

Dentro de nuestra solución creamos tres proyectos de pruebas. Estos son las pruebas de Domain, las cuales son nuestras clases de dominio, las pruebas de BusinessLogic, que son las que testean los servicios, y también las pruebas de WebApi, las cuales testean la correcta comunicación de los controladores con los servicios.

## Clean Code

Para la implementación del código de nuestro obligatorio, seguimos las prácticas de clean code de Robert C. Martin. Las mismas son una serie de prácticas que aseguran que el código sea fácil de leer, comprender y mantener por cualquier desarrollador que interactúe con él en el futuro.

### Nombres de variables y funciones claros y descriptivos

Uno de los pilares de clean code es elegir correctamente los nombres de las variables y métodos. De forma que sean claros y descriptivos sobre su funcionalidad. Esto reduce la necesidad de comentarios adicionales y facilita la comprensión del código.

### Funciones cortas y con una sola responsabilidad

Este principio establece que las funciones no deben ser demasiado largas, y que las mismas deben tener una única responsabilidad. Esto facilita el mantenimiento del mismo y permite la reutilización de código.

Por ejemplo, en el método de DeleteUser de UserService se utiliza el método GetById. El mismo busca dentro del repositorio al usuario deseado y si no lo encuentra retorna un error. Esto permite que la función DeleteUser se encargue únicamente de borrar a usuario y que no tenga que validar si el usuario existe.

### Evitar la duplicación de código (DRY - Don't Repeat Yourself)

En varias partes del código se identificaron áreas donde era posible la duplicación de lógica. Por ejemplo, se vio que varios métodos precisaban verificar si un usuario se encontraba dentro de un hogar en HomeService, por esto se creó el método IsMember que toma un user y una home y verifica si el mismo se encuentra dentro de la home.

## Manejo de excepciones claro y consistente

El manejo de excepciones se realizó de manera clara y consistente en todo el código. Se lanzaron excepciones específicas cuando algo salía mal, y los mensajes de error fueron claros y específicos, lo que facilita la depuración y mejora la experiencia del usuario.

Por ejemplo, en el caso de `DeleteUser` si no se encuentra el usuario a eliminar se lanza una excepción que especifica justamente eso: "User not found".

## TDD

Utilizamos TDD porque nos permite garantizar que cada parte del código cumple con los requisitos antes de avanzar al siguiente paso. Primero, escribimos pruebas automatizadas que definen el comportamiento esperado del código. Luego, chequeamos que las pruebas fallen. Finalmente, desarrollamos el código necesario para pasarlas. Este ciclo de escribir pruebas, desarrollar código y refactorizar asegura que el código sea robusto y esté libre de errores desde el principio. Además, TDD facilita el mantenimiento y la evolución del sistema, ya que las pruebas actúan como una red de seguridad que nos alerta de cualquier problema cuando realizamos cambios en el código.

## Uso de Mocks en los tests

Para facilitar la creación de los tests, hicimos uso de mocks. Los mocks son objetos simulados que imitan el comportamiento de dependencias externas o colaboradoras, permitiendo aislar la unidad de código bajo prueba. Su uso facilita que enfoquemos las pruebas en el *system under test* (SUT), es decir, el componente que estamos probando directamente y no sus dependencias externas como bases de datos o servicios externos.

Al utilizar mocks en los tests de servicios, simulamos el comportamiento de los repositorios para asegurar que la lógica del servicio funcione como se espera sin necesidad de acceder a una base de datos real. Sin embargo, esto implica que cuando testeamos los servicios no estamos validando completamente el comportamiento de los repositorios, ya que los mocks solo imitan sus respuestas.

## Descripción de la estrategia de tdd seguida: Inside-out


Para el desarrollo de las siguientes pruebas, utilizamos la estrategia de desarrollo *inside-out*. Es por esto que, desarrollamos el sistema tomando en cuenta en primer lugar, cómo se iba a trasladar a la base de datos. Es decir, creamos las clases principales de

dominio y, antes de implementar cualquier servicio, nos aseguramos de migrar todo a la nueva base de datos.

Este enfoque minimizó el riesgo de problemas relacionados con la base de datos en etapas posteriores del desarrollo, asegurando una transición más suave y menos propensa a errores. También, nos permitió tener una menor cantidad de dependencias externas, como no fue necesario simular o controlar grandes partes del sistema.

## TDD Documentation

Asociar dispositivos al hogar:

feat: [RED] Add device, home not found  matiHirCab committed 8 minutes ago	34efa07  <>
feat: [GREEN] Add device, home not found  matiHirCab committed 8 minutes ago	3cc88e8  <>
feat: [RED] Add device, device not found  matiHirCab committed 5 minutes ago	6bbbd5c  <>
feat: [GREEN] Add device, device not found  matiHirCab committed 5 minutes ago	eb7000e  <>
feat: [RED] add device, successfully added device  matiHirCab committed 2 minutes ago	10dd393  <>
feat: [GREEN] add device, successfully added device  matiHirCab committed 1 minute ago	9e1cfb2  <>

Como muestra la imagen anterior, la funcionalidad de asociar dispositivos al hogar fue implementada siguiendo estrictamente el ciclo de RED, GREEN y REFACTOR de TDD.

Para las pruebas de esta funcionalidad se utilizó tres mocks, uno para el repositorio de devices, otro para el de homes y otro para los objetos homeDevice, los cuales representan qué cierto device se encuentra en cierta home. Estos simulan la interacción que tenga nuestro servicio con la base de datos. Dentro de esta funcionalidad se encontraron dos casos bordes, los cuales fueron qué no se encuentre el hogar al que agregar el device dentro de la base de datos y qué no se encuentre el device a agregar dentro de la base de datos.

En un principio, se realizó una prueba unitaria en la cual se intenta agregar un device a una home no existente, por lo que el resultado esperado es un error con el mensaje home not found. Luego de creada dicha prueba, de la clase HomeService, se agregó la función AddDevice, la cual tomando únicamente un homeld, como de momento es el único dato que

esta función precisa retorna una excepción si no encuentra a dicho hogar dentro del repositorio.

```
[TestMethod]
1 test OK 2 Admin
public void Should_Return_Exception_AddDevice_Home_Not_Found()
{
    var homeId = 1;

    var deviceId = 1;
    var exception = Assert.ThrowsException<Exception>(() => _homeService.AddDevice(homeId, deviceId));
    Assert.AreEqual(expected: exception.Message, actual: "Home not found");
}
```

Luego, se creó una prueba unitaria para el caso en el qué se intenta agregar un device a una home, pero no se encuentra el device qué se está intentando agregar en nuestra base de datos, por lo qué el resultado esperado es un error con el mensaje Device not found. Luego de creada dicha prueba, se modifico la función AddDevice para qué la misma tome también un deviceId y lo busque en un repositorio, retornando un error si el mismo no es capaz de encontrarlo. Para esta prueba se le especificó al mock qué la función get debe devolver una home, esto es para evitar caer en el caso anteriormente testado.

```
[TestMethod]
1 test OK 2 Admin More...
public void Should_Return_Exception_AddDevice_Device_Not_Found()
{
    var homeId = 1;

    var deviceId = 1;
    _homeRepositoryMock.Setup(expression: repo => repo.Get(searchCondition: It.IsAny<Expression<Func<Home, bool>>>()), includes: null)).Returns(new Home());

    var exception = Assert.ThrowsException<Exception>(() => _homeService.AddDevice(homeId, deviceId));
    Assert.AreEqual(expected: exception.Message, actual: "Device not found");
}
```

Por último, se creó la prueba unitaria en la qué se intenta agregar un device a una home en la qué tanto el device como la home se encuentran dentro del repositorio, por lo qué el resultado esperado es qué se inserte un objeto HomeDevice dentro de su respectivo repositorio. Luego de creada dicha prueba, se modifico el método AddDevice para qué luego de los chequeos anteriores cree un objeto HomeDevice y lo inserte dentro de su respectivo repositorio.

```
[TestMethod]
2 Admin
public void Should_Add_Device()
{
    var homeId = 1;

    var deviceId = 1;

    _homeRepositoryMock.Setup(expression: repo => repo.Get(searchCondition: It.IsAny<Expression<Func<Home, bool>>>()), includes: null)).Returns(new Home());
    _deviceRepositoryMock.Setup(expression: repo => repo.Get(searchCondition: It.IsAny<Expression<Func<Device, bool>>>()), includes: null)).Returns(new WindowSensor());

    _homeService.AddDevice(homeId, deviceId);

    _homeDeviceRepositoryMock.Verify(expression: x => x.Insert(entity: It.IsAny<HomeDevice>()), Times.Once);
}
```

## Creación de una empresa:

[RED] AddCompany user not found exception matiHirCab committed 14 minutes ago	79af71a <>
[GREEN] AddCompany user not found exception matiHirCab committed 13 minutes ago	da70796 <>
[RED] AddCompany user not company owner exception matiHirCab committed 9 minutes ago	feaacd4 <>
[GREEN] AddCompany user not company owner exception matiHirCab committed 8 minutes ago	f52a99c <>
[RED] AddCompany should add company matiHirCab committed 6 minutes ago	92116fc <>
[GREEN] AddCompany should add company matiHirCab committed 5 minutes ago	7b43acf <>
[RED] AddCompany user already has a company exception matiHirCab committed 2 minutes ago	59d4d30 <>
[GREEN] AddCompany user already has a company exception matiHirCab committed 2 minutes ago	19ba2ca <>

Como muestra la imagen anterior, la funcionalidad de crear una empresa fue implementada siguiendo estrictamente el ciclo de RED, GREEN y REFACTOR de TDD. Para estas pruebas se utilizaron dos mocks, uno para el repositorio de user y otro para el repositorio de company. Estos simulan la interacción de nuestro servicio CompanyService con la base de datos. Dentro de esta funcionalidad se encontraron tres casos bordes, qué el user que esté intentando crear una company no se encuentre dentro de nuestra base de datos, qué el user que este intentando crear una company no tenga los permisos necesarios para hacerlo y qué el usuario que este intentando crear una company ya tenga una asociada a su cuenta.

En principio, se creó la prueba unitaria para el caso en el que el user que está intentando crear una company no se encuentre dentro del repositorio. Para esto, se configuró el mock del repositorio de usuarios de forma tal que su función get retorna null cuando se la invoca. De esta forma, simulando que no se lo encuentra al usuario dentro de la base de datos. El resultado esperado de esta prueba es que retorne una excepción. Luego de creada esta prueba se creó el método AddCompany el cual en este caso recibe únicamente el id del user que quiere crear la company, como este es el único dato que precisa para pasar la prueba anteriormente creada.

```

[TestMethod]
1 test OK  Nicolás Russo +1
public void AddCompany_UserNotFound_ShouldThrow()
{
    _userRepositoryMock // Mock<IGenericRepository<...>>
        .Setup( expression: repo => repo.Get( searchCondition: It.IsAny<Expression<Func<User, bool>>>(), includes: It.IsAny<List<string>>())
        ) // ISetup<IGenericRepository<...>>
        .Returns((User?)null);

    var companyOwnerId = 1;

    var company = new CompanyDto();

    Assert.ThrowsException<ArgumentException>(() => _companyService.AddCompany(company, companyOwnerId));
}

```

Luego, se creó una prueba unitaria para el caso en el que el user que está intentando crear una company no posee el rol de CompanyOwner. Para esto, se configuró el mock de user de forma tal que su función get devuelve un user con el rol distinto a CompanyOwner. Por este motivo, su resultado esperado es una excepción. Luego de creada esta prueba unitaria, se modificó la función AddCompany para que la misma chequee que el rol del user sea el de CompanyOwner.

```

[DataRow(UserRoles.HomeOwner)]
[DataRow(UserRoles.Administrator)]
[TestMethod]
2 tests OK  Nicolás Russo +1
public void AddCompany_UserNotCompanyOwner_ShouldThrow(UserRoles role)
{
    var user = new User() { UserId = 1, Role = role };

    _userRepositoryMock // Mock<IGenericRepository<...>>
        .Setup( expression: repo => repo.Get( searchCondition: It.IsAny<Expression<Func<User, bool>>>(), includes: It.IsAny<List<string>>())
        ) // ISetup<IGenericRepository<...>>
        .Returns(user);

    var company = new CompanyDto();

    Assert.ThrowsException<Exception>(() => _companyService.AddCompany(company, user.UserId));
}

```

Después del caso anteriormente mencionado, se consideró el de que un user esté intentando crear una company cuando el mismo ya posee una. Es por esto, que se configuró el mock del repositorio de user de forma tal que su función get devuelva un usuario en el cual el campo company no sea nulo. El resultado esperado de dicha prueba es que retorne una excepción. Luego de creada la misma, se modificó el método AddCompany para que el mismo, luego de buscar en el repositorio el usuario, chequee si el mismo ya tiene una Company y, en caso que la tenga, retorne una excepción.



```

[TestMethod]
1 test OK  Nicolás Russo +1
public void AddCompany_UserAlreadyHasCompany_ShouldThrow()
{
    var user = new User()
    {
        UserId = 1,
        Role = UserRoles.CompanyOwner,
        Company = new Company(),
    };

    _userRepositoryMock // Mock<IGenericRepository<...>>
        .Setup( expression: repo => repo.Get( searchCondition: It.IsAny<Expression<Func<User, bool>>>(), includes: It.IsAny<List<string>>())
        ) // ISetup<IGenericRepository<...>>
        .Returns(user);

    var company = new CompanyDto();

    Assert.Throws<Exception>(() => _companyService.AddCompany(company, user.UserId));
}

```

Por último, se consideró el caso en el que un user podría agregar una company. Para esto se configuró el mock de user de forma tal que su función get devuelve un user con el rol adecuado, CompanyOwner y sin una company. De esta forma, el resultado esperado es que se le asigne dicha company al user y se inserte dentro del repositorio de company. Luego de creada la misma, se modificó el método AddCompany para que él mismo le asigne al user

```

[TestMethod]
1 test OK  Nicolás Russo +1
public void AddCompany_ShouldAddCompany()
{
    var user = new User() { UserId = 1, Role = UserRoles.CompanyOwner };

    _userRepositoryMock // Mock<IGenericRepository<...>>
        .Setup( expression: repo => repo.Get( searchCondition: It.IsAny<Expression<Func<User, bool>>>(), includes: It.IsAny<List<string>>())
        ) // ISetup<IGenericRepository<...>>
        .Returns(user);

    var company = new CompanyDto();

    _companyService.AddCompany(company, user.UserId);

    _companyRepositoryMock.Verify( expression: repo => repo.Insert( entity: It.IsAny<Company>(), Times.Once);
}

```

Detección

de

movimiento:

<b>[RED] detect motion should return error when security camera not found</b> 👤 vickychappuis committed 5 hours ago	b7d47a1
<b>[GREEN] detect motion should return error when security camera not found</b> 👤 vickychappuis committed 5 hours ago	a7b5662
<b>[RED] detect motion should return error when security camera can't detect motion</b> 👤 vickychappuis committed 5 hours ago	e34c3db
<b>[GREEN] detect motion should return error when security camera can't detect motion</b> 👤 vickychappuis committed 5 hours ago	26035cd
<b>[RED] detect motion should return error when device not found in home</b> 👤 vickychappuis committed 4 hours ago	9da489b
<b>[GREEN] detect motion should return error when device not found in home</b> 👤 vickychappuis committed 4 hours ago	885bacb
<b>[RED] detect motion should return error when device not connected</b> 👤 vickychappuis committed 4 hours ago	6073e8d
<b>[GREEN] detect motion should return error when device not connected</b> 👤 vickychappuis committed 4 hours ago	87c4cbd
<b>[RED] detect motion should add notification</b> 👤 vickychappuis committed 4 hours ago	ecf2dab
<b>[GREEN] detect motion should add notification</b> 👤 vickychappuis committed 4 hours ago	b9e83fa

Como muestra la imagen anterior la funcionalidad de detección de movimiento fue implementada siguiendo estrictamente el ciclo de RED, GREEN y REFACTOR de TDD. Para estas pruebas se utilizaron tres mocks, uno para el repositorio de device, otro para el de home y otro para el de notification. Estos simulan la interacción de nuestro NotificationService con la base de datos. Dentro de esta funcionalidad se encontraron cuatro casos borde, el caso qué no encuentra la security camera deseada en el repositorio, el caso en el qué la security camera deseada no puede detectar movimiento, el caso en el qué la security camera deseada no se encuentra en la home y el caso en el qué la security camera no esta conectada.

En principio, se creó una prueba unitaria para el caso en el qué no se encuentra la security camera en el sistema. Para esto, se configuró el mock del repositorio de device para qué cuando se le realice la función get devuelva null. De esta forma, el resultado esperado de CreateNotification es una excepción indicando qué no se encontró el dispositivo buscado. Luego, se creó la función CreateNotification la cual recibe únicamente el deviceId y chequea si se encuentra dicho device en el repositorio de device y, si no lo encuentra, retorna una excepción.

```

[TestMethod]
1 test OK * Nicolás Russo +1
public void DetectMotion_WhenSecurityCameraNotFound_ShouldReturnError()
{
    var deviceId = 1;
    var homeId = 2;

    _deviceRepositoryMock // Mock<IGenericRepository<...>>
        .Setup(expression: repo => repo.Get(searchCondition: It.IsAny<Expression<Func<Device, bool>>>()), includes: null)) // ISetup<IGenericRepository<...>>
        .Returns((Device)null); // Converting null literal or possible null value into non-nullable type.

    var ex = Assert.ThrowsException<Exception>{
        () => _notificationService.CreateNotification(deviceId, homeId, notificationType: "Motion")
    };
    Assert.AreEqual(expected: "Security Camera not found", actual: ex.Message);
}

```

Luego, se creó la prueba unitaria para el caso en el se encuentra la security camera, pero la misma no es capaz de detectar movimiento. Para esto se configuró el mock para qué el método get retorne una security camera la cual no pueda detectar movimiento. Debido a qué no puede detectar movimiento, el resultado esperado de la función es qué retorne una excepción. Luego de realizada la prueba unitaria se modificó el método CreateNotification para qué el mismo busque en el repositorio la security camera y verifique si puede detectar movimiento o no, en el caso qué no pueda, deberá retornar una excepción.

```

[TestMethod]
1 test OK * Nicolás Russo +1
public void DetectMotion_WhenSecurityCameraCannotDetectMotion_ShouldReturnError()
{
    var deviceId = 1;
    var homeId = 2;
    var camera = new SecurityCamera { CanDetectMotion = false };

    _deviceRepositoryMock // Mock<IGenericRepository<...>>
        .Setup(expression: repo => repo.Get(searchCondition: It.IsAny<Expression<Func<Device, bool>>>()), includes: null)) // ISetup<IGenericRepository<...>>
        .Returns(camera);

    var ex = Assert.ThrowsException<Exception>{
        () => _notificationService.CreateNotification(deviceId, homeId, notificationType: "Motion")
    };
    Assert.AreEqual(expected: "Security Camera cannot detect motion", actual: ex.Message);
}

```

Luego, se creó la prueba unitaria para el caso en el qué se encuentra la security camera dentro de los devices, la misma puede detectar movimiento, pero no está dentro de la home. Es por esto qué se configuró el mock del repositorio de device para qué su método get retorne una security camera con detección de movimiento y el mock del repositorio de HomeDevice para qué su método get retorne null, simbolizando qué no se encontró la security camera dentro de la home. Es por esto qué el retorno esperado es una excepción indicando qué no se encontró el device en la home. Luego de realizada la prueba unitaria, se modificó el método CreateNotification para qué este busque la security camera dentro de la casa seleccionada y, en caso de qué no la encuentre, retorne una excepción.

```

[TestMethod]
1 test OK * Nicolás Russo +1
public void DetectMotion_WhenDeviceNotFoundInHome_ShouldReturnError()
{
    var deviceId = 1;
    var homeId = 2;
    var camera = new SecurityCamera { CanDetectMotion = true };

    _deviceRepositoryMock // Mock<IGenericRepository<...>>
        .Setup( (expression: repo:IGenericRepository<Device> => repo.Get( searchCondition: It.IsAny<Expression<Func<Device, bool>>>(), includes: null)) // ISetup<IGenericRepository<...>>
        .Returns(camera);
    _homeDeviceRepositoryMock // Mock<IGenericRepository<...>>
        .Setup( (expression: repo:IGenericRepository<HomeDevice> => repo.Get( searchCondition: It.IsAny<Expression<Func<HomeDevice, bool>>>(), includes: null)) // ISetup<IGenericRepository<...>>
        .Returns((HomeDevice)null); // Converting null literal or possible null value into non-nullable type.

    var ex = Assert.ThrowsException<Exception>(
        () => _notificationService.CreateNotification(deviceId, homeId, notificationType: "Motion")
    );
    Assert.AreEqual( expected: "Device not found in home", actual: ex.Message);
}

```

Posteriormente se creó la prueba unitaria para el caso en el qué se encuentra la security camera, la misma puede detectar movimiento y se encuentra en la home seleccionada, sin embargo la misma no se encuentra conectada. Es por esto qué se configuró el mock del repositorio de device para qué la función get retorne una security camera con detección de movimiento y el mock del repositorio del HomeDevice retorne un objeto HomeDevice el cual no se encuentre conectado. Es por esto qué el retorno esperado debe ser una excepción indicando qué el HomeDevice no se encuentra conectado. Luego de creada la prueba unitaria se modificó el método CreateNotifications para qué el mismo chequee si el HomeDevice buscado se encuentra conectado o no y, en caso de qué no se encuentre conectado, retorne una excepción.

```

[TestMethod]
1 test OK * Nicolás Russo +1
public void DetectMotion_WhenDeviceNotConnected_ShouldReturnError()
{
    var deviceId = 1;
    var homeId = 2;
    var camera = new SecurityCamera { CanDetectMotion = true };
    var homeDevice = new HomeDevice { IsConnected = false };

    _deviceRepositoryMock // Mock<IGenericRepository<...>>
        .Setup( (expression: repo:IGenericRepository<Device> => repo.Get( searchCondition: It.IsAny<Expression<Func<Device, bool>>>(), includes: null)) // ISetup<IGenericRepository<...>>
        .Returns(camera);
    _homeDeviceRepositoryMock // Mock<IGenericRepository<...>>
        .Setup( (expression: repo:IGenericRepository<HomeDevice> => repo.Get( searchCondition: It.IsAny<Expression<Func<HomeDevice, bool>>>(), includes: null)) // ISetup<IGenericRepository<...>>
        .Returns(homeDevice);

    var ex = Assert.ThrowsException<Exception>(
        () => _notificationService.CreateNotification(deviceId, homeId, notificationType: "Motion")
    );
    Assert.AreEqual( expected: "Device is not connected", actual: ex.Message);
}

```

Por último, se creó una prueba unitaria para el caso en el qué se encuentra la security camera, la misma puede detectar movimiento, se encuentra en la home seleccionada y está conectada. Para esto se configuró el mock de repositorio de device para qué retorne una security camera con detección de movimiento y el mock de repositorio de HomeDevice para qué el mismo retorne un HomeDevice qué esté conectado. El resultado esperado para esta prueba es qué se inserte una notificación dentro del repositorio de notificaciones, debido a qué se cumplen todas las condiciones necesarias para hacerlo. Luego, se modificó el método CreateNotification para qué con los datos provistos anteriormente cree la notificación y la inserte dentro del repositorio de notificaciones.

```

[TestMethod]
1 test OK * Nicolás Russo +1
public void DetectMotion_ShouldAddNotification()
{
    var deviceId = 1;
    var homeId = 2;
    var camera = new SecurityCamera { CanDetectMotion = true };
    var homeDevice = new HomeDevice { IsConnected = true, HardwareId = 5 };

    _deviceRepositoryMock // Mock<IGenericRepository<Device>>
        .Setup(x => x.Get(searchCondition: It.IsAny<Expression<Func<Device, bool>>>(), includes: null))) // ISetup<IGenericRepository<Device>>
        .Returns(camera);
    _homeDeviceRepositoryMock // Mock<IGenericRepository<HomeDevice>>
        .Setup(x => x.Get(searchCondition: It.IsAny<Expression<Func<HomeDevice, bool>>>(), includes: null))) // ISetup<IGenericRepository<HomeDevice>>
        .Returns(homeDevice);

    _notificationService.CreateNotification(deviceId, homeId, notificationType: "Motion");

    _notificationRepositoryMock.Verify(
        x => x.Insert(entity: It.IsAny<Notification>()),
        Times.Once
    );
}

```

## Mantenimiento de cuentas de administrador

<b>[REFACTOR] delete admin</b> vickyhappuis committed 2 days ago	f3eff1f	<>
<b>[GREEN] delete admin</b> vickyhappuis committed 2 days ago	574a9e8	<>
<b>[RED] delete admin</b> vickyhappuis committed 2 days ago	e5fcb3	<>
<b>[REFACTOR] add admin</b> vickyhappuis committed 2 days ago	985332d	<>
<b>[GREEN] add admin</b> vickyhappuis committed 2 days ago	fac20d7	<>
<b>[RED] add admin</b> vickyhappuis committed 2 days ago	08994c6	<>

Como muestra la imagen anterior la funcionalidad de detección de movimiento fue implementada siguiendo estrictamente el ciclo de RED, GREEN y REFACTOR de TDD. Para estas pruebas se utilizó un mock para el repositorio de user. Se implementaron dos funciones, una que permite a un administrador crear un user y otra que permite hacer delete de un user. Se identificaron varios casos bordes, entre ellos el caso en el que el usuario a borrar no se encuentre en la base de datos, que la cuenta a la que se está intentando borrar no sea un administrador y, por ende, no se pueda borrar y que el usuario que se está intentando crear no sea un administrador.

En principio, se creó la prueba unitaria para el caso en el que no se encontró al user que se está intentando eliminar de la base de datos. Para esto, se configuró el mock del repositorio de user de forma tal que su función get retorne null, es decir que no lo encuentre dentro del repositorio. Es por esto, que el resultado esperado de dicha prueba es una excepción mostrando que el user buscado no se encontró. Luego de creada dicha prueba se creó la función DeleteUser que toma únicamente un userId y busca dicho user id dentro del

repositorio. En caso de que no lo encuentre retorna una excepción.

```
[TestMethod]
1 test OK  Victoria Chappuis +1
public void DeleteUser_UserNotFound_ShouldThrow()
{
    var userId = 1;

    _userRepositoryMock // Mock<IGenericRepository<...>>
        .Setup( expression: repo => repo.Get<User>() =>
            repo.Get( searchCondition: It.IsAny<Expression<Func<User, bool>>>(), includes: It.IsAny<List<string>>())
        ) // ISetup<IGenericRepository<...>,...>
        .Returns((User?)null);

    var exception = Assert.ThrowsException<Exception>(() => _userService.DeleteUser(userId));

    Assert.AreEqual( expected: "User not found", actual: exception.Message);
}
```

A continuación, se consideró el caso en el que si se encuentra al user en el repositorio, pero el mismo no es administrador, por lo que no se puede eliminar. Para esto se configuró el mock del repositorio del user de forma tal que el get retorne un user con un rol distinto a administrador. Por lo que, el resultado esperado es una excepción diciendo que no es posible eliminar el usuario. Luego de creada la prueba se modificó el método DeleteUser para que luego de encontrar el user chequee su rol y si no es administrador retorne una excepción.

```
[DataRow(UserRoles.HomeOwner)]
[DataRow(UserRoles.CompanyOwner)]
[TestMethod]
2 tests OK  Victoria Chappuis +1
public void DeleteUser_UserNotAdmin_ShouldThrow(UserRoles role)
{
    var userId = 1;
    var user = new User { UserId = userId, Role = role };

    _userRepositoryMock // Mock<IGenericRepository<...>>
        .Setup( expression: repo => repo.Get<User>() =>
            repo.Get( searchCondition: It.IsAny<Expression<Func<User, bool>>>(), includes: It.IsAny<List<string>>())
        ) // ISetup<IGenericRepository<...>,...>
        .Returns(user);

    var exception = Assert.ThrowsException<Exception>(() => _userService.DeleteUser(userId));

    Assert.AreEqual( expected: "Cannot delete. User is not an administrator", actual: exception.Message);
}
```

Después, se consideró el caso en el que se puede borrar a un usuario administrador, es decir que se encuentra en el repositorio y el mismo es administrador. Para esto se configuró el mock del repositorio de user para que el get del mismo retorne un user con rol administrador. El resultado esperado de esta prueba es entonces que se elimine al user del

repositorio. Luego de creada esta prueba, se modificó el método DeleteUser para que después de hacer todos los chequeos anteriores elimine al user del repositorio.

```
[TestMethod]
1 test OK  Victoria Chappuis +2
public void DeleteUser_IsAdmin_ShouldDeleteUser()
{
    var userId = 1;
    var expectedUser = new User { UserId = userId, Role = UserRoles.Administrator };

    _userRepositoryMock // Mock<IGenericRepository<...>>
        .Setup( expression: repo => repo.Get( searchCondition: It.IsAny<Expression<Func<User, bool>>>(), includes: It.IsAny<List<string>>())
        ) // ISetup<IGenericRepository<...>,...>
        .Returns(expectedUser);

    _userService.DeleteUser(userId);

    _userRepositoryMock.Verify( expression: repo => repo.Delete( entity: It.IsAny<User>()), Times.Once);
}
```

Luego, consideramos el caso en el que se está intentando crear un usuario que no sea administrador. Como un administrador puede crear únicamente cuentas de CompanyOwner o otras cuentas de Administrador, el resultado esperado es una excepción, indicando lo anteriormente mencionado. Luego de creada la prueba unitaria se creó el método CreateUser, el mismo recibe un dto con los datos del usuario a crear y verifica su rol, en caso de que no sea uno de los roles que el administrador puede crear se retorna una excepción.

```
[TestMethod]
1 test OK  Victoria Chappuis +2
public void CreateUser_NotAdminOrCompanyOwner_ShouldThrow()
{
    var user = new UserDto { Role = UserRoles.HomeOwner.ToString() };

    var exception = Assert.ThrowsException<Exception>(() => _userService.CreateUser(user));

    Assert.AreEqual(
        expected: "Cannot create a user that's not a company owner or administrator",
        actual: exception.Message
    );
}
```

Por último, se consideró el caso en el que el administrador puede crear el usuario, es decir que su rol es CompanyOwner o Administrador. Por esto, el resultado esperado de dicha prueba es que se inserte un usuario dentro del repositorio de usuarios. Luego de creada la prueba unitaria se modificó el método CreateUser para que luego de realizar las validaciones anteriores el mismo cree el usuario y lo inserte en el repositorio.

```

[DataRow(UserRoles.Administrator)]
[DataRow(UserRoles.CompanyOwner)]
[TestMethod]
2 tests OK 2 Victoria Chappuis +2
public void CreateUser_CompanyOwnerOrAdministrator_ShouldCreateUser(UserRoles role)
{
    var user = new UserDto { Role = role.ToString() };

    _userService.CreateUser(user);

    _userRepositoryMock.Verify(expression: repo => repo.Insert(entity: It.IsAny<User>()), Times.Once);
}

```

## Cobertura de pruebas unitarias

La cobertura de todas las pruebas es del 100% del código como se ve en la siguiente imagen.

✓ Total	100%	0/1714
> smarthome.WebApi	100%	0/195
> smarthome.BusinessLogic	100%	0/263
> smarthome.Domain	100%	0/173
> smarthome.Domain.tests	100%	0/107
> smarthome.BusinessLogic.tests	100%	0/656
> smarthome.WebApi.tests	100%	0/320

> ✓ [C#] smarthome.BusinessLogic.tests (125 tests) Success
> ✓ [C#] smarthome.Domain.tests (13 tests) Success
> ✓ [C#] smarthome.WebApi.tests (44 tests) Success