

# Universidad ORT Uruguay

Docente: Gabriel Piffaretti

## **Obligatorio 2**

### **Diseño de Aplicaciones 2**

Agustín Ferrari 240503

Francisco Rossi 219401

<https://github.com/ORT-DA2/240503-219401>

## *Índice:*

<b>Evidencia de la aplicación de TDD y Clean Code</b>	<b>3</b>
Mantenimiento de un Bug	4
Agregar un bug	4
Modificar un Bug	8
Mantenimiento de Proyecto	12
Borrar un proyecto DataAccess	12
Borrar Bugs de un proyecto BusinessLogic	14
Cantidad de bugs por proyecto	15
Clean Code	15
Nombres	15
Funciones	15
Comentarios	16
Formato	16
Manejo de errores Excepciones	17
Teste Coverage	17

## *Evidencia de la aplicación de TDD y Clean Code*

En este proyecto el equipo se propuso como objetivo, desarrollar código profesional, fácil de entender, consistente y extensible. Para llevar a cabo esto, se implementaron varias técnicas aprendidas en otros cursos, como, Test Driven Development (TDD), recomendaciones de Clean Code, entre otras.

La idea fue seguir estrictamente TDD, plantear un test que no anduviera, escribir el código mínimo para que esta pasara, y luego realizar un refactor. Esto impedía generar código sin que exista una prueba para ello. Esto genera una ventaja enorme, al saber que los métodos que el equipo iba a utilizar funcionaban de forma correcta o existía una prueba que lo respalde. Se siguió una estrategia mixta (inside-out y outside-in) dependiendo la funcionalidad, y cómo evaluamos mejor cada caso.

Sin embargo, algunas clases no tienen pruebas generadas por el equipo. Clases como Startup, Program del paquete WebApi o las migraciones que generamos para actualizar la base de datos. Esto influyó de manera negativa en el test coverage de nuestro proyecto, pero dado que es código externo al desarrollado por el equipo propiamente, no requiere acción sobre el tema.

## *Test Driven Development (TDD)*

### Mantenimiento de un Bug

Se planteó la funcionalidad de poder crear, modificar y eliminar un bug del sistema a un tester, con la restricción de que el usuario con el rol de tester, debía ser parte del proyecto al cual el bug iba a tener una de las 3 acciones posibles.

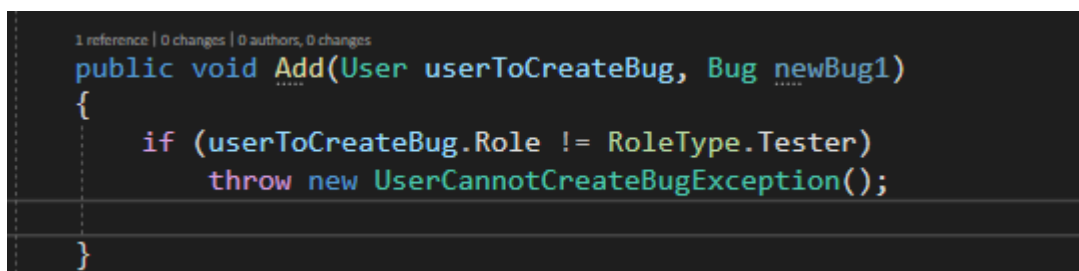
Entendimos que la primera acción a realizar era chequear que el rol del usuario a crear un bug debía ser tester. Por lo que decidimos crear un test donde se espera lanzar una excepción sobre esta restricción de roles sobre la acción. Este test se crea en un estado rojo (red stage) según TDD, lo que significa que al momento de escribir el código del test, no compila, la excepción no existe ni tampoco la funcionalidad a testear.

Se comenzaron los test para crear un bug en el proyecto `DataAccessTest`, especializado en encapsular las pruebas por entidad del dominio.

### Agregar un bug

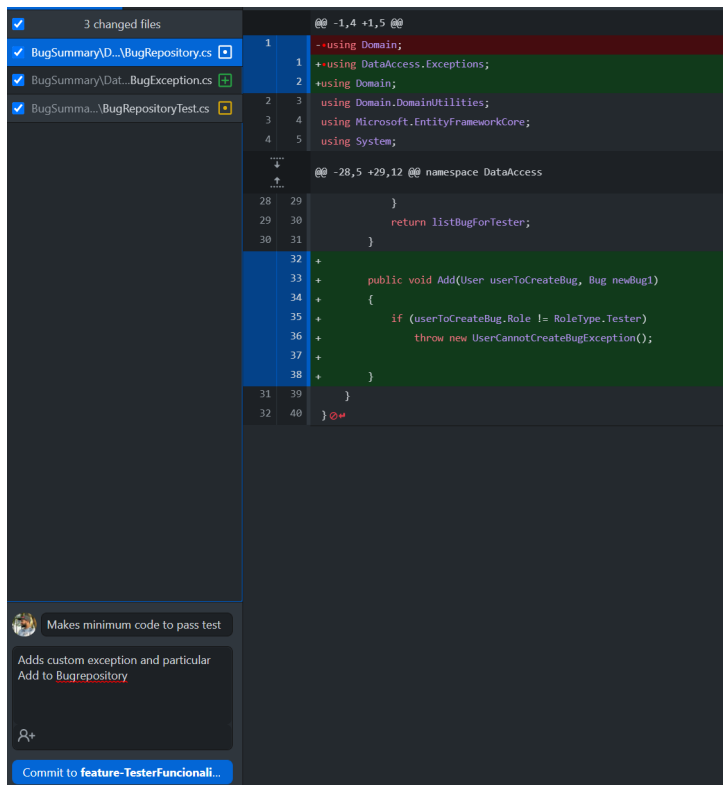
Se agrega el test en estado rojo, sin que este compile dado que la excepción no existía ni el método del `BugRepository` `add`, se crea un mensaje, indicando que hace el test, en que estado esta para finalmente realizar el primer commit.

Como el test espera a que se lance una excepción si el usuario que desea agregar un bug no tiene un rol de tester, lo único que se permite hacer es el mínimo código posible para lanzar esta excepción si no se cumple la restricción.



```
1 reference | 0 changes | 0 authors, 0 changes
public void Add(User userToCreateBug, Bug newBug1)
{
    if (userToCreateBug.Role != RoleType.Tester)
        throw new UserCannotCreateBugException();
}
```

Se pudo haber solo lanzado la excepción y luego realizar un refactor, pero nos pareció adecuado ya agregar el `if`, para chequear el rol del usuario, porque finalmente ese es el objetivo final de la prueba.



```
@@ -1,4 +1,5 @@
1  -using Domain;
1  +using DataAccess.Exceptions;
2  +using Domain;
2  using Domain.DomainUtilities;
3  using Microsoft.EntityFrameworkCore;
4  using System;
5

...
@@ -28,5 +29,12 @@ namespace DataAccess
28  }
29  return listBugForTester;
30  }
31
32  +
33  + public void Add(User userToCreateBug, Bug newBug1)
34  + {
35  +     if (userToCreateBug.Role != RoleType.Tester)
36  +         throw new UserCannotCreateBugException();
37  +
38  + }
39  }
40  }
```

Makes minimum code to pass test

Adds custom exception and particular  
Add to [Bugrepository](#)

Commit to feature-TesterFuncionali...

Se crea un nuevo mensaje indicando lo que se realizó en esta etapa de TDD (green stage) para que la prueba pasará. Además de agregar el código para pasar la prueba, se creó la excepción para que indica que el usuario no es un tester, por lo que no tiene permisos para realizar la acción. La excepción tiene un mensaje personalizado, para poder usar ese mensaje en otra capa de ser necesario.

Esta parte del código solo lanza la excepción, todavía no se agrega el bug al data access, que es la funcionalidad que queremos. Para esto necesitamos otra prueba que nos haga fuerce a generar esa parte del código.

Aunque esta etapa no sea la final ya se pudo realizar un cambio en otro archivo, se agregó la firma del método a la interfaz que utiliza el BugRepository, dado que los parámetros se mantendrán estables.

Para finalizar un ciclo de TDD (blue stage), se realiza un refactor sobre el test original, para disminuir la cantidad de líneas del mismo y presentar más simple la prueba. Lo interesante de este refactor es que sin cambiar ninguna línea el código de la funcionalidad, sigue pasando el test.

Luego para generar la funcionalidad, se creó otro test para efectivamente agregar un bug a un proyecto, siendo un tester y que el usuario contenga ese proyecto. Este test es más complejo, porque se quiso probar generar el objeto User, con rol tester, un proyecto, el cual tiene ese usuario tester como una property de él y su id es la misma a la que contiene el bug como FK a Project.

```
266 +         {
267 +             testerUser
268 +         }
269 +     };
270 +     testerUser.Projects.Add(projectTester);
271 +     Bug newBug = new Bug
272 +     {
273 +         Id = 1,
274 +         Name = "Bug1",
275 +         Description = "Bug en el servidor",
276 +         Version = "1.4",
277 +         State = BugState.Active,
278 +         Project = projectTester,
279 +         ProjectId = 1
280 +     };
281 +     _bugRepository.Add(testerUser, newBug);
282 +     _bugSummaryContext.SaveChanges();
283 +
284 +     List<Bug> bugsExpected = new List<Bug>()
285 +     {
286 +         new Bug
287 +         {
288 +             Id = 1,
289 +             Name = "Bug1",
290 +             Description = "Bug en el servidor",
291 +             Version = "1.4",
292 +             State = BugState.Active,
293 +             Project = projectTester,
294 +             ProjectId = 1
295 +         }
296 +     };
297 +     List<Bug> bugsDataBase = this._bugRepository.GetAllByTester(testerUser).ToList();
298 +
299 +     using (var context = new BugSummaryContext(this._contextOptions))
300 +     {
301 +         Assert.AreEqual(1, bugsDataBase.Count());
302 +         CollectionAssert.AreEqual(bugsExpected, bugsDataBase, new BugComparer());
303 +     }
304 +
305 + }
306 +
```

Create test Tester add bug

Create test for adding a bug only if user is tester, stage red

Commit to feature-TesterFuncionali...

Decidimos utilizar using context, para que los resultados a comparar en los asserts, no sean datos guardados en memorias, sino que efectivamente crea un contexto para traerlo del Dbset de bugs, confirmando el correcto funcionamiento.

Como primer acercamiento realizamos un foreach Project en los proyectos del usuario, para confirmar si la FK entre esta relación de un proyecto y un bug existía, y si la encontraba la agregaba al DbSet de bugs que se encuentra en Context. (Context.Bugs.Add(newBug).

```
30 30 @@ -30,11 +30,19 @@ namespace DataAccess
31 31         return listBugForTester;
32 32     }
33 33 - public void Add(User userToCreateBug, Bug newBug1)
34 34 + public void Add(User userToCreateBug, Bug newBug)
35 34 {
36 35 - if (userToCreateBug.Role != RoleType.Tester)
37 35 + if (userToCreateBug.Role == RoleType.Tester)
38 36 {
39 37     foreach (Project project in userToCreateBug.Projects)
40 38     {
41 39         if (project.Id == newBug.ProjectId)
42 40             Context.Bugs.Add(newBug);
43 41     }
44 42 }
45 43 else
46 44     throw new UserCannotCreateBugException();
47 45 }
48 46 }
49 47 }
50 48 }
```

Creates test for passing both test

Check if user trying to create bug is tester and has that project

Commit to feature-TesterFuncionali...

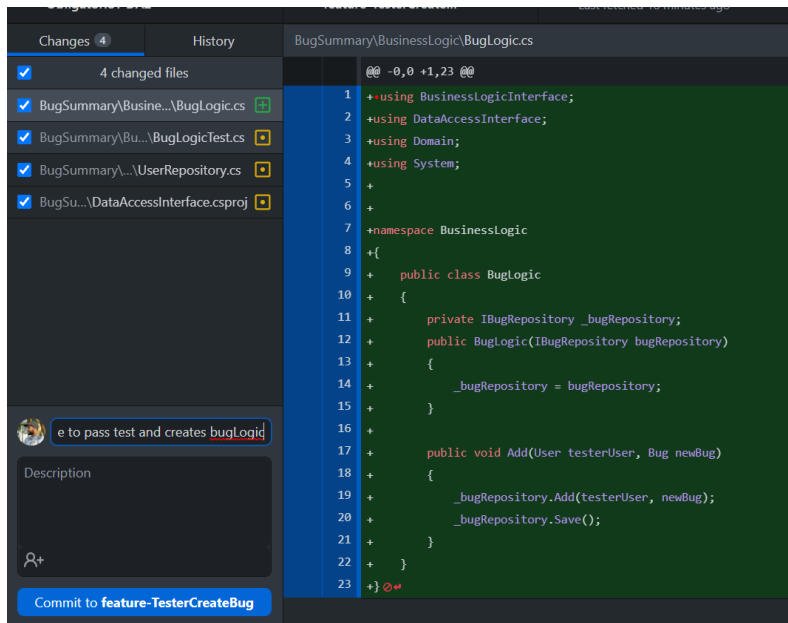
Se crea el código, sin afectar la primera prueba creada, para chequear el rol de usuario.

Se realizó otro refactor para incluir que se esté lanzando correctamente la excepción deseada y cuente en el test coverage de los tests. Indagamos para poder utilizar una herramienta del framework y encontramos `Assert.Throws<CustomException>`, pero no pudimos hacer que se cuente en el test coverage. A partir de esta situación se creó una clase para testear que el mensaje que lanza una excepción sea, la personalizada que nosotros decidimos indicar y cuente en el test coverage.

Por ejemplo, se puede utilizar de esta manera, sin realizar un assert.

```
TestExceptionUtils.Throws<ProjectNameIsNotUniqueException>(
    () => _projectRepository.Add(projectToAdd), "The project name chosen was already taken, please enter a different name"
);
```

Luego para avanzar en entre capas del sistema, se crea un test en `BugLogicTest` para poder definir un método, que termina agregando el bug desde la lógica de negocio en el repositorio de Bugs.



De esta manera, el primer test, va haciendo que la funcionalidad se propague hacia las otras capas, terminando en un endpoint de Post de bugs.

El código de agregar un bug no quedó fijo, a medida que se iban creando ramas nuevas, o se analizaba de otra forma la funcionalidad, se realizaban cambios en los test, luego en el código, siguiendo los estándares de TDD. Finalmente, obtuvimos este código:

```
public void Add(User userToCreateBug, Bug newBug)
{
    Project userProject = userToCreateBug.Projects.Find( match: p:Project => p.Id == newBug.ProjectId);
    if (userProject == null)
        throw new ProjectDoesntBelongToUserException();
    Context.Bugs.Add(newBug);
}
```

El código es mucho más simple de entender y prolijo, cumpliendo una sola responsabilidad. No es que simplemente se eliminó el chequeo de que el rol del usuario debía ser Tester, sino que esta responsabilidad, se delegó para que cualquier user que llegue a este punto del código debe de ser un tester. Esto se implementó con un authorization filter en la capa de WebApi.

## Modificar un Bug

Continuando con la relación entre capas vamos a analizar el desarrollo de esta funcionalidad para modificar un update, con la condición de que el user debe ser un tester.

```
}  
Bug updatedBug = new Bug  
{  
    Id = 1,  
    Name = "BugNuevo",  
    Description = "Bug Nuevo",  
    Version = "1.5",  
    State = BugState.Done,  
    ProjectId = 1  
};  
_bugRepository.Update(testerUser, updatedBug);  
_bugRepository.Save();  
  
using (var context = new BugSummaryContext(this._contextOptions))  
{  
    Bug databaseBug = context.Bugs.ToList().First(u:Bug => u.Id == updatedBug.Id);  
    CompareLogic compareLogic = new CompareLogic();  
    ComparisonResult deepComparisonResult = compareLogic.Compare(expectedObject: updatedBug, actualObject: databaseBug);  
    Assert.IsTrue(deepComparisonResult.AreEqual);  
}
```

Este fue el primer test que se realizó sobre update, se vuelve a utilizar el using context para corroborar que se esté modificando correctamente y no se esté comparando con datos en memoria.

A continuación se procede a crear el método Update con 2 parámetros en el BugRepository y crear el mínimo código posible para que pase la prueba. Utilizamos un CompareLogic para poder comparar todas las properties de dos entidades sin tener que definirse un Comparer para cada uno.

Aún sin hacer el código necesario para pasar la prueba, se crea otro test que prueba una excepción personalizada que se quiere lanzar en caso de que no exista el bug a actualizar.

En este prueba, ya utilizamos la clase propia para testear la excepción a lanzar y la creamos para que al crear el código necesario ya de correcto el assert.

```
public void Update(User testerUser, Bug updatedBug)  
{  
    Bug bugFromDb = Context.Bugs.Include("Project").FirstOrDefault(u:Bug => u.Id == updatedBug.Id);  
    if (bugFromDb != null)  
    {  
        bugFromDb.Name = updatedBug.Name;  
        bugFromDb.Description = updatedBug.Description;  
        bugFromDb.ProjectId = updatedBug.ProjectId;  
        bugFromDb.Version = updatedBug.Version;  
        bugFromDb.State = updatedBug.State;  
        Context.Bugs.Update(bugFromDb);  
    }  
    else  
        throw new InexistentBugException();  
}
```



Generamos el código necesario para que lance la excepción si no existe el bug a actualizar y también a su vez pase la primera prueba que creamos para generar el método de Update. Se crea otro test más para controlar la misma condición que requería el agregar un bug, que el user tenga un rol de Tester.

Avanzamos en la siguiente etapa del TDD y escribimos el código para que pase la prueba.

```
if (testerUser.Role != RoleType.Tester)
    throw new UserMustBeTesterException();
Bug bugFromDb = Context.Bugs.Include("Project").FirstOrDefault(u:Bug => u.Id == updatedBug.Id);
if (bugFromDb != null)
{
```

Luego se crea otro test más con el mismo objetivo, lanzar una excepción en el caso de que el usuario no pertenezca al proyecto el cuál el bug se encuentra.

```
TestExceptionUtils.Throws<ProjectDontBelongToUser>()
{
    action: () => _bugRepository.Update(testerUser, updatedBug), message: "New project to update bug, does not belong to tester"
};
```

Todos estos test se crean para que casos bordes como por ejemplo que el bug no exista en la base de datos, y no se quiera acceder a una property de un objeto null resultando en una excepción y posible caída del sistema.

```
4 usages  FranRossi *
public void Update(User testerUser, Bug updatedBug)
{
    if (testerUser.Role != RoleType.Tester)
        throw new UserMustBeTesterException();
    if (testerUser.Projects.Find(match: p:Project => p.Id == updatedBug.ProjectId) == null)
        throw new ProjectDontBelongToUser();
    Bug bugFromDb = Context.Bugs.Include("Project").FirstOrDefault(u:Bug => u.Id == updatedBug.Id);
    if (bugFromDb != null)
    {
        bugFromDb.Name = updatedBug.Name;
        bugFromDb.Description = updatedBug.Description;
        bugFromDb.ProjectId = updatedBug.ProjectId;
        bugFromDb.Version = updatedBug.Version;
        bugFromDb.State = updatedBug.State;
        Context.Bugs.Update(bugFromDb);
    }
    else
        throw new InexistentBugException();
}
```

Generamos el código necesario para que pase todos los test y lance las excepciones correspondientes según el caso.

Luego se actualizó la interfaz de DataAccess correspondiente, para poder exponer este método hacia la otras capas.

```

[DataRow("1pojJYCG2Uj8WMMXBteJYRqqcJZIS3dNL")]
[DataTestMethod]
new *
public void UpdateValidBug(string token)
{
    Bug updatedBug = new Bug
    {
        Id = 1,
        Name = "BugNuevo",
        Description = "Bug en el cliente",
        Version = "1.5",
        State = BugState.Done,
        ProjectId = 1
    };
    Bug sentBugToBeUpdated = null;
    Mock<IBugRepository> mockBugRepository = new Mock<IBugRepository>(MockBehavior.Strict);
    mockBugRepository.Setup(expression: mr:IBugRepository => mr.Update(It.IsAny<User>(), It.IsAny<Bug>())) //ISetup<IBugRepository>
        .Callback((Bug bug) =>
        {
            sentBugToBeUpdated = bug;
        });
    mockBugRepository.Setup(expression: mr:IBugRepository => mr.Save());
    Mock<IUserRepository> mockUserRepository = new Mock<IUserRepository>(MockBehavior.Strict);

    BugLogic bugLogic = new BugLogic(mockBugRepository.Object, mockUserRepository.Object);
    bugLogic.Update(token, updatedBug);

    mockBugRepository.VerifyAll();
    Assert.AreEqual(expected: updatedBug, actual: sentBugToBeUpdated);
}

```

En BussinesLogicTest, creamos un test, donde se realizan mocks del Update que recién se creó. Decidimos usar .CallBack, para simular el bug que fue enviado a modificarse y poder testearlo como una variable en el assert. El token identifica un usuario único en la base de datos, por lo que moqueamos el repositorio de usuario para poder obtener el tester, con ese token en particular.

Buglogic no cuenta con la implementación de Update, pero lo se espera es que se comunique con DataAccess para modificar el bug por un tester, para eso tiene que acceder al repositorio de bugs.

```

1 usage new *
public void Update(string token, Bug updatedBug)
{
    UserLogic userLogic = new UserLogic(_userRepository);
    User userByToken = userLogic.Get(token);
    _bugRepository.Update(userByToken, updatedBug);
    _bugRepository.Save();
}

```

Luego de obtener el user, para corroborar si cuenta con el proyecto que el bug va a ser modificado, se llama a la función Update del DataAccess.

Moqueamos que tire la excepción de que un bug no existe para identificar que las dependencias entre capas para las excepciones estén funcionando correctamente.

```

[DataRow("1pojjYC62Uj8WMXBteJYRqgcJZIS3dNL")]
[DataTestMethod]
[TestMethod]
public void UpdateValiBug(string token)
{
    BugModel bug = new BugModel
    {
        Name = "Bug2021",
        Description = "ImportanteBug",
        State = BugState.Active,
        Version = "2",
        ProjectId = 1,
    };
    Mock<IBugLogic> mock = new Mock<IBugLogic>(MockBehavior.Strict);
    User receivedBug = null;
    mock.Setup(expression: m => m.Update(It.IsAny<string>(), It.IsAny<User>())).Callback((string token, User sentBug) =>
    {
        receivedBug = sentBug;
    });
    BugsController controller = new BugsController(mock.Object);

    IActionResult result = controller.Post(bug);

    mock.VerifyAll();
    Assert.IsInstanceOfType(result, typeof(OkResult));
    CompareLogic compareLogic = new CompareLogic();
    ComparisonResult deepComparisonResult = compareLogic.Compare(expectedObject: bug.ToEntity(), actualObject: receivedBug);
    Assert.IsTrue(deepComparisonResult.AreEqual);
}

```

Por último llegamos al proyecto de WebApiTest, donde creamos una test, para probar a partir de un BugModel que recibimos del cliente, transformarlo a una entidad del dominio, y poder modificar un bug. En este caso moqueamos IBugLogic la interfaz que expone el método de update. Definimos el Post que devuelve un resultado de la operación o acción a realizar, y luego hicimos un refactor para que esta función recibiera un token para poder encontrar el user, con sus proyectos y rol.

También este test, nos hizo definir un BugModel y sus funcionalidades para convertir un modelo que se envía a partir de un JSON en el Postman, a una entidad del dominio.

ToEntity() la creamos a partir de que queremos comparar entre bugs no entre modelos. No cuenta con un seteo de Id del bug porque esto lo definimos que sea haga de manera automática en la base de datos.

Luego se expuso el Update en el IBugLogic para que se pueda utilizar en el proyecto de WebApi. Finalmente generamos un refactor para cambiar a Put en vez de Post y recibe un token del header, para autenticarse.

Se agregó un test para verificar que solo un developer pueda hacer esta acción, y esto se hizo en otras ramas, agregando filtros de autorización, por lo que se sacó el chequeo en DataAccess y el método Put se le agregó un filtro.

```

[HttpPut]
[AuthorizationWithParameterFilter(argument: new[]{RoleType.Tester})]
public IActionResult Put([FromHeader] string token, [FromBody] BugModel bug, [FromQuery] int bugId)
{
    _bugs.Update(token, bug.ToEntityWithID(bugId));
    return Ok();
}

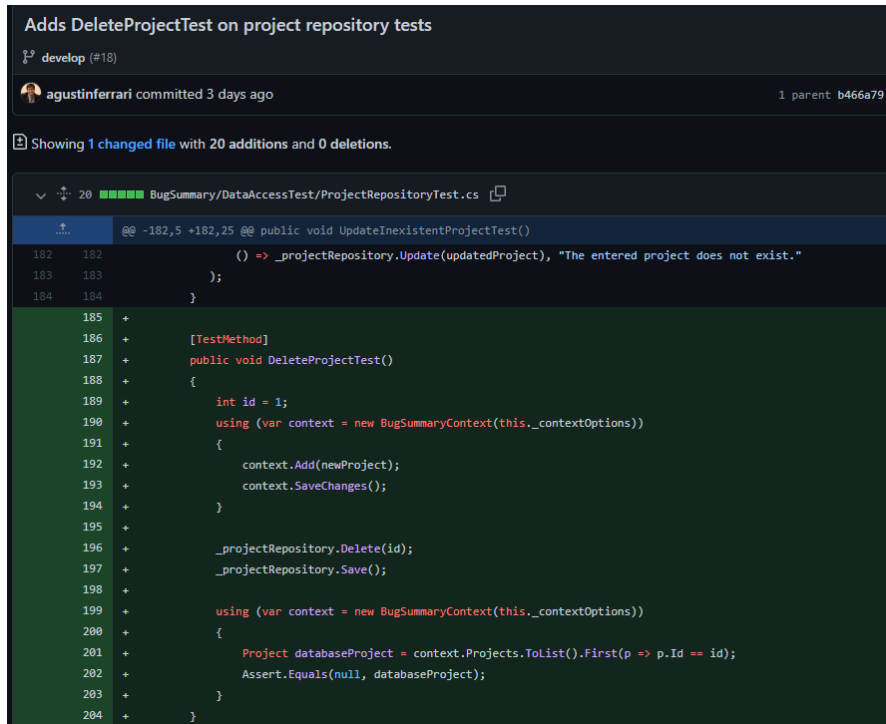
```

Para borrar un bug, se utilizó la mismas estrategias ya planteadas, empezando desde el DataAccess hasta llegar a la WebApi, pasando un token y un id de bug para borrar.

## Mantenimiento de Proyecto

### Borrar un proyecto DataAccess

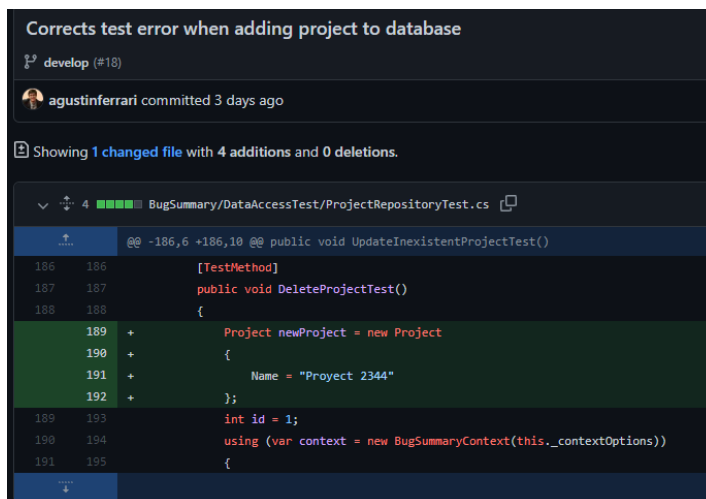
Se muestra una serie de commits del repositorio en github para ver el avance de uno de los integrantes del equipo aplicando TDD para crear el código en DataAccess, con mensajes en los commits indicando que se realizó en cada paso.



The screenshot shows a GitHub commit titled "Adds DeleteProjectTest on project repository tests" by user agustinferrari. The commit message is "Adds DeleteProjectTest on project repository tests". The commit shows a change in the file BugSummary/DataAccessTest/ProjectRepositoryTest.cs. The code snippet shows the addition of a new test method, DeleteProjectTest, which tests the Delete method of the ProjectRepository. The method uses a context to add a new project, then calls Delete, and finally asserts that the project is null in the database.

```
@@ -182,5 +182,25 @@ public void UpdateInexistentProjectTest()
182 182     () => _projectRepository.Update(updatedProject), "The entered project does not exist."
183 183     );
184 184 }
185 +
186 + [TestMethod]
187 + public void DeleteProjectTest()
188 + {
189 +     int id = 1;
190 +     using (var context = new BugSummaryContext(this._contextOptions))
191 +     {
192 +         context.Add(newProject);
193 +         context.SaveChanges();
194 +     }
195 +
196 +     _projectRepository.Delete(id);
197 +     _projectRepository.Save();
198 +
199 +     using (var context = new BugSummaryContext(this._contextOptions))
200 +     {
201 +         Project databaseProject = context.Projects.ToList().First(p => p.Id == id);
202 +         Assert.Equals(null, databaseProject);
203 +     }
204 + }
```

Primer test para borrar un proyecto.



The screenshot shows a GitHub commit titled "Corrects test error when adding project to database" by user agustinferrari. The commit message is "Corrects test error when adding project to database". The commit shows a change in the file BugSummary/DataAccessTest/ProjectRepositoryTest.cs. The code snippet shows the correction of the DeleteProjectTest method, where the project name is set to "Proyect 2344" (note the typo in the original image).

```
@@ -186,6 +186,10 @@ public void UpdateInexistentProjectTest()
186 186     [TestMethod]
187 187     public void DeleteProjectTest()
188 188     {
189 +         Project newProject = new Project
190 +         {
191 +             Name = "Proyect 2344"
192 +         };
189 193     int id = 1;
190 194     using (var context = new BugSummaryContext(this._contextOptions))
191 195     {
```

Se arregla el test antes de codificar la función a borrar.

Implements delete method in project repository

develop (#18)

agustinferrari committed 3 days ago

Showing 1 changed file with 7 additions and 1 deletion.

```
BugSummary/DataAccess/ProjectRepository.cs
@@ -25,7 +25,6 @@ public override void Add(Project project)
    else
        throw new ProjectNameIsNotUniqueException();
    }
-
+    public override IEnumerable<Project> GetAll()
+    {
+        return Context.Projects.ToList();
+    }
@@ -42,5 +41,12 @@ public void Update(Project updatedProject)
    else
        throw new InexistentProjectException();
    }
+
+    public void Delete(int projectId)
+    {
+        Project projectFromDB = Context.Projects.FirstOrDefault(u => u.Id == projectId);
+        Context.Projects.Remove(projectFromDB);
+    }
```

Se implementa el código para que pase la prueba.

Creates DeleteInexistentProjectTest on project repository tests

develop (#18)

agustinferrari committed 3 days ago

Showing 1 changed file with 11 additions and 0 deletions.

```
BugSummary/DataAccessTest/ProjectRepositoryTest.cs
@@ -284,5 +284,16 @@ public void DeleteProjectWithUsersTest()
    Assert.IsTrue(deepComparisonResult.AreEqual);
    }
+
+    [TestMethod]
+    public void DeleteInexistentProjectTest()
+    {
+        int id = 1;
+
+        TestExceptionUtils.Throws<InexistentProjectException>(
+            () => _projectRepository.Delete(id), "The entered project does not exist."
+        );
+    }
```

Se crea test para tirar una excepción si el id del proyecto no existe en la base de datos.

```
Refactors delete method on project repository to throw exception if p...
...project does not exist

develop (#18)
agustinferrari committed 3 days ago 1 parent 7157b90

Showing 1 changed file with 6 additions and 1 deletion.

BugSummary/DataAccess/ProjectRepository.cs
@@ -46,7 +46,12 @@ public void Update(Project updatedProject)
46 46     public void Delete(int projectId)
47 47     {
48 48         Project projectFromDB = Context.Projects.FirstOrDefault(u => u.Id == projectId);
49 - Context.Projects.Remove(projectFromDB);
49 + if (projectFromDB != null)
50 + {
51 +     Context.Projects.Remove(projectFromDB);
52 + }
53 + else
54 + throw new InexistentProjectException();
50 55     }
51 56 }
52 57 }
```

Se hace un refactor para tirar la excepción.

Luego si se sigue el hilo de los commits, se puede ver cómo se llega hasta la WebApi de manera muy similar a como se mostró en el update de bugs.

## Borrar Bugs de un proyecto BusinessLogic

Ahora queremos hablar en una capa distinta (BusinessLogic) como se trató el borrado de bugs que pertenecen a un proyecto.

Se agregaron dos test para borrar un bug que su objetivo es implementar el código en BugLogic para comunicarse con el DataAccess y borrar el bug.

Se crearon dos test correspondientes para desarrollar esta funcionalidad, uno probando que la excepción de DataAccess llega hasta la WebApi sin problemas de dependencias.

Luego se generó el código correspondiente para utilizar para pasar las pruebas e implementar el código necesario.

Para poder acceder a los métodos de la capa de BusinessLogic desde WebApi se añade la firma a la interfaz para que la WebApi lo pueda utilizar sin conocer la implementación en sí.

Esta funcionalidad se puede encontrar con TDD en el repositorio

## Cantidad de bugs por proyecto

La implementación de esta funcionalidad no sigue los estándares de TDD con respecto a los commits (un commit por estado). Codificando se respetó el orden de iteración como veníamos trabajando, crear primera la prueba en esta rojo, generar el código para que pase y por último refactorizar, pero no se realizaron commits por cada stage, esto se realizó de esta manera priorizando terminar la funcionalidades del proyecto faltando 2 días al obligatorio, para poder dedicarle más tiempo a la documentación y test expiatorios. En el repositorio se puede encontrar que se creó esta funcionalidad 2 días antes de la entrega del obligatorio 6/10/2021.

No se violó TDD porque existen tests que respaldan todas las líneas del código de la funcionalidad, y se implementó a partir de esos test, no al revés. Pero los commits hacen varias etapas en una sola instancias.

## Clean Code

A medida que el código fue creciendo, en función de las pruebas y los principios de Clean Code, se fueron haciendo refactors para mejorar la solución.

## Nombres

Se trató de crear una regla para los nombres de variable o de funciones que revelen intención, que no se tenga que leer en tanto detalle el código para saber para qué era esa variable en específico.

Se definió un estándar para los nombres de los métodos que indiquen una acción, que sean breves y que no se repita sobre qué entidad se está realizando la acción ya que se encuentra bien dividido, ya sea el DataAccess, BusinessLogic o WebApi. Por ejemplo pueden haber distintos métodos que se llamen Update ambos, pero si identificamos en qué archivos nos encontramos, rápidamente sabes qué entidad del dominio queremos modificar.

En los proyectos de test, se trató de seguir estas reglas de rápida identificación de variables, como por ejemplo “testerUser”, para saber que es un user que tiene rol de tester.

Para nombrar los archivos del proyecto seguimos un estándar de nombres. Por cada capa (DataAccess, BusinessLogic, WebApi) existen proyectos de test e de implementación, por lo que los proyectos de test se les agregó la palabra Test al final para agruparlos con lo de implementación. Lo mismo para los que exponen interfaces hacia otras capas, por ejemplo BusinessLogicInterface.

## Funciones

La idea fue tener funciones cortas y simples de entender a lo largo de todos los proyectos.

Esto no se respetó en los proyectos de test debido al tiempo, se pensó hacer paquetes auxiliares para que generen entidades auxiliares según la necesidad de cada prueba, pero se dejó en To-Do de nuestros kanban para realizarlo como un refactor si sobraba el tiempo. Sin embargo, dentro de los proyectos de implementación se trató de aplicar de manera estricta el largo de las funciones para que contengan una sola responsabilidad.

Por ejemplo los métodos de las controladoras en la WebApi, no contienen más de 2 líneas de código, literalmente.

```

2+3 usages
public void Update(User testerUser, Bug updatedBug)
{
    if (testerUser.Projects.Find( match: p => p.Id == updatedBug.ProjectId) == null)
        throw new ProjectDoesntBelongToUserException();
    Bug bugFromDb = Context.Bugs.Include("Project").FirstOrDefault(u => u.Id == updatedBug.Id);
    if (bugFromDb != null)
    {
        bugFromDb.Name = updatedBug.Name;
        bugFromDb.Description = updatedBug.Description;
        bugFromDb.ProjectId = updatedBug.ProjectId;
        bugFromDb.Version = updatedBug.Version;
        bugFromDb.State = updatedBug.State;
        Context.Bugs.Update(bugFromDb);
    }
    else
        throw new InexistentBugException();
}

```

Esta es una de las funciones más largas que tenemos entre DataAccess, BusinessLogic y las controladoras de WebApi, y tampoco llega a 20 líneas de código.

Las funciones deben recibir 1 o 2 parámetros, en el caso de que se necesite un tercero, se ponía a discusión del equipo para encontrar una mejor solución o aceptar la excepción a la regla.

Al ser cortos los métodos, no ayudó a manejar una única responsabilidad en la mayoría de los casos, por lo que evitamos los switch a lo largo del obligatorio, que por definición realizan más de una cosa a la vez. Aunque en una clase no se nos ocurrió como mejorar el manejo de excepciones para un filtro ExceptionFilter, dentro del proyecto de WebApi, el cual queremos devolver un mensaje de error y un código según corresponda, y utilizamos varios if else, como un caso único.

## Comentarios

*"Don't comment bad code - rewrite it." - Brian W. Kernighan and P.J. Plaugher*

Nos tomamos esta frase en serio, casi nunca dejamos código comentado en las clases antes de hacer un pull request, se planteó como equipo poder hacer un refactor antes que dejar un comentario como un To-Do. En alguna prueba puntual, existe algún comentario breve, explicando porque se agrega esa línea de código, debido a que no tuvimos tiempo de refactorizar los test como hubiésemos querido.

## Formato

El formato del código se puede ver en cada clase, si esta contiene Properties, se colocan al principio con sus get y set. Luego las variables de clase, preferiblemente privadas para uso interno de la clase misma, siguiendo del constructor.

Entre método y método se deja un espacio en blanco para que el código “respire”.

Las variables utilizan camelCase para su definición, y por lo contrario los métodos se definen con su primera letra en mayúscula.

Los if, else que se pudieran evitar las llaves, debido a que el código era una sola línea, se evitó poner, para que verticalmente sea menos denso.

No se estableció una regla estricta para el largo de una línea de código horizontalmente, pero la idea era que no se salga de la pantalla.



Lo principal fue elegir este formato y respetarlo dentro de los proyectos para que no haya ambigüedad.

## Manejo de errores Excepciones

Para el manejo de errores, decidimos crear excepciones personalizadas que fueron utilizadas en distintas clases a lo largo del proyecto. Todas las excepciones de dominio heredan de una padre que es la que se catchea en un filtro de excepciones en la WebApi y tienen un mensaje de error para poder ser mostrado en el frontend (Postman en este caso), lo mismo ocurre en el caso de data access y exceptions de models. Se planteó de esta manera para que a la hora de capturarlas fuera más sencillo y seguir las recomendaciones de Clean Code.

## Teste Coverage

agust_DESKTOP-PFVBI7S 2021-10...	12	0.78%	1530	99.22%
businesslogic.dll	0	0.00%	99	100.00%
dataaccess.dll	0	0.00%	504	100.00%
domain.dll	0	0.00%	186	100.00%
filehandler.dll	0	0.00%	104	100.00%
filehandlerfactory.dll	0	0.00%	12	100.00%
utilities.dll	12	12.63%	83	87.37%
webapi.dll	0	0.00%	339	100.00%

Excluimos las migrations, los proyectos de test y los archivos autogenerados de WebApi para alcanzar este número muy cercano a 100% de las líneas del código testeadas. Utilities cuenta con 87% dado que se agregó una función para chequear filtros de búsqueda en el GetAll de bugs y no se probaron absolutamente todos los casos posibles. La función es la siguiente, no se realizaron más test por falta de tiempo, y decidimos hacer una excepción en este caso.

```
public bool MatchesCriteria(Bug bug)
{
    bool matches = (bug.Name == Name || Name == null) &&
        (bug.State == State || State == null) &&
        (bug.ProjectId == ProjectId || ProjectId == null) &&
        (bug.Id == Id || Id == null);
    return matches;
}
```