Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 2 - Obligatorio 1

Evidencia de Clean Code y TDD

https://github.com/ORT-DA2/OBL-Asadurian-Reyes.git

Diego Asadurian 198874 - Hernán Reyes 235861

Tutor: Gabriel Piffaretti, Nicolas Blanco, Daniel Acevedo

07 de Octubre de 2021

Evidencia de Clean Code y de la aplicación de TDD

El fin de este documento es mostrar y dejar en evidencia la metodología utilizada para el desarrollo de dicha aplicación y mostrar que el código cumple con los estándares de Clean Code.

Para desarrollar este sistema se implementó la metodología TDD, la cual implica llevar a cabo 3 grandes etapas, primero se escribió inicialmente las pruebas unitarias, luego se escribio el codigo necesario para que dicha prueba pasarán de forma correcta y por último en caso de necesitarlo se realizó un refactor en el código con el fin de mejorarlo y seguir un estándar determinado por el equipo.

Pruebas unitarias

Para llevar a cabo los proyectos Test, se utilizo el Framework MSTest y para implementar correctamente las pruebas utilizamos la estructura Arrange, Act, Assert.

Las pruebas unitarias, tal como indica su nombre, realizan pruebas de funcionalidades de manera unitaria.

Nuestras clases del sistema se relacionan entre sí, incluso con clases de otros paquetes. Para poder cumplir con las pruebas unitarias correctamente y no terminar realizando pruebas de integración se utilizaron mocks.

Nuestros mocks nos permiten simular distintos comportamientos a funcionalidades que no corresponden probar.

Para esto realizamos el setup de nuestro mock, mockeando la correspondiente funcionalidad que nuestro método a probar utiliza.
Donde se debe simular el comportamiento de dichos métodos indicando que métodos esperamos que se llamen, que parámetros recibirán y que deben devolver.

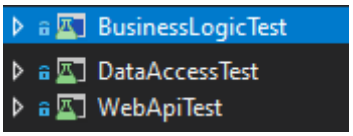
En la siguiente imagen podemos visualizar lo explicado previamente, en dicho ejemplo estamos mockeando a nuestra Interface ProjectRepository (la encargada de realizar las operaciones de Project en nuestra base de datos). En este caso queremos que el metodo GetById, que recibe un id de proyecto por parametro devuelva un proyecto en particular.

```
[TestMethod]
0 references
public void GetAllTesterInProject()
{
    project.Users.Add(tester);

    List<User> users = new List<User>();
    users.Add(tester);

    mockProjectRepository.Setup(x => x.GetById(project.Id)).Returns(project);

    List<User> allUsers = projectLogic.GetAllTesters(project);

    mockProjectRepository.VerifyAll();

    Assert.IsTrue(allUsers.SequenceEqual(users));
}
```

De esta manera podemos garantizar que dichos métodos no sean un problema en nuestra prueba y en caso de que la prueba falle tener la certeza de que fue nuestro código el que falló y no el de un 3ero.

Estructura

Para la realización de nuestras pruebas unitarias se crearon 3 proyectos de prueba, BussinessLogicTest, DataAccessTest y WebApiTest.



Cada proyecto/clases de prueba de nuestro sistema cumplen con el principio FIRST, lo cual quiere decir a grandes rasgos que las pruebas deben ser:

- **Rápidas** (poder ejecutar un gran número de pruebas en un lapso corto de tiempo)
- **Independientes** (que no haya dependencia entre una prueba y otra)
- **Repetible** (que el resultado de las pruebas sea el mismo independientemente del servidor o lugar donde se ejecuten, no deben depender de configuraciones particulares)
- **Auto Evaluable** (se ejecutan de forma automática, sin necesidad de nuestra interacción)
- **Oportunas** (quiere decir que las pruebas deben ser oportunas, y se deben comenzar a desarrollar antes de comenzar a desarrollar el código de producción)

BussinessLogicTest

En este paquete se implementaron todas las clases de prueba necesarias de nuestro negocio.

Se implementó una clase de test por clase de nuestra lógica. Las mismas son encargadas de ejecutar las distintas funcionalidades de nuestro negocio como también la correcta creación de diferentes entidades cumpliendo con las validaciones requeridas por el cliente.

WebApiTest

Dicho paquete es el encargado de realizar distintas pruebas de los controllers de la WebApi. Para la implementación de dichas pruebas se llevaron a cabo una clase de test por controlador de WebApi.

El principal objetivo de nuestras pruebas es validar el correcto envío de mensajes al cliente como también los correctos datos que deben recibir.

DataAccessTest

En este paquete se encuentran las pruebas a realizar para distintas operaciones sobre nuestra base de datos.

En este caso, al contrario de la WebApiTest y BussinessLogicTest, no fue necesaria la utilización de mocks para nuestras pruebas unitarias, dado que nuestros métodos son 100% independientes (no utilizaban llamados a otros metodos).

El principal objetivo es probar el correcto funcionamiento de distintas operaciones sobre nuestra base de datos.

Para llevar a cabo dichas pruebas se utilizó SqliteConnection, de esta manera estamos creando nuestro DBContext por una base SqlLite en memoria.

Evidencia de Clean Code

En nuestra codificación del sistema utilizamos los estándares oficiales de Microsoft c# . NET así como también buenas prácticas de Clean Code.

Se trató de mantener un estándar en la codificación entre el equipo, de esta manera nos fue útil poder entender código realizado por otro compañero de una manera más fácil y rápida. Por otro lado garantizamos que nuestro código puede ser entendido por un tercero sin mucha complejidad.

Para esto hizo bastante hincapié en la utilización de variables nemotécnicas, métodos cortos, codificación en inglés, diferenciación en variables plurales y singulares, utilización de constantes para evitar Magic Constan, entre otras.

```
public class UserLogic : IUserLogic
{
    private const string notExistUser = "User not exist";
    private const string existingUser = "The user already exists";
    private const string invalidRol = "You must entry a valid rol";

    private IUserRepository userRepository;
    private IRepository<Rol, Guid> rolRepository;

    0 references
    public UserLogic() { }

    1 reference
    public UserLogic(IUserRepository userRepository, IRepository<Rol, Guid> rolRepository)
    {
        this.userRepository = userRepository;
        this.rolRepository = rolRepository;
    }

    15 references | 5/5 passing
    public User Get(Guid id)
    {
        User user = userRepository.GetById(id);

        if (user == null)
        {
            throw new NoObjectException(notExistUser);
        }

        return user;
    }

    8 references | 2/2 passing
    public List<User> GetAll()
    {
        return userRepository.GetAll();
    }

    11 references | 9/9 passing
    public User Create(User userToCreate)
    {
        IsValidUser(ref userToCreate);
        NotExistUser(userToCreate);

        userToCreate.Projects = new List<Project>();

        User userCreate = userRepository.Create(userToCreate);

        return userCreate;
    }

    2 references
    public void Update(User user)
    {
        IsValidUser(ref user);
        userRepository.UpdateUser(user);
    }
```

En la imagen anterior se puede apreciar fragmento de nuestro código que evidencia la aplicación de clean code, estándares que fueron adoptados por el equipo, algunos ejemplos son:

- Variables con “camelCase”
- Métodos con “PascalCase”
- Nombre de clases con “PascalCase”
- Manejo de constantes evitando Magic Constant
- Nombre de interfaces comenzando con “I” para su identificación.
- Manejo de abstracciones (IUserRepository, IRepository) para lograr bajo acoplamiento.
- Métodos con una única responsabilidad.
- Métodos que no superan las 20 líneas de código.
- Nombres de los métodos que revelen intención.
- Nombres nemotécnicos.
- Código en inglés.
- No se utilizan comentarios. Código claro y limpio.
- Separación con líneas en blanco entre métodos.
- Se evitaron prefijos.

Justificación y evidencia de TDD

La metodología TDD se basa en crear nuestras pruebas y luego escribir el código de la funcionalidad para que las pruebas pasen. Luego de esto se realiza una etapa de redactor del código implementado.

Comenzamos nuestro TDD desde la lógica de negocios, luego pasando por el acceso a datos y por último por nuestra web api.

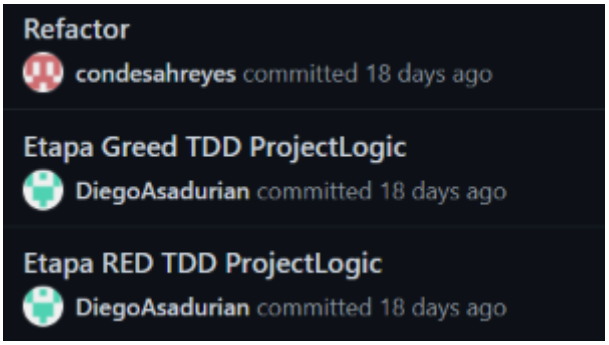
Lo decidimos realizar de esta manera dado que debimos primero definir una estructura de negocio para cumplir con las distintas funcionalidades del negocio definiendo así una estructura de nuestro dominio y entidades.

Terminada la etapa de TDD de nuestra lógica continuamos con el TDD del acceso a datos ya que contábamos con un dominio y lógica estable.

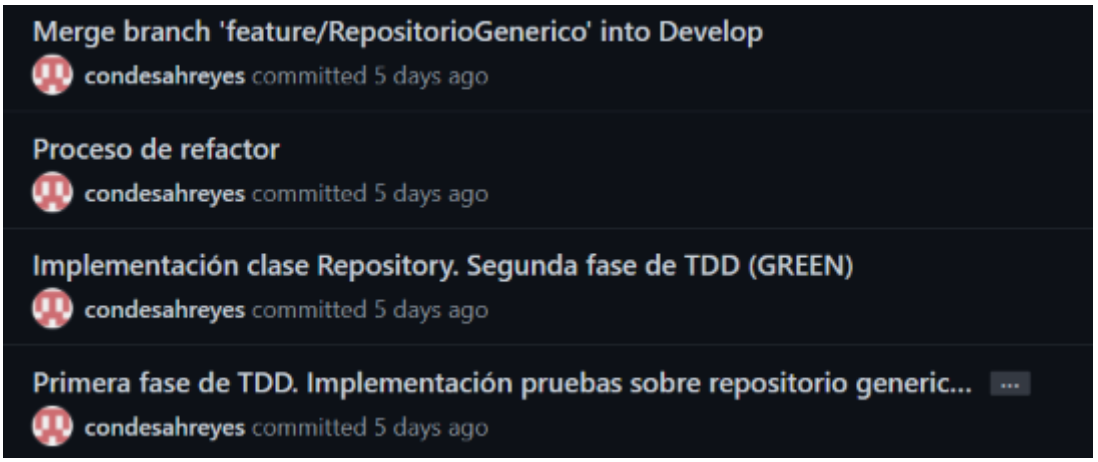
Por último realizamos dicho proceso en nuestra WebApi ya que el mismo podía tener distintas flexibilidades a la hora de ser implementado debido a la escasez de lógica y estructura que la conforma comparado a nuestra lógica de negocio y acceso a datos.

Evidencia de commits

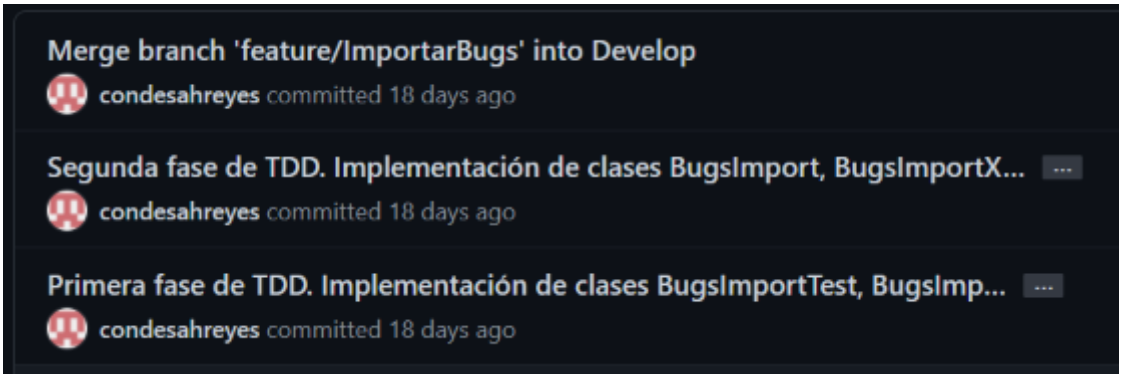
Implementación de clase ProjectLogic, TDD



Implementación del repositorio genérico, TDD



Implementación funcionalidad importación de bugs, TDD



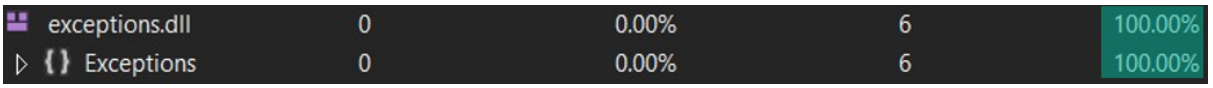
Implementación de BugController, TDD

Análisis de cobertura de pruebas

WebAPI

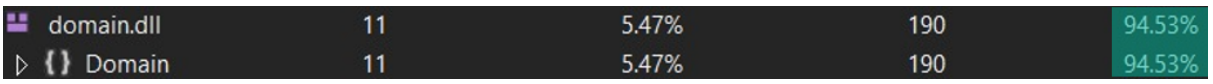
OBLDA2.Models	78	23.93%	248	76.07%
WebApi.Controllers	20	15.50%	109	84.50%

Exceptions



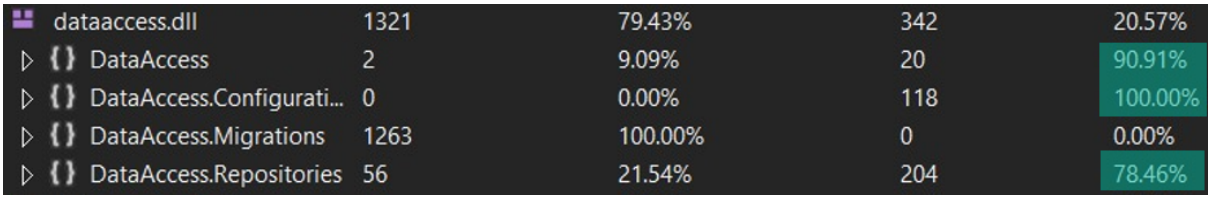
exceptions.dll	0	0.00%	6	100.00%
Exceptions	0	0.00%	6	100.00%

Dominio



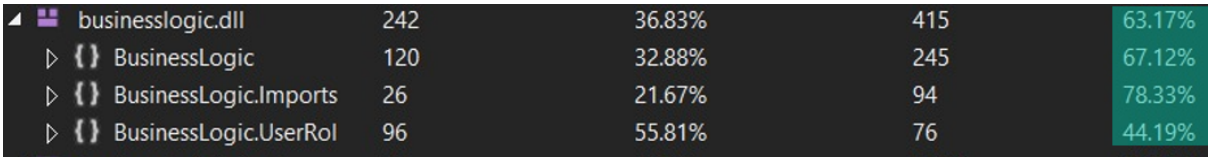
domain.dll	11	5.47%	190	94.53%
Domain	11	5.47%	190	94.53%

DataAcces



dataaccess.dll	1321	79.43%	342	20.57%
DataAccess	2	9.09%	20	90.91%
DataAccess.Configurati...	0	0.00%	118	100.00%
DataAccess.Migrations	1263	100.00%	0	0.00%
DataAccess.Repositories	56	21.54%	204	78.46%

BussinessLogic



businesslogic.dll	242	36.83%	415	63.17%
BusinessLogic	120	32.88%	245	67.12%
BusinessLogic.Imports	26	21.67%	94	78.33%
BusinessLogic.UserRol	96	55.81%	76	44.19%

Se deberá mejorar para una segunda instancia las pruebas de la lógica de negoció.