

Universidad ORT Uruguay

Obligatorio 1 **Diseño de Aplicaciones 2**

Agustina Disiot 221025

Ivan Monjardin 239850

<https://github.com/ORT-DA2/Disiot-221025-Monjardin-239850>

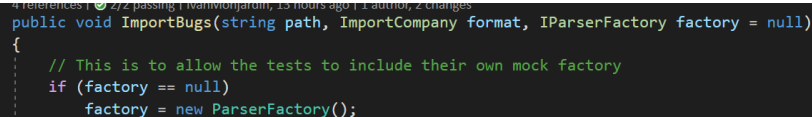
Índice:

Clean Code	3
TDD	4
Estrategia de TDD	4
Fotos de los commits de las funcionalidades con *	5
Proyecto	5
Creación de Usuario	5
Cantidad de bug por proyecto	6
Mantenimiento de un bug	6
Partes del código que no son cubiertas por las pruebas	7
Factory de Parser en business logic	7
Repository design y migrations	7
Excepciones	7
WebApi	7
Comparers	8
Factory bug parser	8

Clean Code

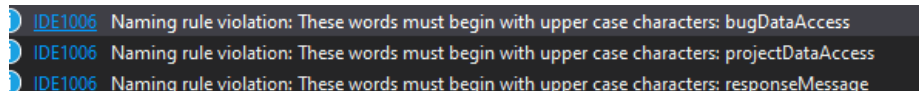
Desde el principio se siguieron las recomendaciones de Clean Code con el objetivo de que otra persona pueda entender nuestro código sin nuestra ayuda. Se tomaron en cuentas reglas como:

- Evitar magic numbers
- Comentarios para explicar el por que y no el cómo



```
public void ImportBugs(string path, ImportCompany format, IParserFactory factory = null)
{
    // This is to allow the tests to include their own mock factory
    if (factory == null)
        factory = new ParserFactory();
}
```

- Se usaron enums para hacer el código más legible (como en la foto de arriba)
- Se usaron variables descriptivas con nombres descriptivos
- Uso de inyección de dependencias
- Ley de Demeter: una clase sólo conoce sus dependencias directas
- Evitamos dependencia lógica. No escribimos métodos que funcionan correctamente solo cuando dependen de algo más de la misma clase.
- Utilizamos nombres descriptivos y nemotécnicos
- Los nombres de métodos y variables son pronunciables.
- Las funciones realizan solo una cosa, tienen una sola responsabilidad.
- Funciones con pocos parámetros.
- Funciones cortas *
- No se usan métodos estáticos
- Utilizamos un Linter para detectar posibles fallos en nuestro código y poder arreglarlo, por ejemplo:



```
IDE1006 Naming rule violation: These words must begin with upper case characters: bugDataAccess
IDE1006 Naming rule violation: These words must begin with upper case characters: projectDataAccess
IDE1006 Naming rule violation: These words must begin with upper case characters: responseMessage
```

- Se utilizó Code Cleanup de Visual Studio para eliminar using que no usamos y mejorar la consistencia del código en general

En cuanto a los tests:

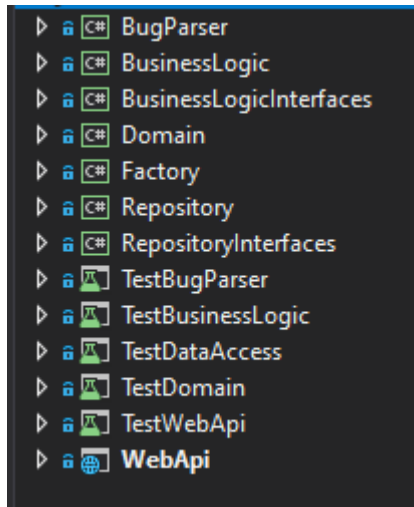
- Un assert por test, al menos que sea para se para testear dos características de la misma prueba (ej. el código y el tipo de respuesta de la web api).
- Independientes (uso de mocking)
- Repetibles
- Leibles
- Rápidos, controlamos el tiempo que muestra visual studio

* Algunas funciones de Testing son más largas ya que se crean varios objetos y eso puede llevar más líneas.

TDD

Se utilizó TDD a lo largo del proyecto, en su gran mayoría para pruebas unitarias ya que se utilizaron mocks para independizar las pruebas. De esta forma se logró probar las funcionalidades independientes de otras de otras capas o de la misma capa.

Se mantuvieron separados los proyectos de Testing y los que son testeados. No se pone el Proyecto de Test en la misma clase que testea, sino en un proyecto a parte.



Separación de proyectos de Test y los testeados

Estrategia de TDD

Al comienzo nos planteamos realizar pruebas transversales de una funcionalidad, en nuestro caso Bug. Para este caso comenzamos por la clase de Dominio - BusinessLogic - WebApi - DataAccess

Una vez pronta esa etapa, se desarrolló gran parte del dominio y una parte de la business logic.

Después las siguientes funcionalidades se agregaron en su mayoría empezando por WebApi-BusinessLogic hasta DataAccess

Realizamos más una mezcla entre ambas, pero se podría decir que al principio hicimos inside in y al final hicimos más outside in.

Además se estableció una convención para los commits de tdd:

Nombre del commit. Stage red

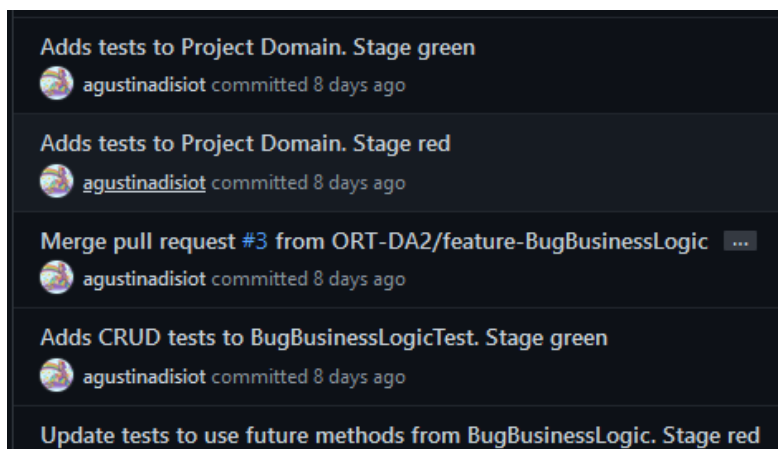
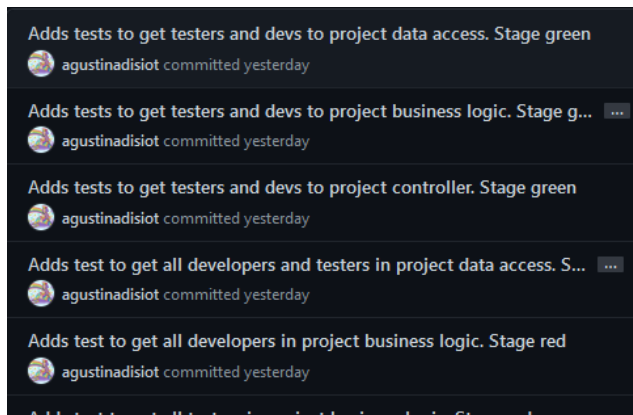
Nombre del commit. Stage green

Nombre del commit. Stage refactor

En comparación a diseño 1, esta vez realizamos más pruebas en etapa “red” antes de pasar a la green, por lo tanto cuando pasamos a la green a veces era más complicado encontrar una solución que precise mucho refactor.

*Fotos de los commits de las funcionalidades con **

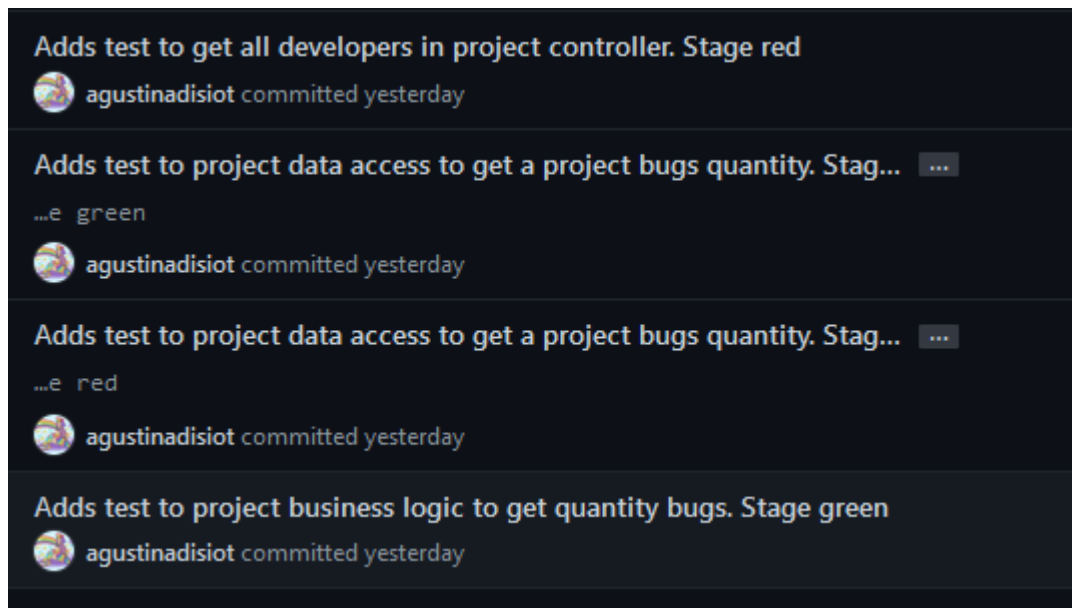
Proyecto



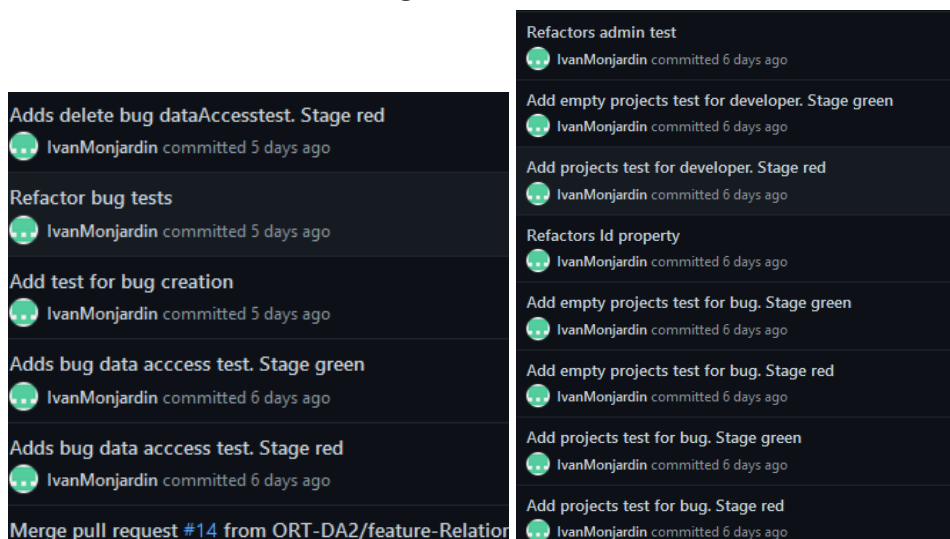
Creación de Usuario



Cantidad de bug por proyecto



Mantenimiento de un bug



Partes del código que no son cubiertas por las pruebas

Antes de mostrar el code coverage completo, vamos a explicar algunas cosas donde no se llegó a más de 90, como por ejemplo, algunas clases vinculadas a los Data Access.

Factory de Parser en business logic

Esto se debe a que se incluye un parámetro opcional que sea utilizado solo por las pruebas. Ya que el factory de que parser usar (xml, txt etc.) es uno y queda fijo en el código. Entendemos que no es ideal cambiar el código por las pruebas, pero no nos pareció adecuado testear de nuevo la factory y los parseo ya testeados en otro proyecto.

La idea era testear como business logic utilizar la factory y el parseo, no si el parseo en sí funciona bien. Por eso se utilizaron mocks para no estar testeando dos cosas a la vez

```
public void ImportBugs(string path, ImportCompany format, IParserFactory factory = null)
{
    // This is to allow the tests to include their own mock factory
    if (factory == null)
        factory = new ParserFactory();
    IBugParser parser = factory.GetBugParser(format);
    List<Bug> bugsToImport = parser.GetBugs(path);
    foreach (var bug in bugsToImport)
    {
        // ...
    }
}
```

Repository design y migrations

▷ { } Repository.Design	10	100,00 %	0	0,00 %
▷ { } Repository.Migrations	1895	100,00 %	0	0,00 %

Ni el Design ni las migrations se testearon ya que son o creadas semi automáticamente o son utilizados solo por el código en “producción” y no por los tests donde se utiliza una base de datos en memoria.

Excepciones

```
[TestMethod]
public void InvalidXML()
{
    string fullPath = baseDirectory + "InvalidXML.xml";
    Assert.ThrowsException<XmlException>(() => bugParser.GetBugs(fullPath));
}
```

Algunos tests que utilizan excepciones aparecen como no cubierto la función que se llama para que tire la excepción. No tenemos del todo claro porque sucede.

WebApi

Algunas clases de WebApi como Startup ya venían o no ameritaba tests. Los controladores si están 100% cubiertos.

WebApi	62	100,00 %
Program	10	100,00 %
Program.<>c	2	100,00 %
Startup	33	100,00 %
Startup.<>c	9	100,00 %

Comparers

En Dominio incluimos comparadores (ya que consideramos que dominio es el experto sobre si dos clases tuyas son o no iguales), que utilizamos para tests y no todas llegan al 100% Sin embargo Dominio como tal, que no es usado para tests, sí está en 100%

domain.dll	24	12,06 %	175	87,94 %
Domain.Utils	24	19,83 %	97	80,17 %
Domain	0	0,00 %	78	100,00 %
bugparser.dll	6	9,09 %	60	90,91 %

Factory bug parser

El diseño original iba a utilizar clases estáticas, ya que, por ejemplo, no se pensaba guardar ningún dato. Sin embargo al tener que realizar un test en business logic y poder utilizar mocks tuvimos que cambiar el código de la misma manera que explicamos arriba en el factory en business logic.












Dicho factory es la única clase que no tiene 100% de cobertura dentro del paquete de bug parser

bugparser.dll	6	9,09 %	60	90,91 %
BugParser	6	9,09 %	60	90,91 %
ParserFactory	6	100,00 %	0	0,00 %
GetBugParser(Do...	6	100,00 %	0	0,00 %
BugModel	0	0,00 %	34	100,00 %
BugParserXML	0	0,00 %	16	100,00 %
BugXML	0	0,00 %	8	100,00 %
BugsXML	0	0,00 %	2	100,00 %

Tampoco se toma en cuenta el cover coverage de las clases de Test

Esas son todas las partes donde no hay cobertura de más o igual a 90%

Code Coverage Final

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
▸  bugparser.dll	6	9,09 %	60	90,91 %
▸  businesslogic.dll	2	1,32 %	150	98,68 %
▸  businesslogicinterfaces.dll	1	10,00 %	9	90,00 %
▸  domain.dll	30	14,22 %	181	85,78 %
▸ { } Domain	0	0,00 %	78	100,00 %
▸ { } Domain.Utils	30	22,56 %	103	77,44 %
▸  repository.dll	1925	80,65 %	462	19,35 %
▸ { } Repository	20	4,15 %	462	95,85 %
▸ { } Repository.Design	10	100,00 %	0	0,00 %
▸ { } Repository.Migrations	1895	100,00 %	0	0,00 %
▸  testbugparser.dll	2	2,53 %	77	97,47 %
▸  testbusinesslogic.dll	8	0,74 %	1073	99,26 %
▸  testdataaccess.dll	13	1,56 %	820	98,44 %
▸  testdomain.dll	0	0,00 %	301	100,00 %
▸  testwebapi.dll	1	0,11 %	907	99,89 %
▸  webapi.dll	79	39,50 %	121	60,50 %
▸ { } WebApi	61	100,00 %	0	0,00 %
▸ { } WebApi.Controllers	0	0,00 %	121	100,00 %
▸ { } WebApi.Filters	18	100,00 %	0	0,00 %