

Universidad ORT
Facultad de Ingeniería

Evidencia de la aplicación de TDD y Clean Code

Diseño de Aplicaciones 2 - 6/10/2022

[Repositorio Git](https://github.com/ORT-DA2/DA2-163471-223139-201250.git)

<https://github.com/ORT-DA2/DA2-163471-223139-201250.git>

Federico Czarniewicz - 201250 - M6C-1

Ignacio Olivera - 223139 - M6C

Cristhian Maciel - 163471 - N6A

Índice:

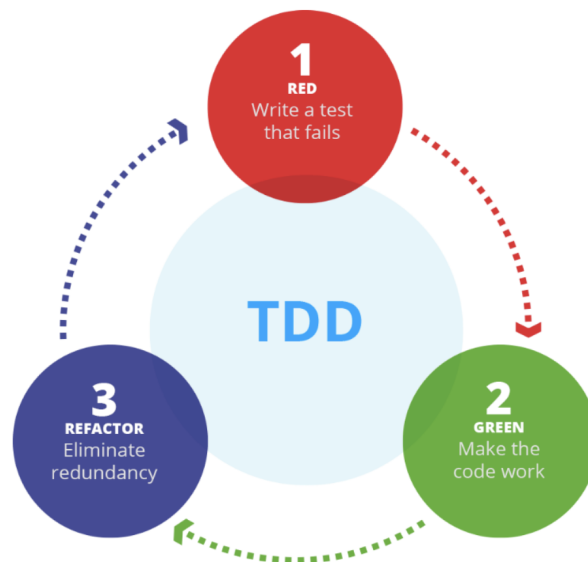
1. Descripción de la estrategia de TDD	2
2. Justificación de clean code	4
2.1 Nombres nemotécnicos	4
2.2 Funciones	5
2.3 Comentarios	6
2.4 Pruebas unitarias	6
2.5 Otros aspectos Clean Code de consistencia	7
3. Ejecución de los test	9
3.1 Informe de cobertura para todas las pruebas desarrolladas	9

1. Descripción de la estrategia de TDD

El sistema fue desarrollado siguiendo TDD (desarrollo guiado por pruebas) para poder lograr un diseño emergente partiendo de los requerimientos.

Tuvimos en cuenta las siguientes reglas:













- No hay que crear código hasta que haya fallado una prueba
- No hay que crear más de una prueba que falle
- El código escrito debe ser el mínimo para que la prueba se exitosa



Aplicamos estas reglas partiendo desde los controladores hasta la capa de persistencia (desde afuera hacia adentro) teniendo en cuenta solamente los requerimientos solicitados. Siguiendo los pasos red, green, refactor hicimos especial atención a los requerimientos marcados como importantes para seguir esta práctica. Sin embargo partimos de una discusión inicial del Dominio del problema y un diagrama abstracto del tipo de arquitectura que queríamos implementar.

Gracias a la aplicación de TDD nos vimos beneficiados en tener un sistema testeable y contar con devolución temprana del impacto de cambios sobre otros módulos. También evitamos agregar código que no necesitamos, generando un código más limpio y al final asegurarnos que el sistema cumple exactamente lo solicitado.

En la siguiente ilustración se evidencia alguno de los commits que se realizaron utilizando la técnica de TDD.

[GREEN] - GetSetSurname method is OK Martinrezy committed 23 days ago		 14e0855	
[RED] - Surname property is not implemented Martinrezy committed 23 days ago		 cd49e0d	
[REFACTOR] - GetSetName without non empty string Martinrezy committed 23 days ago		 c276f51	
[GREEN] - GetSetEmptyName is OK Martinrezy committed 23 days ago		 f373ecf	
Commits on Apr 4, 2019			
[GREEN] - GetSetEmptyName method is OK Martinrezy committed 25 days ago		 97266f6	
[RED] - Manager class and Name property not implemented Martinrezy committed 25 days ago		 b67d6b7	

2. Justificación de clean code

Para cumplir con los lineamientos de Clean Code, se decidió tener en consideración varios de los objetivos que nos plantea Robert C. Martins para obtener un código limpio. A continuación evidenciamos su cumplimiento con algunos ejemplos:

2.1 Nombres nemotécnicos

Se busca que los nombres sean intencionados y descriptivos, evitando abreviaciones, prefijos, secuencia de números en variables y palabras redundantes. Los nombres de las clases no deben ser verbos y se deben evitar sufijos. A su vez, para los nombres de los métodos se deben usar verbos. Cada nombre debe ser corto, explícito y claro. Si la intención es técnica se deben usar nombres técnicos, por ejemplo cuando estamos aplicando algún patrón.

Evidencias:

```
namespace Domain
{
    public class Purchase
    {
        public int Id { get; set; }
        public double TotalPrice { get; set; }
        public DateTime Date { get; set; }
        public string UserEmail { get; set; }
        public List<PurchaseItem> Items { get; set; }
    }
}
```

▼ Domain

> Dtos

Domain.csproj

Drug.cs

Invitation.cs

Pharmacy.cs

Purchase.cs

PurchaseItem.cs

Role.cs

Session.cs

User.cs

```
private Role getExistantRole(string roleName)
{
    Role role;
    try
    {
        role = _roleLogic.GetRoleByName(roleName);
    }
    catch (ResourceNotFoundException e)
    {
        throw new ValidationException("role doesn't exist");
    }
    return role;
}
```

```
public virtual User GetUserByUserName(string userName)
{
    return _userRepository.GetFirst(u => u.UserName == userName);
}
```

2.2 Funciones

Se busca que las mismas sean reducidas (20 líneas) y con nombres descriptivos. Los parámetros que se le pasan a una función no deberían ser más de 3 como máximo. Cada función debe hacer lo que dice su nombre y nada más que eso. Se recomienda extraer funciones en los try y catch.

Evidencias:

```
public virtual Invitation GetInvitationByCode(string invitationCode)
{
    return _invitationRepository.GetFirst(i => i.Code == invitationCode);
}
```

```
private void checkIfUserNameIsRepeated(string userName)
{
    bool userExist = true;
    try
    {
        User user = _userLogic.GetUserByUserName(userName);
    }
    catch (ResourceNotFoundException e)
    {
        userExist = false;
    }

    if (userExist)
    {
        throw new ValidationException("username already exists");
    }
}
```

2.3 Comentarios

No debe haber comentarios salvo que sean aclaratorios y que sumen a la solución como una advertencia.

2.4 Pruebas unitarias
















Se debe cumplir en todo el desarrollo con las 3 leyes de TDD:

- No hay que crear código hasta que haya fallado un unit test
- No hay que crear nunca más de una prueba que falle
- El código creado debe ser el mínimo para que la prueba pase

Si bien este punto no fue respetado durante todo el transcurso del proyecto, si se pueden mostrar evidencias de haberlo realizado en algunos momentos.

Evidencias:

```
[TestMethod]
[ExpectedException(typeof(ValidationException))]
public void CheckEmptyUserNameThrowsException()
{
    Invitation invitation = new Invitation()
    {
        UserName = "",
        Role = new Role()
        {
            Name = "Employee"
        }
    };
}
```

RED: create purchase test controller  crishianfms committed 4 days ago	 dbddd71 
RED: purchase controller created and its models  crishianfms committed 4 days ago	 960f0c3 
RED: Models mapper tests  crishianfms committed 4 days ago	 f6e3df2 
GREEN: created purchase okey  crishianfms committed 4 days ago	 a1c6792 
REFACTOR: fix names of some controllers  crishianfms committed 4 days ago	 48ce374 

GREEN: GetSolicitudesOK and ModelsMapper with test ok federicocz committed 4 hours ago	231574b	<>
RED: solicitud controller and models federicocz committed 6 hours ago	3b1faf7	<>
RED: Get solicitudes test controller federicocz committed 6 hours ago	5e4fcbb	<>

2.5 Otros aspectos Clean Code de consistencia

A continuación evidenciamos algunas concordancias y consistencias varias que se deben tener en cuenta a la hora de respetar clean code.

- Las variables deben comenzar con minúscula.

```
{
    Invitation invitation = new Invitation()
    {
        UserName = "Cris01",
    }
}
```

métodos deben comenzar con mayúscula.

- Los

```
public virtual User GetUserByUserName(string userName)
```

- Las llaves de apertura y clausura sean utilizadas exclusivamente en una línea aparte.

```
public virtual Role GetRoleByName(string roleName)
{
    return _roleRepository.GetFirst(r => r.Name == roleName);
}
```

- Las properties deben de comenzar con mayúscula.

```
public class User
{
    public int Id { get; set; }
    public string UserName { get; set; }
    public int RoleId { get; set; }
    public Role Role { get; set; }
    public string Email { get; set; }
    public string Address { get; set; }
    public string Password { get; set; }
    public int? PharmacyId { get; set; }
    public Pharmacy Pharmacy { get; set; }
    public DateTime RegistrationDate { get; set; }
}
```

- Las interfaces deben de comenzar con la letra I


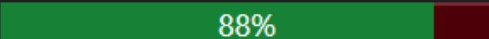

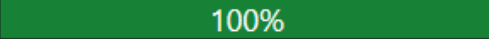

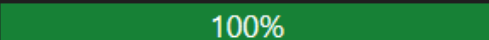

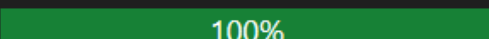

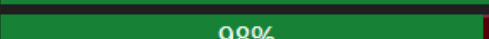

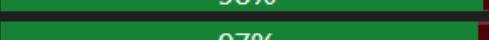

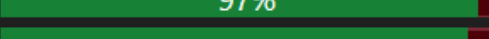

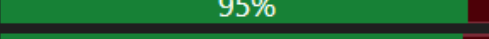

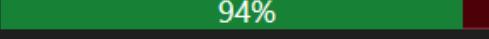



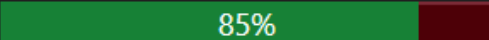

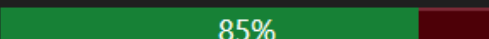

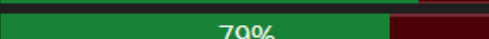

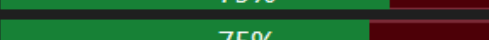

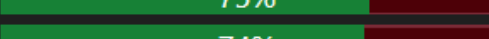

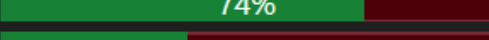

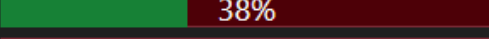

```
namespace IDataAccess
{
    public interface ISessionRepository
    {
        Session Create(Session session);
        Session FindSessionByUserId(int id);
    }
}
```

3. Ejecución de los test

3.1 Informe de cobertura para todas las pruebas desarrolladas

Para realizar el análisis de cobertura del programa se uso una aplicación llamada JetBrains dotCover, la cual es compatible con Visual Studio y permite determinar la cobertura de las pruebas unitarias.

Ocurrió un desvío del 80% total esperado ya que no probamos ni las migrations ni la factory, ya que las migrations no se prueban y la factory fue una clase creada por nosotros para desacoplar las inyecciones de la API, y como se observa salvo por las pruebas del dominio las demás clases de prueba superan el 90% de cobertura.

Symbol	Coverage (%) ▼	Uncovered/Total Stmts.
▲  Total	88% 	348/2863
▶  WebApi.Test	100% 	0/189
▶  WebApi.Models.Test	100% 	0/108
▶  WebApi.Filter.Test	100% 	0/105
▶  WebApi.Filter	98% 	1/65
▶  DataAccess.Test	97% 	11/343
▶  BusinessLogic.Test	95% 	20/366
▶  AuthLogic.Test	94% 	13/203
▶  AuthLogic	87% 	25/199
▶  Domain	85% 	58/398
▶  BusinessLogic	85% 	53/349
▶  Domain.Test	79% 	22/103
▶  WebApi	75% 	29/115
▶  WebApi.Models	74% 	69/264
▶  Exceptions	38% 	15/24
▶  Factory	0% 	31/31
▶  Migrations	0% 	1/1