

Universidad ORT Uruguay

Obligatorio 2

Diseño de Aplicaciones 2

Agustina Disiot 221025

Ivan Monjardin 239850

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

Índice:

Descripción General del trabajo	4
Backend	4
Web API	5
Business Logic	6
Repository	7
Domain	8
DTO	9
Bug Parser	9
Funcionalidades implementadas	10
Mecanismo de Acceso a datos	11
Decisiones de diseño	12
Interpretación/Decisiones de letra	13
Independencia de Librerías	13
Resumen de las mejoras de diseño	15
User	15
Reflection	16
Endpoints comunes	16
DTO	17
Requerimientos de Extensibilidad	18
CustomBugImportation (verde)	18
Posicional Text Importer	20
Test de Reflection	20
Documentación de cómo implementar tu importador	20
Análisis de Métricas	21
Métricas cohesión relacional	21
Abstracción, inestabilidad y distancia	23
Principios	26
Angular	26
Usabilidad	27
Descripción del manejo de excepciones	27
Anexo	29
Anexo Heurísticas de Nielsen	29
Anexo API	29
Endpoints	30

Login:	30
Admin:	31
Bug:	31
Developer:	33
Project:	33
Tester:	34
Work:	35
Anexo Cobertura de Código	35
Factory de Parser en business logic	35
Repository design y migrations	36
Excepciones	36
WebApi	36
Comparers	36
Factory bug parser	37
Custom Bug Importer	37
Code Coverage Final	38
Anexo Respaldos Base de Datos	39
Anexo Pendientes obligatorio 1	39
Anexo Decisiones de Diseño obligatorio 1	39
Anexo Decisiones Repositorio	41
Anexo Carpeta de Importadores	41
Anexo Guia de instalación	41
Anexo diagrama de secuencia	42
Anexo Pantallas de Aplicación	42

Descripción General del trabajo

Se diseñó una aplicación para administrar los incidentes en los proyectos de software. Para esta segunda entrega se implementó el frontend en Angular y, que incluye una API para acceder a los recursos y persistencia de una base de datos.

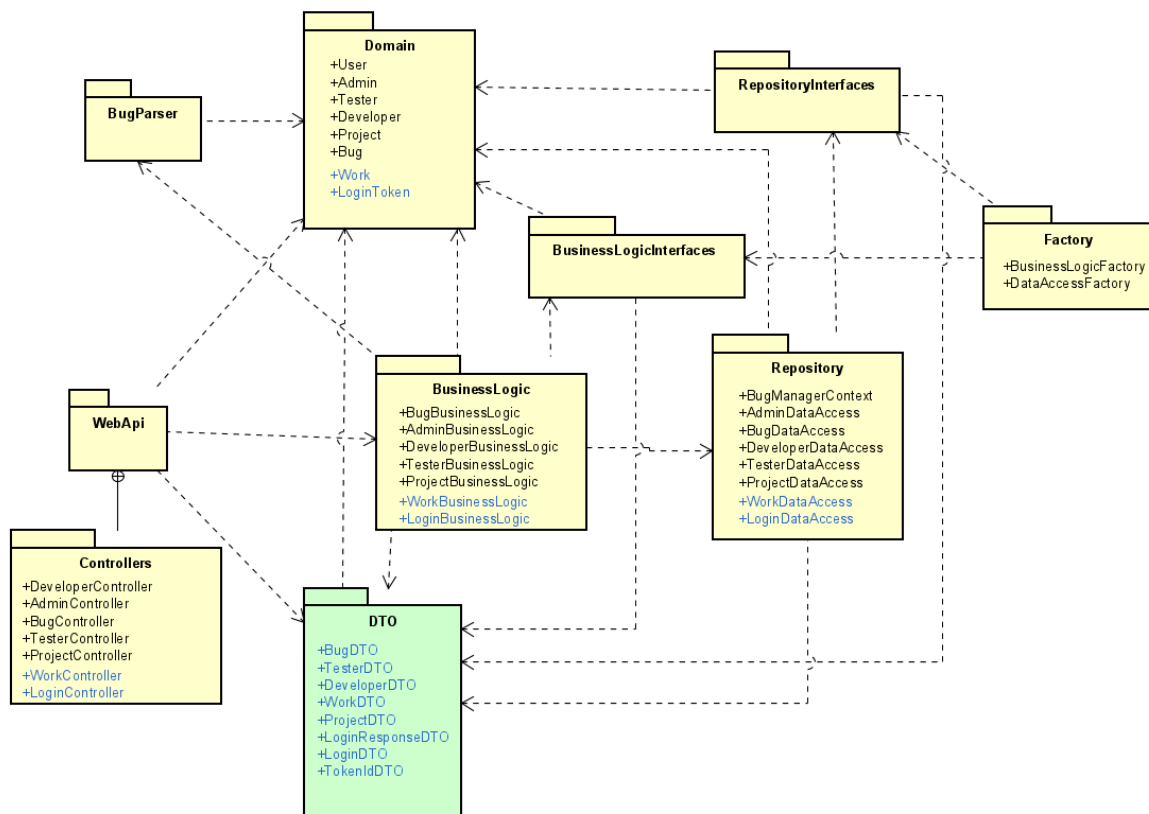
Nota: Este documento es una actualización del documento entregado en el obligatorio 1, se hará énfasis en los cambios, por ejemplo, utilizando otro color en algunos diagramas.

Además las imágenes de los diagramas en formato original de los diagramas se encuentran en la carpeta Documentation/Diagrams

Backend

Para implementar los requerimientos se decidió dividir el proyecto en diferentes capas, Web Api, Business Logic, Domain y DataAccess. Un requerimiento debe atravesar todas las capas para poder completarse por lo tanto en cada capa no están solo las responsabilidades de la capa en sí, sino además, las responsabilidades relacionadas a los requerimientos. Es por esto que decidimos separar cada capa en función responsabilidades más acotadas, apuntando que cada clase de cada capa tenga una única responsabilidad, pensando siempre en el Single Responsibility Principle.

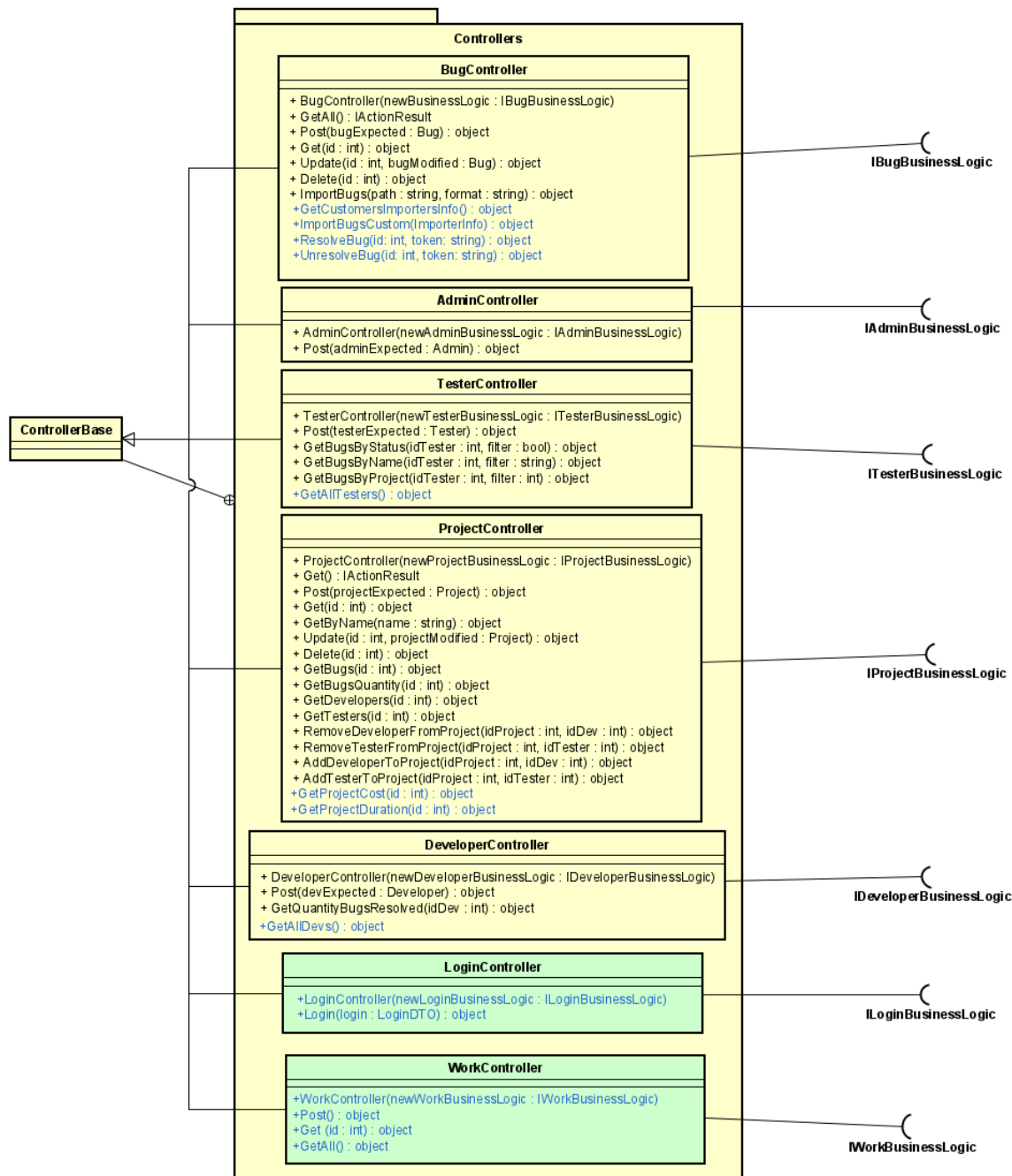
Tomando en cuenta la vista de implementación, un diagrama de las clases de cada capa/paquete es el siguiente:



Acá se puede ver la separación que hicimos principalmente tomando en cuenta la división que hicimos en el dominio. En cada capa hay una clase responsable de Bug, de Developer, de Project etc. Sobre la adición de DTO, hablaremos más adelante.

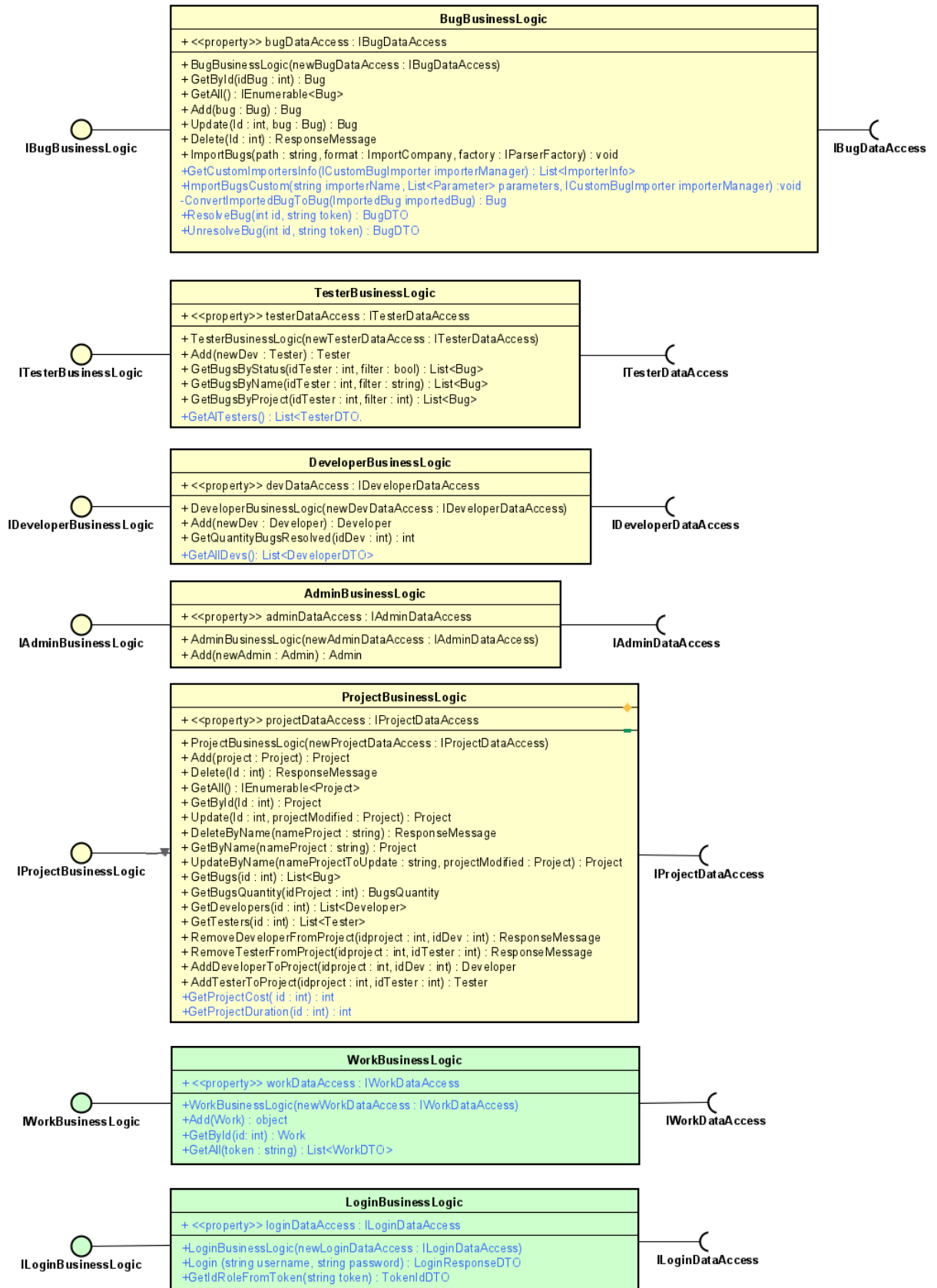
Pasando a la vista lógica vamos a poder ver las clases de cada proyecto

Web API



Es la clase encargada de los controladores que administran los endpoints mediante el cual se comunicará el front-end para mandar/recibir los recursos. En el Anexo API se detallan los cambios en los endpoints en relación a la entrega anterior, pero lo importante es saber que cada controlador tiene su propio principio de URI, ejemplo ProjectController administra todas las request a las URI que comienzo con /projects/, idem para todos los controllers. Esto nos permite hacer una separación de responsabilidad y no dejar todo en un solo controller.

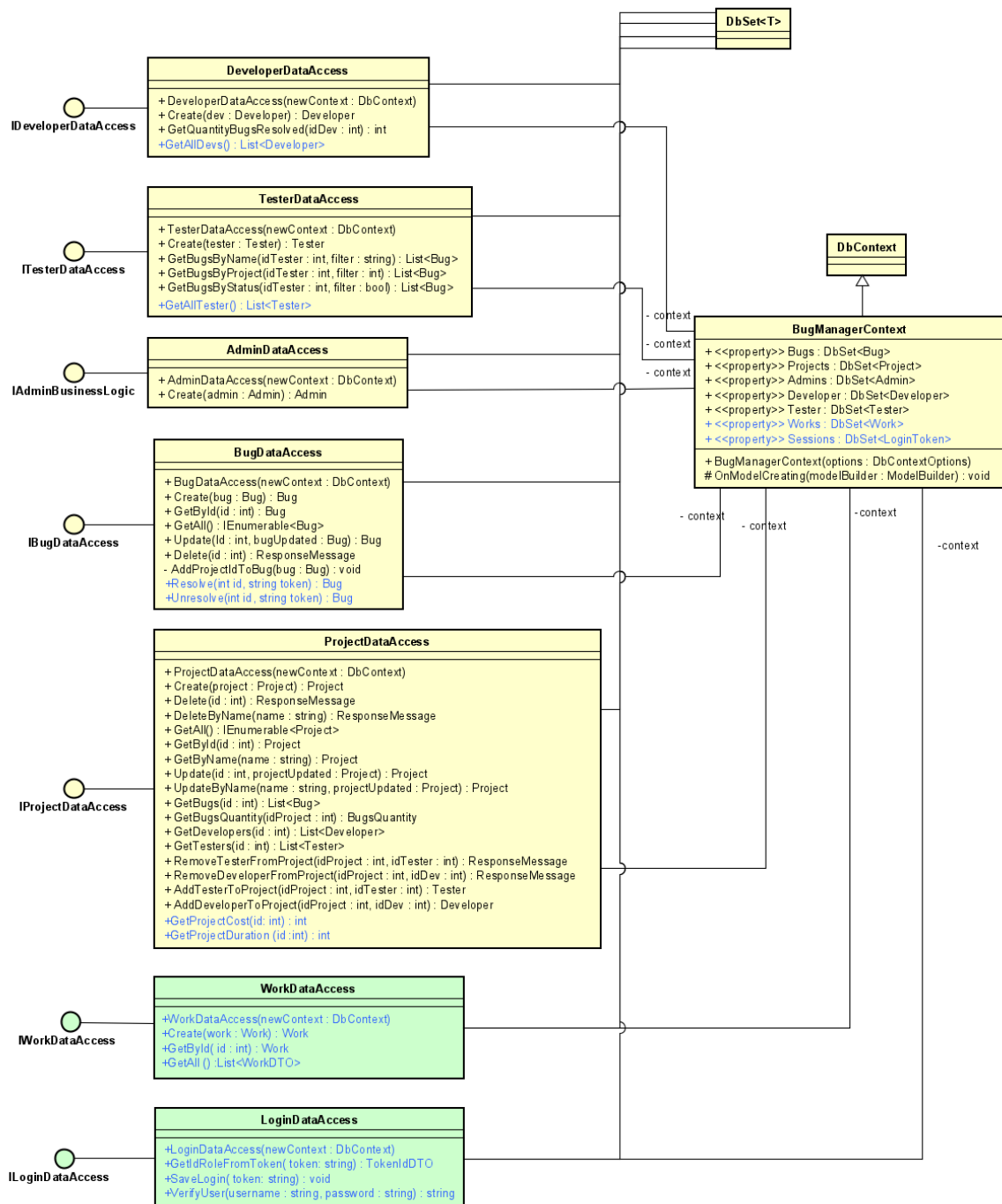
Business Logic



Es la capa encargada de recibir las peticiones de WebApi, aplicar las reglas de negocio necesarias y utilizar el Repository para guardar y recibir los datos.

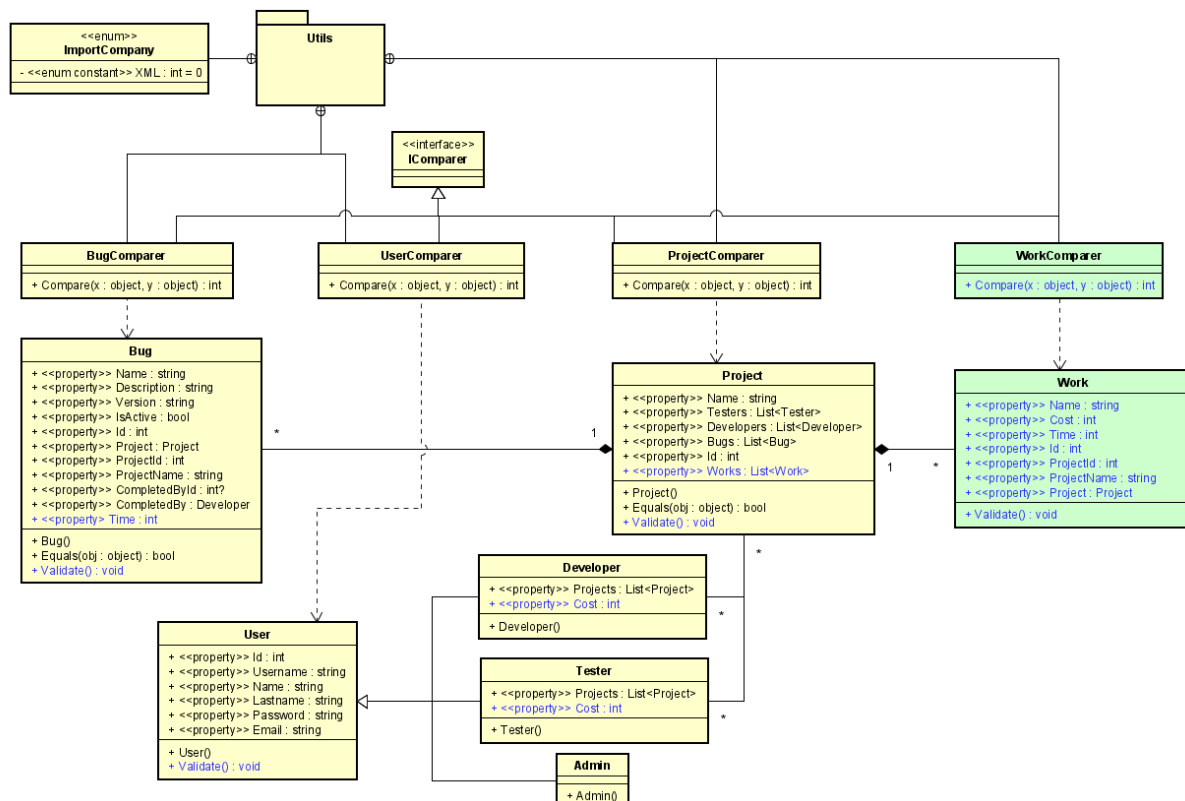
Algo que queremos resaltar es como podemos ver la inyección de dependencias en estos diagramas. A la izquierda pueden ver las interfaces que proveen/ implementa la businesslogic y a la derecha las interfaces que precisa y que van a ser inyectadas. Sucede algo similar con web api y data access, solo que en el primero no hay interfaz que implementan y en el segundo no se le inyecta otras clases.

Repository



Se encarga de guardar los datos del dominio en una base de datos persistente que fue creada utilizando Code First. Si bien Business Logic es el principal encargado de la lógica, operaciones como CRUD y otras que consideramos como responsabilidad de la base de datos se encuentran acá. Una de estas otras responsabilidades sería por ejemplo contar la cantidad de bugs resultados por desarrollador, ya que si bien tiene algo de lógica, se puede ver como una petición básica a la capa de acceso a datos.

Domain



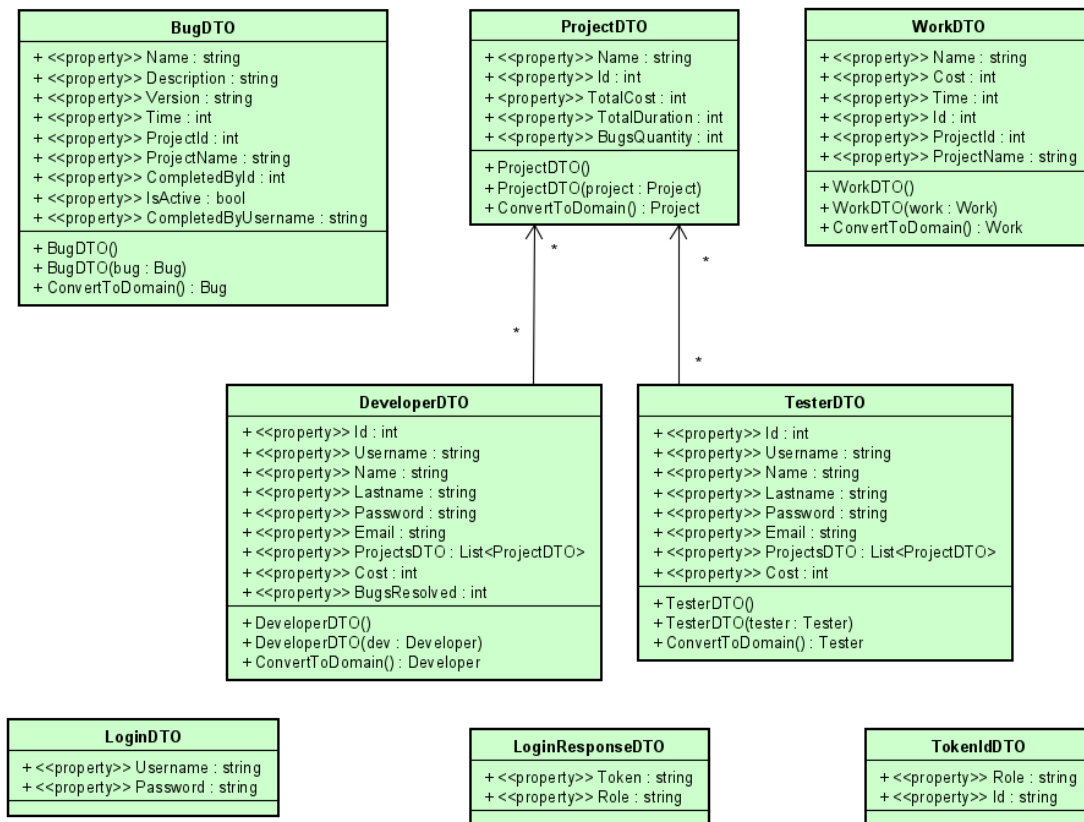
Esta capa no hace referencia a ninguna, es el paquete de las entidades básicas, las de dominio. Cada una incluye un constructor y propiedades relacionadas a esa clase. La lógica que si incluimos esta entrega en estas clases fue las validaciones (se puede ver en azul). Creemos que cada clase del dominio es el experto y debe saber cómo validarse. Esto no solo reparte mejor las responsabilidades, evitando enviar todo a business logic, sino además, evita “AnemicDomain”

Básicamente, cada clase es la encargada de guardar las propiedades relacionadas a ella misma. Esto incluye relaciones con otras entidades del dominio. Se incluyeron ambos lados de la relación (ej. Proyecto con bugs y Bug con Proyecto) ya que facilita la utilización de EntityFramework. Por esa misma razón se incluyeron propiedades como ProjectId que se usa en conjunto con ef core.

Además de los comparadores también se incluyeron las excepciones en el dominio. Originalmente estaban en businessLogic pero tomando en cuenta que todos hacen referencia al dominio y que una excepción se podría ver como una entidad del problema, se decidió

mover a dominio. Sigue habiendo excepciones en business logic pero excepciones relacionadas a la lógica, en dominio hay excepciones como ValidationException.

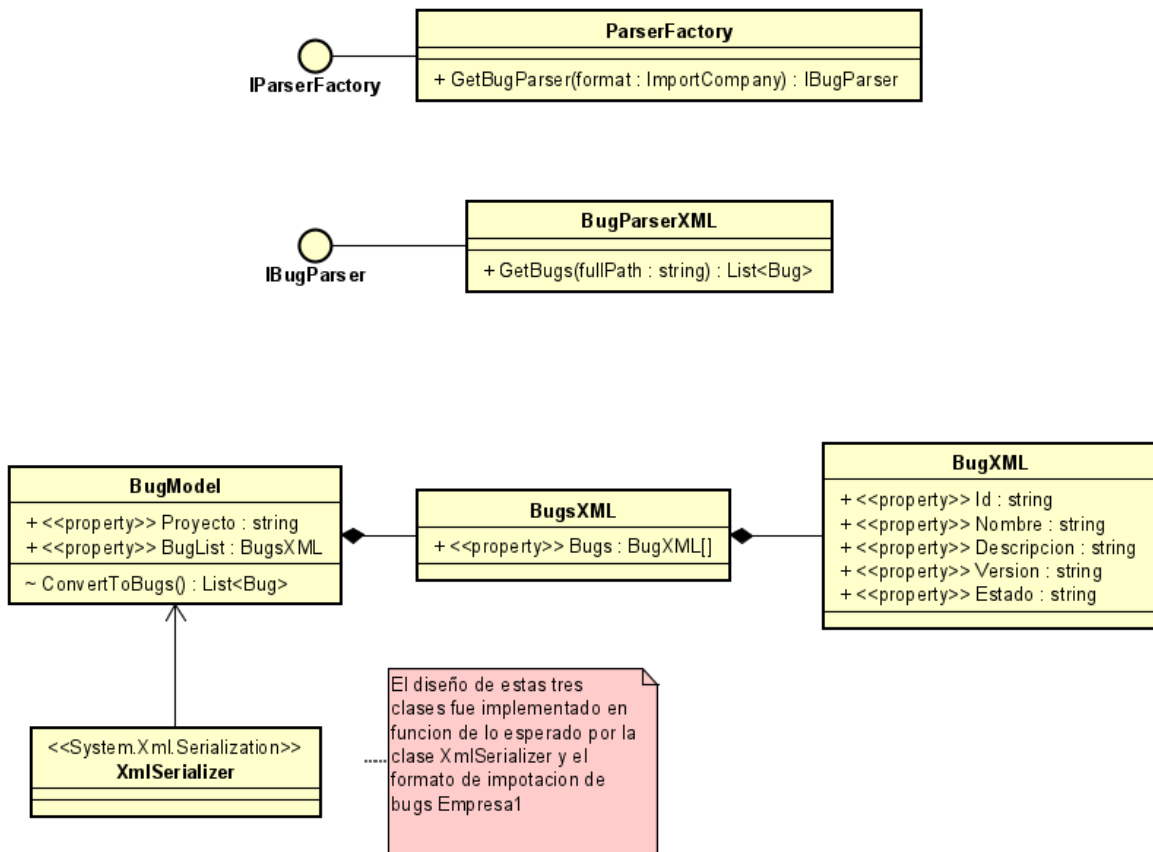
DTO



Para esta entrega se agregó un paquete de DTO para la transferencia de objetos entre business logic y WebApi. Más adelante vamos a hablar más en detalle sobre las decisiones que nos llevaron a crear este proyecto y sus ventajas.

Bug Parser

Para la importación de bugs se agregó la nueva funcionalidad pedida, sin embargo, la implementación original sigue válida y funcionando. La implementación original la llamaremos “clásica” mientras la otra será la extensible o “custom”



Para la implementación clásica de la importación de bugs, se decidió separar del resto de los proyectos la lógica de parsear los archivos. De esta manera es más extensible si en el futuro se quiere agregar una nueva empresa no es necesario cambiar, por ejemplo el business logic.

El business logic utiliza un factory donde le pasa el tipo de archivo de la empresa que quiere importar y la factory ya le devuelve la clase correcta a utilizar.

La idea de esta clase era de hacerla lo más extensible posible. Cumpliendo el principio de abierto al cambio y cerrado a la modificación. Es fácil crear una nueva clase, y no es necesario modificar el resto de los paquetes.

Se incluyó en el diagrama las clases utilizadas para parsear los archivos a xml para lo cual se utiliza la clase de sistema XmlSerializer.

Sobre la vista de procesos, se incluye también un diagrama de secuencia que conecta la front con el back - Anexo Diagrama de Secuencia

Funcionalidades implementadas

Para la primera entrega no tuvimos el tiempo de implementar transversalmente todas las funcionalidades, principalmente con un problema que tuvimos con la capa de acceso a datos el cual en esta entrega pudimos arreglar usando el nuevo proyecto de DTO.

En el anexo “pendientes obligatorio 1” la lista de las funcionalidades que no habíamos llegado a terminar 100%. Sin embargo, para esta entrega si se agregó:

- Roles de usuarios (admin, tester, dev) con su respectivos permisos, cuando un authorization filter - así separamos la responsabilidad de chequear permisos

- Importación mediante texto posicional - se implementó utilizando el sistema mejorado
- Cantidad de bugs resuelto por un desarrollador
- Conseguir la lista de proyectos de un desarrollador
- Endpoints específicos para resolver/desresolver un bug.
- Agregamos validación a dominio, probablemente se podría hacer más exhaustivas o aplicando un Notification pattern pero nos enfocamos más en la del front que son las que consideramos más importantes para esta entrega.

Además de las funcionalidades que no se habían terminado, se agregaron las nuevas de este obligatorio:

- Se agregaron las tareas - hicimos el código en inglés y para no ponerle “Task” y confundir con las task de c#, le pusimos “Work”. Después nos dimos cuenta que la traducción correcta era “Assignment” y esa fue la que utilizamos en el front ya que es la que ve el cliente y por lo tanto tiene prioridad sobre estos tipos de cambios.
- Se agregó costo a los tester y desarrolladores
- Se puede ver el costo de un proyecto - acá nos quedó un error a la hora de calcularlo, originalmente pensamos que solo el desarrollador podía resolver el bug, entonces las cuentas nos quedaron mal. De todos modos el precio si se actualiza dinámicamente y sin ser el cálculo el resto funciona bien.
- Extensibilidad de importadores - quedó implementada con los importadores de JSON y TXT
- Se implementaron todas las funcionalidades utilizando una aplicación Angular (SPA)

Errores o bugs que se encontraron:

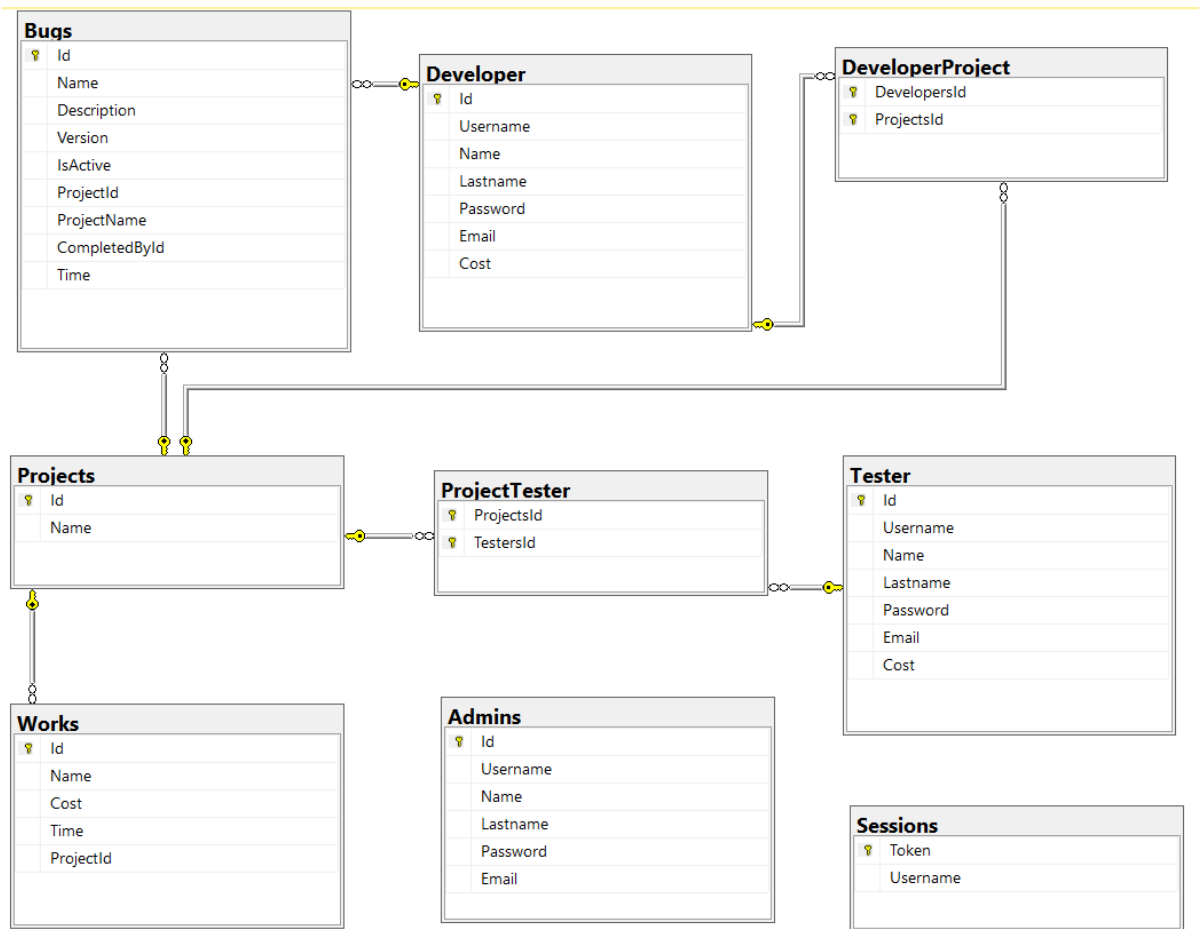
- Cuando abrir por primera vez la pantalla de importadores extensibles, el botón aparece habilitado hasta que elijas un importador o “Ninguno”
- No se agregó que el nombre de usuario sea único. De todos modos se utilizan ids para los usuarios.
- En el login si apretas enter se envían los datos y además se pone en modo visible la contraseña. Si le erraste en tus datos, queda tu contraseña visible.

Mecanismo de Acceso a datos

La capa de acceso a datos se utilizó para persistir los objetos del dominio. Es posible cerrar la aplicación y volverla a abrir y continuar teniendo acceso a los mismos datos.

Para la creación de la base de datos se utilizó la técnica de Code First y EntityFramework.

En función de nuestro dominio generamos la siguiente estructura de base de datos.



Se incluye un Anexo Respaldos Base de Datos, con la información de los respaldos de la base de datos.

Decisiones de diseño

En el primer obligatorio se toman varias decisiones de diseño las cuales siguen válidas. Para no hacer más largo el documento se agregaron las decisiones que siguen válidas en el Anexo "decisiones de diseño obligatorio 1". Además hay un "Anexo decisiones repositorio" sobre cómo utilizamos git, github y gitflow.

A estas decisiones de diseño se agregaron las de este obligatorio. En otras partes se van a hacer énfasis en las decisión de diseño correspondientes a la parte de Reflection y del front-end, pero acá hay algunas más genéricas que consideramos apropiado mencionar.:

- Se decidió que la duración del bug se indique cuando se crea, pudiendo ser modificada tanto por el administrador como por el tester
- El filtrado de bugs a nivel de front-end se decidió implementar a nivel del cliente. Es decir, se cargan los bugs y se filtran sin necesidad de hacer pedidos al endpoints cada vez que se quiere. Consideramos que esto tiene como resultado:
- Mejor usabilidad ya que reduce el tiempo de respuesta
- Poder decidir con más libertad cuando recargas los bugs, si después de cada carácter, si cuando deseleccionas el componente (nosotros usamos este), cuando apretá un

botón, etc. Esto es una decisión que se puede hacer en el front sin necesidad de pensar cuantas request vas a mandar o algún problema de rendimiento.

- Se creó además un componente en Angular que tenga como única responsabilidad el filtrado de bugs, favoreciendo Singles Responsibility, mantenibilidad y OCP. Estos últimos se cumplen ya que si el día de mañana tengo que cambiar el filtrado (ej. el cambiar el criterio) tener que ir a un lugar solo y el mostrar el bugs se maneja en componentes aparte sin necesidad de cambiarlo.
- La id(Número) del bug es un elemento visible a los usuario y no solo un elemento de la base de datos, ya que el tester debe ser capaz de filtrar por el mismo.

Interpretación/Decisiones de letra

- Cuando la letra pide que un tester pueda editar un bug, nosotros entendimos que eso incluye toda la información de un bug. Incluido quien lo resolvió.
- Solo el admin puede crear tareas, todos pueden ver sus tareas, el admin puede ver todas.
- El bug importado debe ser válido y debe existir el proyecto correspondiente, sino se tira la excepción
- Si un bug está resuelto, no es obligatorio que esté indicado el desarrollador. Puede quedar como no definido, en caso que el tester/admin no quiera especificarlo
- Solo el desarrollador puede figurar como la persona que resolvió el bug, aunque el admin y el tester si pueden cambiar su estado

Independencia de Librerías

A lo largo de las clases se ve nuestro uso de interfaces tiene dos principales ventajas:

1. Nos permite utilizar inyección de dependencias, la cual hablaremos más adelante
2. De la mano al anterior, las interfaces nos permiten desacoplar cada capa una de la otra. Ninguna capa depende de una implementación concreta de la otra. Además esto nos permitió realizar tests utilizando mocks (algo que también hablaremos más adelante)

El proyecto se podía haber realizado igual sin utilizar todas las distintas clases que mencionamos anteriormente, por ejemplo, teniendo todo en WebApi. Sin embargo eso iba a traer grandes problemas de extensibilidad, mantenimiento y romper muchos principios como single responsibility, bajo acoplamiento y alta cohesión. Todo eso nos dio la pauta de diseñar un sistema donde los componentes tengan dependencias lo más estables posible. Por ejemplo, dependiendo de interfaces, un “contrato” pensado para que dure aunque se modifique las clases concretas que lo implementan.

Inyección de dependencias

Se utilizó inyección de dependencias para poder evitar dependencias circulares y teniendo una forma fácil de modificar las clases concretas que utilizamos en nuestra aplicación. Esto quiere decir que:

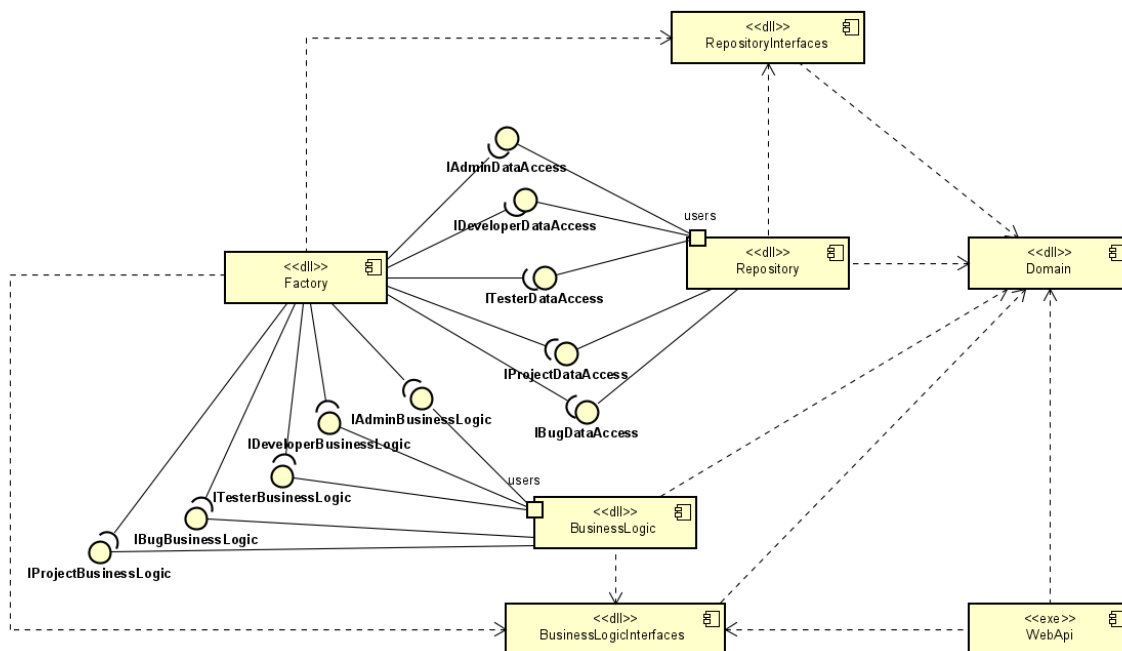
Las clases de business logic no crean una clase concreta de, por ejemplo, data access, para poder acceder a la base de datos. Business logic solo utiliza la interfaz de la clase del repository y sus funciones. Se utiliza inyección de dependencias para especificar a la hora de correr la webApi, que implementación concreta de dicha interfaz utilizar.

Cómo implementamos inyección de dependencias fue mediante una clase que usa el patrón factory en el proyecto web api (ya que ahí es donde se corre el back end). Está factory conecta cada interfaz con su clase concreta que deseamos utilizar.

```
public void AddCustomServices()
{
    serviceCollection.AddScoped<IBugBusinessLogic, BugBusinessLogic>();
    serviceCollection.AddScoped<IProjectBusinessLogic, ProjectBusinessLogic>();
    serviceCollection.AddScoped<IAdminBusinessLogic, AdminBusinessLogic>();
    serviceCollection.AddScoped<IDeveloperBusinessLogic, DeveloperBusinessLogic>();
    serviceCollection.AddScoped<ITesterBusinessLogic, TesterBusinessLogic>();
}
```

Código de la factory de business logic

En el siguiente diagrama de componentes podemos ver como funciona la factory y la inyección de dependencias. Vemos como hay muchas clases que no se utilizan directamente pero que pasan por la Factory para recibir las implementaciones concretas.



El diagrama muestra cómo se pueden reemplazar las dependencias por interfaces y usando la Factory como pasamanos entre ambos lados de la relación. Además de haber utilizado inyección de dependencias en el backend con C#, también se utilizaron a nivel de front-end con Angular. Por ejemplo, los servicios que utilizan la web api son inyectados en los distintos componentes que lo precian.

Resumen de las mejoras de diseño

Desde la entrega anterior se realizaron muchas mejoras, aca algunas que consideramos más relevantes:

User

Uno de los principales problemas que notamos fue nuestro manejo de usuarios. Si bien, al principio, no existía tanto código repetido, nos dimos cuenta que tener siempre clases separadas de admin, tester y dev no iba a ayudar a la mantenibilidad del sistema.

A la hora de realizar refactors y mejorar el código, vimos dos opciones:

Tener tres clases separadas, pero aplicando template method y así facilitando polimorfismo.

Unir todo en una única clase para los tres users.

Nosotros optamos por la primera, después vamos a comentar porque la segunda también podría haber sido, incluso una mejor opción.

Partiendo de tres clases separadas a lo largo de las distintas clases, ej, business logic, dataAccess, e inclusive los tests. Las funcionalidades se empezaron a repetir, al principio era solo crear usuario pero cuando llegamos a la autorización ya se empezó a repetir más código.

Por lo tanto:

Se agregaron clases que juntaban las funcionalidades comunes a los usuarios. Esto se realizó inclusive a nivel de los tests, ya que un cambio en los requerimientos iba a podría provocar un cambio en los tests y por hacer muy costó el cambio.

En algunos casos, por ejemplo en los test de dataAccess se utilizó una clase abstracta con gran parte del código y se utilizaron clases que implementan esa clase completando las funcionalidades que faltan a la clase padre. Esto se realizó siguiendo el patrón de template method.

En otros, se definió una interfaz, por ejemplo en business logic, que todas las clases correspondientes de cada rol de usuario tengan que implementar. Así al agregar nuevas funcionalidades a los usuarios nos aseguramos que todos la implementen, básicamente creando un contrato común.

La segunda opción, que si bien no llegamos a implementar, creemos que sería la siguiente mejora que impactaría más en el diseño.

Al momento de decidir cuál de las dos opciones, no se optó por la segunda porque consideramos que haría más difícil el mantenimiento si en el futuro se agregaron más funcionalidades/propiedades diferentes a cada tipo de usuario. Al final, esto no sucedió tanto como esperamos, pero en su momento no lo sabíamos.

De todos modos, si bien se podría seguir mejorando, este cambio, aportó mucho a la mantenibilidad y a el reuso del código, evitando errores y mejorando la exhaustividad de los tests.

Reflection

Otra de las grandes mejoras a nivel de diseño fue la utilización de Reflection para la importación personalizada de bugs. Más adelante vamos a detallar bien la solución, pero ahora vamos a hablar de las mejoras que tuvo a nivel de diseño.

Nosotros tuvimos el siguiente problema que queríamos resolver:

Se quiere agregar más importadores

Tiene que ser de una manera sencilla

Se tiene que poder agregar en tiempo de ejecución

Este último ya ocurre gracias a la utilización de reflection, los otros 2 también los cumplimos:

El principal patrón de diseño con el que nos basamos fue Open Close Principal. Siempre buscando una solución abierta a la extensión (agregar más tipo de importadores) y cerradas al cambio (no hay ninguna necesidad de cambiar el resto de las soluciones)

La utilización de soluciones diferentes y la separación de responsabilidades permiten realizar cambios de manera sencilla, por ejemplo, agregar otro importado, o agregar nuevos tipos de parámetros.

La utilización de una interfaz independiente de cualquier solución (por lo tanto estable), provoca que el usuario que hace la interfaz no tenga que conocer cómo funciona mi programa o la estructura de código, solo tiene que entender la solución que se le manda que fue creada con la única responsabilidad de crear nuevos importadores. Por ejemplo, no es necesario que el que cree el importador conozca mi dominio, ya que se utilizan DTOs específicos para la importación.

Tampoco se usa el mismo DTO que se usa, por ejemplo, en la web api. Así separamos mucho más las responsabilidades y un cambio en como se muestra un bug en la interfaz (que propiedades es mandan) no afecta la funcionalidades de cómo se importa un bug. Todo esto para poder conseguir los beneficios de OCP.

También, se puede notar el principio de Variaciones protegidas, gracias a la utilización de una interacción y la separación en distintas soluciones y lógicas distintas. Los distintos importadores solo se van a ver afectados si cambia la interfaz, la cual no debería cambiar mucho ya que se espera que sea estable.

El otro principio que también se cumple en menor medida es el de indirección. Antes nuestra solución estaba totalmente vinculada a los importadores y viceversa. Ahora estando la interfaz y la solución de custom importers en el medio, ayuda a poder detener que se propague cualquier cambio de cualquier de los lados.

Endpoints comunes

Otra mejora fue la utilización de endpoints “comunes” para los distintos roles. Por ejemplo el GET Bugs devuelve distintos resultados dependiendo de si sos admin, tester o developer. La lógica para admin es que se devuelvan todos mientras que en el tester y dev. solo los de sus proyectos.

La ventaja de tener el mismo endpoint y que el tenga la single responsibility de conseguir los bugs, es que simplifica el código en el frontend/cliente de la api. Tener una api que permita un código que cumpla “keep it simple” es fundamental para evitar tener mucha lógica de manejo de api en el front. y manteniendo las responsabilidades de cada capa (cumpliendo SRP).

Endpoints diferentes o endpoints que devuelvan muchos datos aumentan la lógica en front, sacándola de donde tendría que estar (BusinessLogic) y genera peor rendimiento, empeorando la usabilidad del mismo.

DTO

Por último y la mejora más importante, fue la utilización de DTO entre business logic y Web API. Que además después eran utilizados para transformarlos a json y mandarlos al front.

En vez de utilizar directamente las clases del dominio para mandar/ recibir en los endpoints, se utilizan DTOs.

Ventajas:

- Permite decidir qué datos mandar al front-end y cuáles no. No hay que darle a conocer todo el - dominio al cliente para que puede utilizar nuestra solución
- Protege ante variaciones/modificaciones del dominio. Si cambia el dominio, yo puedo ajustar mis DTOs para asegurarme que el endpoint funcione igual.
- De igual manera, si se necesitan endpoints diferentes, no tengo que cambiar mi dominio creó nuevos DTOs o modifíco los que tengo. Esto crea una solución que se puede adaptar el principio de segregación de interfaces, si el día de mañana tengo un nuevo cliente, puede cambiar los endpoints y utilizar DTOs diferentes en caso que fuera necesario. No hay necesidad de cambiar el dominio.
- Se cumple con SRP, DTO es el responsable de la información que se expone al cliente.
- El dominio ya en sí tiene responsabilidades importantes y además tiene que tener cuidado con cambia mucho ya que muchas clases dependen de él. Secarle esta responsabilidad al dominio era algo que teníamos pensado desde la entrega anterior.
- Por último, utilizar DTO nos ayudó a resolver problemas con el data access y referencias circulares al transformar a JSON. Los DTO se diseñaron ya pensando en evitar referencias circulares. Nos permitió tener un código más simple/mantenible a nivel de data access. Antes teníamos que hacer operaciones para sacar las referencias, ahora esa responsabilidad cae sobre web api/ dto (cómo debería ser) .

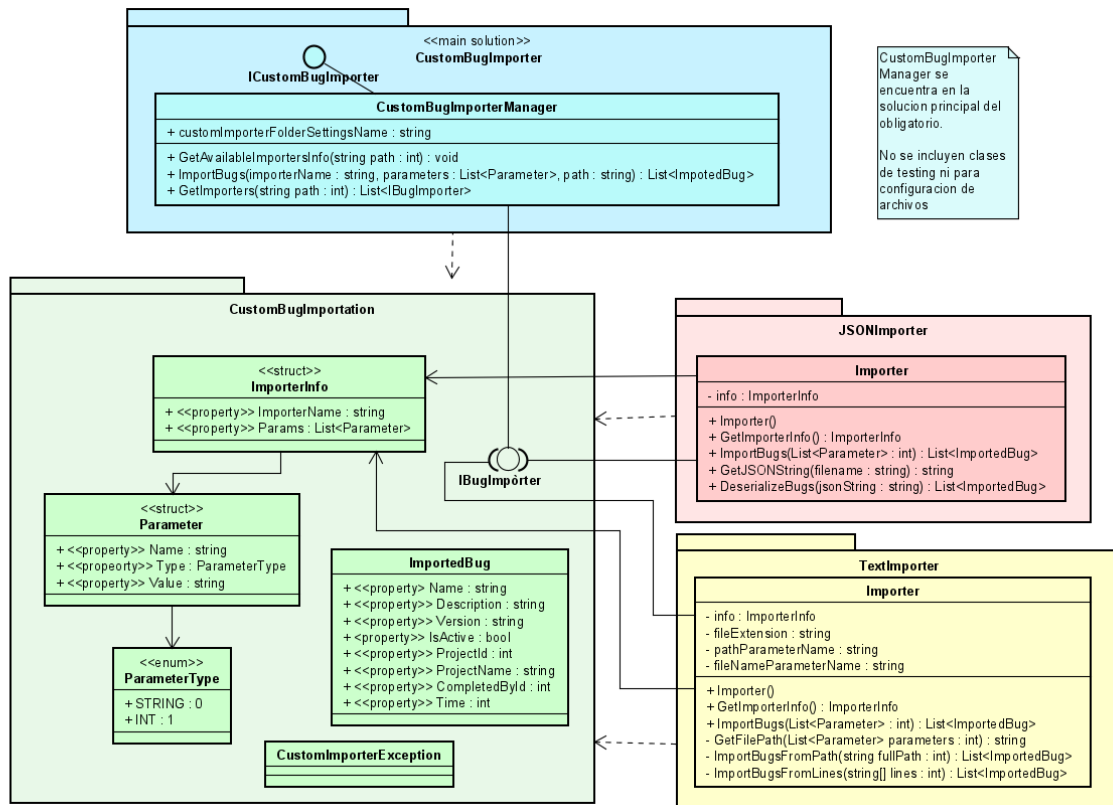
Requerimientos de Extensibilidad

Para el requerimiento de extensibilidad de importadores se buscó una solución que cumpla el principio abierto cerrado:

Se puede agregar más importadores (abierto) sin necesidad de cambiar el código existente (cerrado)

Para esto se usó reflection, no solo nos permitió cumplir con lo anterior sino además agregar importadores en tiempo de ejecución lo cual es otro requerimiento que piden.

Se llegó a la siguiente solución:



CustomBugImportation (verde)

Se crea una solución a parte que implemente lo necesario para que las distintas empresas puedan diseñar su importador.

Se crea una interfaz la cual es usada por todos los importadores y por nuestra solución original. Por lo tanto no depende de ningún otro paquete. Eso nos permite poder pasarle el .dll a la empresa que vaya a crear su importador sin la necesidad de pasarle, por ejemplo, el bug del dominio (y por lo tanto todo Dominio). Esto último también nos mejora las métricas de esta solución haciéndola más estable.

Además de la interfaz este paquete también cuenta con:

- `IBugImporter`: La interfaz con los métodos de importar bug y conseguir la información del importador, la cual deben implementar los importadores.
- La solución principal no usa los importadores directamente, usa la interfaz

- ImportedBug: un DTO que reemplaza bug y poder pasarlo a los importadores sin tener que pasarle dominio.
- ImporterInfo: El nombre y parámetros del importador. Estos son mostrados al usuario para que elegir y utilizar el importador
- Parameter y Parameter Type: Utilizamos para definir los parámetros para utilizar la importación.
- Equals(obj): Implementado en las distintas clases por si los importadores quieren usarlo para testing. No se puede usar un IComparable porque algunos son structs.

Se decidió no dejar fijos los parámetros en la interfaz IBugImporter porque esto limitaría la extensibilidad de los importadores.

Se definieron dos tipos de parámetros iniciales(usando un enum para permitir que fácilmente se pueden agregar más). Se definieron String e Integer como parámetros válidos.

La ventaja de hacer parámetros definidos y no, por ejemplo, usar una lista de object o string, es que nos permite que la front se modifique en función del tipo de parámetro. Por ejemplo:

Folder path

Enter folder path *

File Name

Enter file name (Numbers only) *

La validación y la entrada de parámetros puede depender de los parámetros, aumentando la extensibilidad y más que nada, la usabilidad del sistema.

Por último se definió una excepción para que cada importador la puede utilizar a su discreción, sea por error de los parámetros, del funcionamiento etc. Además que se permite que utilicen sus propios mensajes de error. Esto lo hace aún más extensible ya que no tiene que depender de alguna excepción de la solución principal o utilizar excepciones del sistema (lo cual lo haría menos mantenible y consistente).




Toda las soluciones vinculadas a este requerimiento se encuentran en la carpeta Backend/Reflection

JSONImporter

Se implementó el JSON importer en una solución a parte, dando el .dll del CustomImporter para que pueda implementar la interfaz.

Se definió la estructura del JSON en Anexo: Estructura de JSON

Tanto el JSON como el TXT importer no hacen referencia directa al proyecto principal y se encuentran en soluciones apartes. Se les pasó el DLL de CustomBugImportation para que puedan implementar la interfaz.

 JSONImporter	13/11/21 12:08 p. m.
 TextImporter	13/11/21 1:12 p. m.
 CustomBugImportation.dll	10/11/21 7:33 p. m.

Posicional Text Importer

En la entrega anterior no llegamos a implementar este importador. Por lo tanto para esta entrega se tomó la decisión de ya implementarlo en el nuevo formato extensible.

Además se aprovechó la oportunidad para poner varios parámetros a un importador y en vez de usar todos string, al nombre del archivo lo fijamos como integer. Principalmente para poder mostrar la nueva funcionalidad de importadores con algunos casos más interesantes.

De todos modos, el xml si quedo como importador “clásico” y en el front conviven ambos sistema como se pedía en la letra.

Notas:

- El “Folder path” para el Text Importer tiene que terminar con “\” o la barra correspondiente, no se agrega dinámicamente en el código porque no teníamos claro cómo difiere en los distintos sistemas operativos o incluso dentro de windows.
- Se agregó una duración al final del archivo, se tuvo que cambiar el activo/resuelto por un 1/0 para no cambiar el largo.

Test de Reflection

Para los tests de la solución principal de conseguir los importadores y utilizarlo, se utilizaron importadores dummies, con bugs hardcodeados. Originalmente se encontraban ambos proyectos en una solución y después por refactor se separaron. Esto puede afectar el code coverage, ya que los tests y el código no se encuentran todos en la solución que testea.

Documentación de cómo implementar tu importador

- Conseguir el .dll de CustomBugImportation. Por defecto se encuentra en Backend/Reflection
- Crear una clase que implemente la interfaz IBugImporter
 - En GetImporterInfo devolver un ImporterInfo con el nombre de tu importador y la lista de parámetros
 - Los parámetros pueden ser de tipo string o integer. El cambio es solo a nivel de interfaz gráfica.
 - Los valores de los parámetros al invocar el importador son de tipo string.
 - Es responsabilidad del importador castearlo si necesario (Ejemplo: conseguir un int) y tirar CustomImporterException si ocurren fallas
 - En ImportBugs recibir los parámetros y devolver la lista de bugs importados. Los bugs se van agregando de a uno al sistema, si se quiere agregar todos o ningun en caso de falla, se debe realizar la validación a nivel del importador.

- Por más información pueden consultar los tipos de los structs incluidos en el dll
- Compilar y agregar el .dll del importador a la carpeta de importadores. La misma está definida en el app.setting en el proyecto de WebApi. La clave del valor de la carpeta es CustomImporterFolder
- Solo se soporta un importador por .dll

Para configurar el importador de bugs e indicar la carpeta donde se encuentran ir al “Anexo carpeta de importadores”

Análisis de Métricas

Se utilizó ndepend para el cálculo de métricas y se realizó un análisis de las mismas. Enfocados en las métricas de inestabilidad, abstracción y cohesión.

También incluimos un vídeo de los resultados de ndepend:

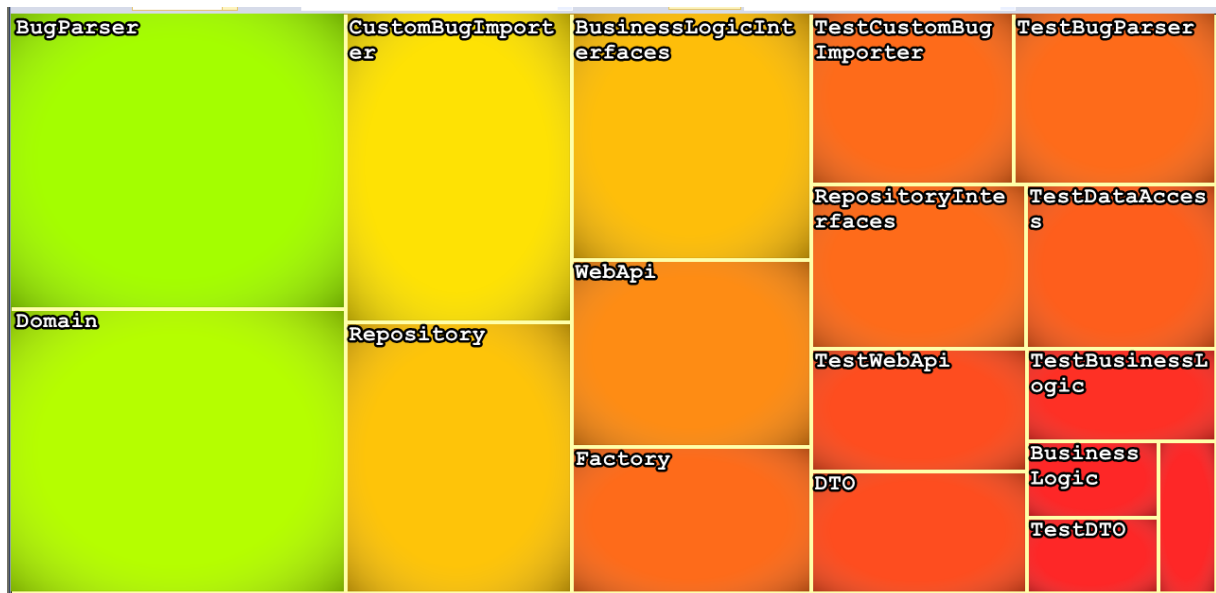
<https://youtu.be/zxyf4OwSZGg>

Aclaraciones: El estudio se realizó antes de que nos dieramos cuenta que quedaron incluida las clases de test y, que se consideran las clases del sistema como relaciones.

Métricas cohesión relacional

Los valores de la cohesión nos dieron:

Project	Cohesión Relacional
Domain	1,37
DTO	0,38
BusinessLogicInterfaces	0,85
RepositoryInterfaces	0,5
CustomBugImporter	1
BugParser	1,43
BusinessLogic	0,14
Repository	0,87
WebAPI	0,64



En este diagrama, tanto el color como el tamaño representan la cohesión.

Algunas cosas que queremos destacar:

En ningún proyecto nos dio muy buena la cohesión, en todos fue por debajo de 1.5. Esto quiere decir que las clases no están muy relacionadas entre ellas o que la cantidad de clases es muy grande. Concluimos que se debe a que las clases de un mismo paquete no se relacionan mucho.

Esto probablemente sea consecuencia de la arquitectura en capas que utilizamos. Hacer paquetes divididos según WebApi, Logic, DataAccess. etc. Si bien dividimos según responsabilidades, esto empeoró nuestras métricas. Tanto business logic como web api apenas tiene relaciones entre las clases, al igual que el data access. Hubiera sido diferente si hubiéramos separado por, por ejemplo, entidad del dominio, teniendo todas las clases de bug en un mismo proyecto, ahí sí se relacionarían más las clases y aumentarían la cohesión.

Tener clases con mayor cohesión nos hubiera ayudado en este segundo obligatorio. Por ejemplo cuando se pidió agregar costo a dev/tester o agregar las tareas. Si hubiéramos seguido más CCP, una opción era tener un paquete de dev, tester y uno nuevo de tarea. Se podrían implementar los nuevos requerimientos sin necesidad de afectar otras clases. Favoreciendo también OCP (podía agregar y extender nuevas funcionalidades sin cambiar muchas otras clases de otros paquetes).

Con nuestro sistema actual, los requerimientos de costo y tarea afectaron a todos los paquetes en todas las capas.

Si bien, se podría decir que nuestro sistema actual apunta más al reuso, ya que cada capa/paquete se podría usar bastante independientemente, cumpliendo más CRP.

Nosotros creemos que hubiera sido mejor apuntar a la mantenibilidad en vez del reuso ya que el software siempre se encontró evolucionando e incluso sabíamos que íbamos a tener un segundo obligatorio que iba a traer más requerimientos. Esto también se vio afectado ya que tuvimos que arreglar elementos de la primera entrega y por lo tanto en ambos obligatorios el backend tuvo muchos cambios, no llegamos a suficientemente estable como para que valga la pena aplicar más CRP como dice Robert Martin.

Un paquete que todavía no mencioné fue Dominio. Este fue uno de los paquetes con mayor cohesión, lo cual tiene sentido si pensamos que su responsabilidad es representar las entidades del problema. Sin embargo no tuvimos tanta cohesión como esperamos por la gran cantidad de clases en relación a las relaciones entre ellas.

Por ejemplo, nosotros tenemos 3 clases para los usuarios en vez de una, y Admin además no se relaciona con nadie. También, al aplicar experto y poner las excepciones de dominio dentro del paquete dominio, eso aumentó las clases, no aumentó las relación y por lo tanto disminuye la cohesión. Sucede lo mismo con los Comparer que tambien se dejaron en dominio por ser el experto. Esto último probablemente se tenga que cambiar, principalmente porque fueron utilizado en su mayoría para test y no estaría bien poner algo del test en una clase como dominio ya que muchas otras clases la utilizan y un cambio en dominio genere la recopilación de todo.

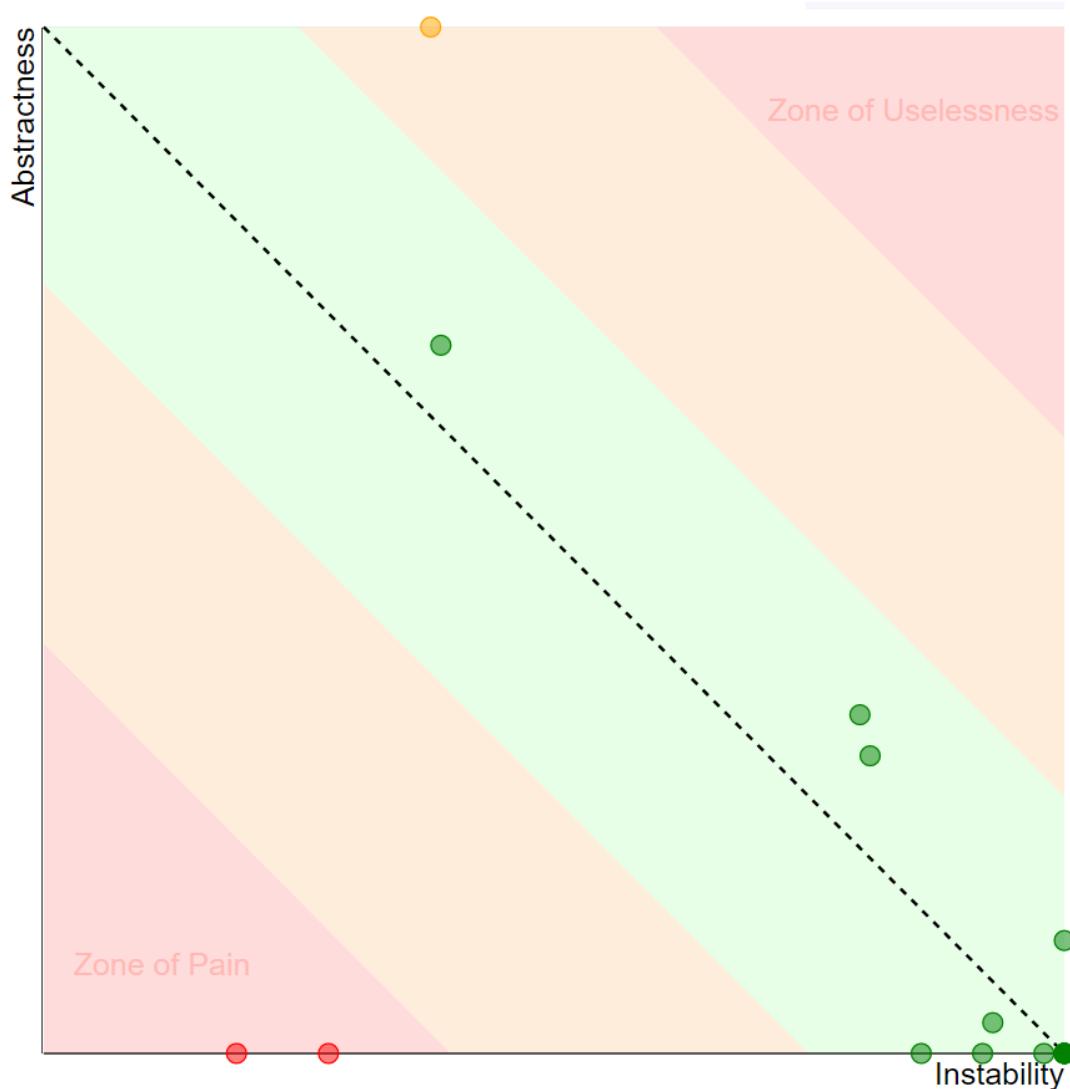
La otra clase que nos dio mejor la cohesión fue el BugParser (la importación original de bugs) principalmente porque se diseñó desde el principio para poner todas las clases relacionadas a la misma en un mismo paquete. Esto facilita en un futuro poder modificar o reemplazar ese paquete por el más nuevo sin tener que ir sacando muchas lógica de, por ejemplo, business logic.

También tenemos el paquete DTO que tiene las mismas entidades que Dominio pero menos relaciones (ya que se quiso evitar relaciones circulares y buscando dejar los dto simples), por lo tanto la cohesión es más baja.

Abstracción, inestabilidad y distancia

Project	Abstractness	Instability
Domain	0	0,19
DTO	0	0,28
BusinessLogicInterfaces	0,69	0,39
RepositoryInterfaces	1	0,38
CustomBugImporter	0,33	0,8
BugParser	0,29	0,81
BusinessLogic	0	0,86
Repository	0,03	0,93
WebAPI	0	0,92
Factory	0	0,98

TestDataAccess	0,11	1
TestDTO	0	1
TestWebAPI	0	1
TestDomain	0	1
TestCustomBugImporter	0	1
TestBusinessLogic	0	1
TestBugParser	0	1



En general estos valores no dieron mejores que los anteriores, por ejemplo la distancia para la mayoría de los paquetes se encuentra en la zona verde. Sin embargo, hay algunos paquetes que queremos resaltar:

Primero de todo dominio, el cual se encuentra en la zona de dolor. Esto se debe a que es muy estable ya que muchos dependen de él y él depende de muy poco (de hecho, probablemente depende de ninguno si sacamos las clases del sistema). Además, el dominio no tiene ninguna interfaz o clase abstracta por lo tanto es puramente concreto y su abstracción es nula.

Generalmente lo que se haría con un paquete de esta zona es intentar hacerlo más abstracto y que las otras clases dependen de la abstracción y no de la clase concreta. Esto lo movería más cerca de la zona verde de inestabilidad. Esto nos ayudaría a cumplir más con el SDP, ya que hoy hay dependencias más estables que dominio que van a dominio, cuando debería ser que un paquete dependa solo de los paquetes más estables que el. De todos modos, este un resultado “malo” de la métricas pero que aceptamos porque consideramos que dominio tampoco tiene que cambiar tanto una vez que se haga y que si llega a cambiar, al representar las entidades del problema de la vida real, tiene sentido que el resto de proyectos cambien con el.

Sucede algo similar con el paquete de DTO, es un paquete estable porque muchos dependen de él pero es poco abstracto porque solo hay clases concretas. Si siguiéramos SDP tendríamos que modificar este paquete. De todos modos, al igual que dominio, sería complicar mucho el código (no siguiendo keep it simple) realizar una interfaz y se perdería la simpleza de usar DTO para facilitar el traslado de objetos de una capa a otra.

Por otro lado, tenemos a RepositoryInterfaces que es totalmente abstracto pero no tantos paquetes dependen de él. Business logic es el único que usa repositoryInterfaces. Si bien las métricas nos dirían que una clase abstracta capaz no es tan necesaria, se tomó esa decisión para aprovechar la inyección de dependencias y el mocking de los tests.

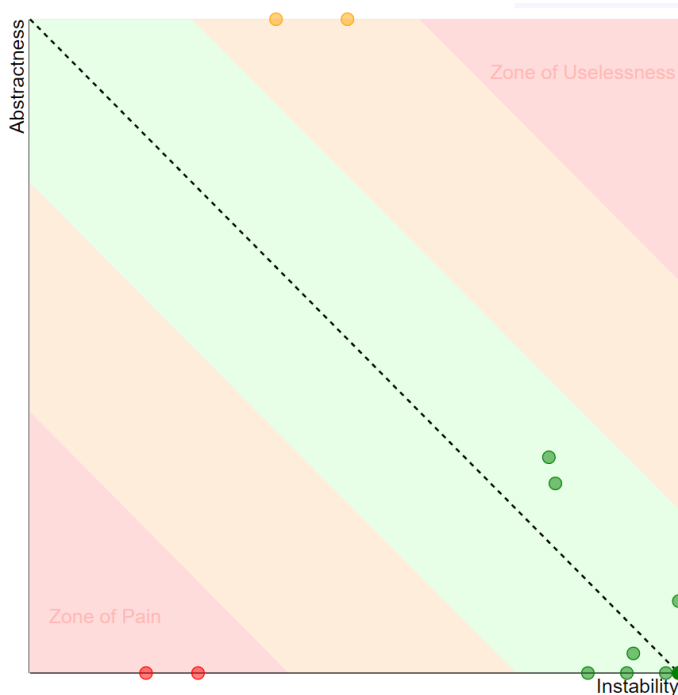
Cuando vimos esta última métrica nos cuestionamos por qué BusinessLogicInterfaces no es totalmente abstracto también. La razón es porque contenía en una carpeta DTO, objetos que se usaban para comunicarse con la WebApi y que eran mandados al cliente. Tener estas clases concretas en una clase tan abstracta nos mejoraba las métricas. Sin embargo, nos dimos cuenta que esa no era la mejor ubicación para esas clases y que ahora tenían que ir en el nuevo proyecto de DTO. Por lo tanto se decidieron sacar de BusinessLogicInterfaces aunque eso empeorara las métricas, preferimos mejorar el diseño, la mantenibilidad y seguir principios como SRP y Experto antes que tener mejor métricas.

Después de la modificación:

Movimos los Utils de BusinessLogicInterfaces a DTO

Ahora DTO es más estable y BusinessLogicInterfaces es más abstracto. A continuación los resultados finales. Como esperábamos BusinessLogicInterfaces tiene una peor D

Project	Abstractness	Instability	Cohesion
DTO	0	0,26	0,25
BusinessLogicInterfaces	1	0,49	0,5



Principios

Ya hablamos de la mayoría de los principios de paquetes pero queríamos mencionar:

- Principio de Equivalencia Reuso-Liberación: Si bien no se cumplió en su totalidad (ejemplo no en WebApi donde tenemos filtros), si se tomo en cuenta en la mayoría de los proyectos/assemblies, donde solo se colo un proyecto por assembly y viceversa. Por ejemplo si se cumple en DTO, BusinessLogicInterfaces, CustomBugImporter etc..
- Principio de dependencias acíclicas, se evitaron las dependencias acíclicas, utilizando inyección de dependencias, factories tanto en el back como en el front.

En resumen: Creo que reestructurar los proyectos puede generar grandes mejoras en nuestra métricas. Pero nos decidimos por un diseño en capas y en darle más importancia al diseño que solo a las métricas. De todos modos, nos sirvieron mucho para saber donde tener más cuidado y saber si teníamos que realizar cambios como explicamos antes.

Angular

Para el front end se utilizó una aplicación SPA en Angular. A continuación una breve descripción de las principales decisiones de diseño:

- Se realizó utilizando Angular Material, no una template, así evitar tener código de terceros en el mismo nivel que el nuestro pero que no entendamos
- Organización y contenido de la carpeta FrontEnd/Bugmanager/src/app:
 - /views: se definieron 5 vistas principales: un login, un page not found y una por cada rol
 - app-routing-module: se encarga de redireccionar a las vistas principales
 - guards/authorization.guard: se usa para chequear que el usuario tenga el rol indicado antes de entrar a la vista indicada
 - components: se divide en:

- una subcarpeta por cada rol con los componentes exclusivos de ese rol. Ejemplo, en tester esta filter-bug-component
 - Los componentes que sean compartidos por 2 o más roles se colocan directamente en /components. Se intentó agrupar en carpeta pero el vscode tiro error (posible mejorar para después)
- models: con las clases que representan las entidades del dominio, ejemplo Bug, User, Project etc.
- services: servicios que se utilizan para comunicarse con la web api. Se aplicó experto y SRP para que solo en una parte del código estén todas las clases que se comunican con la web api
- utils/Display: se utiliza para mostrar datos del dominio de una manera más amigable. Ejemplo en vez de true/false, usa Resolved/Unresolven. Cumple OCP ya que permite agregar más visualizaciones sin modificar las anteriores y ya son compatibles con todos los componentes.
- environment: donde se encuentra el endpoint para no tener que repetirlo en todos los servicios distintos y así cumplir con Clean Code
- Una de las principales decisiones de diseño orientada a la no repetición de código fue la utilización de componentes propios y “genéricos”
 - Contamos con una tabla y form genérica las cuales pueden ser extendida por otros componentes
 - Si un componente quiere usar una tabla no tiene que definirla de nuevo o copiar el código puede extender la que ya existe, pasándole como Input los valores y headers que quiere mostrar
 - Todo esto mejoró la mantenibilidad (ej. un bug/cambio en una tabla se arregla solo en un lado), reducción de la repetición de código, asegurar una interfaz similar y clara en todos lados
- Si bien no se definió como un requisito utilizar las styleguide de angular, si se usaron como referencia y se siguieron muchas de ellas. Ej: el nombre de las clases (service, component, module), funciones chicas, separación en distintos archivos de html, css, ts y en más archivos de cada uno si se volvía complicado de entender en uno solo. SRP para cada componente.
- En ningún lado se pregunta por el rol del usuario logueado, permitiendo extensibilidad a la hora de agregar más roles.

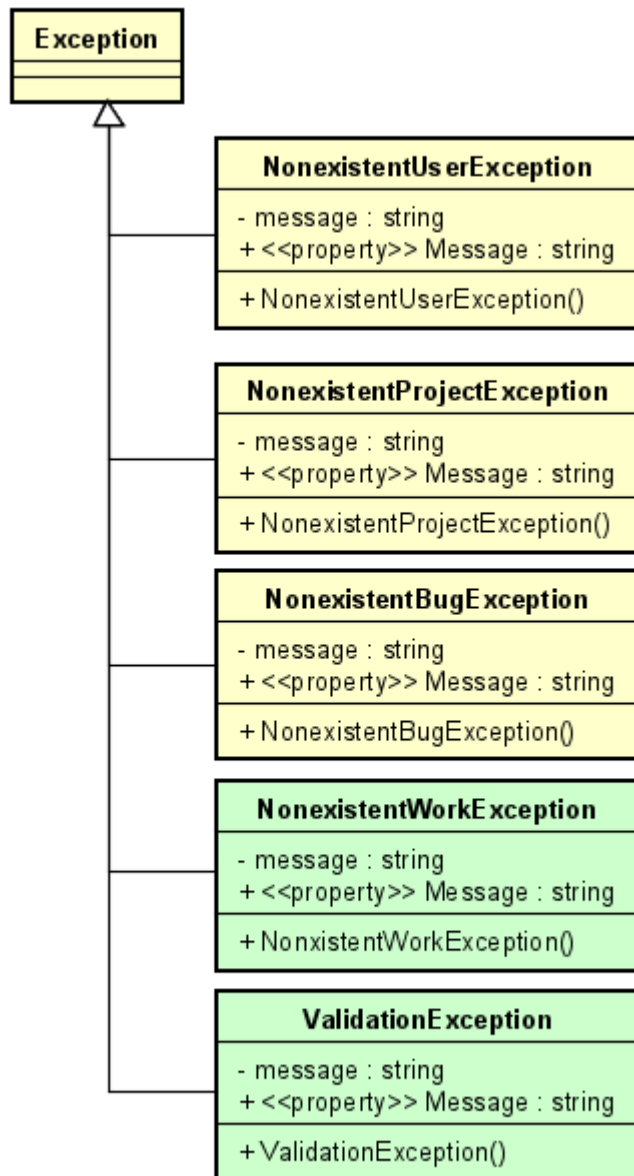
Usabilidad

Durante el desarrollo de la frontend se buscó llegar a un sistema con buena usabilidad. Por lo tanto se tuvieron en cuenta las heurísticas de Nielsen. Se incluye un análisis de las mismas en el Anexo Heurísticas de Nielsen

Descripción del manejo de excepciones

El manejo de excepciones a nivel de backend se mantiene igual que la entrega anterior, como está explicado en el "Anexo manejo de excepciones obligatorio 1"

Lo que si querías recalcar acá son algunas nuevas excepciones que agregamos, notablemente la validation exception y excepción de Tarea:



Anexo

Anexo Heurísticas de Nielsen

- El diseño de nuestra aplicación mantiene siempre informado al usuario, mostrando alertas y mensajes de error cuando los datos son ingresados de forma errónea. También se utiliza un icono de carga, por ejemplo al guardar o hacer el login, para brindarle al usuario un feedback del estado de la aplicación.
- Se informa también cuando los cambios fueron implementados correctamente con mensajes en color verde, indicando una acción positiva y en rojo en el caso contrario.
- Se utilizan términos y conceptos simples y familiares para cualquier tipo de usuario y logos intuitivos que acompañan acorde a esos conceptos. El usuario tiene el control y la libertad de deshacer su acción mediante el botón de cancelar y navegar por la sidebar todas las opciones desde cualquier vista de la aplicación.
- Existe consistencia en toda la aplicación en cuanto a los conceptos, en este caso la interfaz es en inglés, un ejemplo sería que en cada vista de la aplicación para las tareas se usa la palabra “Assignment” y nunca un sinónimo de esta, como podría ser “Task” o “Work”.
- Nuestra aplicación informa sobre posibles errores en los formularios. Se realiza la validación de los campos antes de permitirle al usuario guardar un nuevo proyecto o bug.
- La información que necesita el usuario siempre se muestra en pantalla, no es necesario que el usuario recuerde interfaces anteriores. Por ejemplo al agregar una tarea no es necesario que el usuario recuerde el nombre del proyecto al cual quiere agregar la tarea, sino que se mostrará la lista de proyectos disponibles.
- Diseño minimalista y estético, no se muestra nada más de lo necesario en la interfaz. Hay consistencia de colores en los botones y en el resto del contenido visual.
- Se usan mensajes de error tradicionales, en letra roja, en lenguaje plano, sin mostrar código de errores irrelevantes para el usuario.

Anexo API

Se incluye en la documentación una colección de postman actualizada con todos los endpoints.

Para diseñar la api se diseñaron endpoint que cumplan con las recomendaciones de nombre de Rest Api, por ejemplo lo que estan detallado aca: <https://restfulapi.net/resource-naming/>

Algunas decisiones que se tomaron:

- Se definieron recursos como admin, project, developer y se escriben en plural al principio del endpoint
- Se utilizaron los métodos HTTP para indicar acciones sobre los recursos y no se incluyeron verbos en la URI. Por ejemplo:
 - GET /bugs/{bug_id} devuelve un bug específico
 - DELETE /bugs/{bug_id} lo elimina
- Todos los endpoints son en minúscula y no se incluye “/” al final
- Se utiliza tanto los parámetros por el body, los header o la route de la request dependiendo de que es mejor para cada caso

URL base: <http://localhost:5000/>

Códigos de error:

Como se mencionó en otro documento se establecieron códigos de error como 404 y 500 para cuando ocurre una falla en la petición del recurso. En caso que no ocurra error se devuelve 200 OK, al menos que sea una eliminación donde se puede devolver 201 NoContent

Endpoints

Login:

Login	
POST	/login

Ejemplo de la request en el body:

```
{
  "username": "admin",
  "password": "123admin"
}
```

Respuesta:

```
{
  "token": "4b922d69-2c00-4665-a91b-3b88088ff710",
  "role": "admin"
}
```

Admin:

Admin	
POST	/admins

Ejemplo de la request en el body:

```
{
  "username": "Administrator1",
  "name": "Lucía",
  "lastname": "López",
  "password": "admin123",
  "email": "lucia123@gmail.com"
}
```

Y además en los headers tiene que haber un token de login de admin

<input checked="" type="checkbox"/> token	4b922d69-2c00-4665-a91b-3b88088ff71d
---	--------------------------------------

Sin el token la respuesta será:

```
{
  "responseMessage": "You aren't logged."
}
```

Con 401 Unauthorized.

Con el token correcto se devuelve el mismo admin que se agregó con 200 OK.

Bug:

Bug		▼
GET	/bugs	
POST	/bugs	
GET	/bugs/{id}	
PUT	/bugs/{id}	
DELETE	/bugs/{id}	
POST	/bugs/import/{format}	

Se definieron todas las operaciones para el Bug, y todas parten desde la misma URI “bugs”

GET	/bugs
-----	-------

Se devuelve 200 OK cuando se logra conseguir los bugs correctamente.

Utiliza el token login para filtrar los bugs dependiendo si se está logueado como admin, developer o tester utilizando el mismo endpoint.

Los únicos cambios que tienen los siguientes endpoints es que utilizan el authorization filter

POST	/bugs
GET	/bugs/{id}
PUT	/bugs/{id}
DELETE	/bugs/{id}
POST	/bugs/import/{format}

Para la importación de bugs se definió que se elija el formato o “empresa” en la URI y el path al archivo en el servidor que esté en los headers. De esta manera, no hay complicaciones a la hora de probar el path en Postman u otra herramienta. Si se pusiera en la URI mismo los espacios y los “/” podrían complicar la lectura.

Entendemos que una regla es no incluir formato de archivos en las URI, sin embargo en este caso, el formato representa más una empresa que el formato mismo. Un posible “formato” podría ser “empresa2” y no necesariamente cada formato estar asociado a un tipo de archivos en particular.

Nuevos endpoints:

GET	/bugs/custom-importers
POST	/bugs/custom-importers
PUT	/bugs/{id}/resolve
PUT	/bugs/{id}/unresolve

POST	/bugs/custom-importers
Parameters	
No parameters	
Request body	
Example Value Schema	
<pre>{ "importerName": "string", "params": [{ "name": "string", "type": 0, "value": "string" }] }</pre>	

PUT	/bugs/{id}/resolve
PUT	/bugs/{id}/unresolve

Resolve y unresolve utilizan el authorization filter para que solo developer lo pueda utilizar y la request se hace solo por route

Developer:

Al igual que para bugs se definió una ruta donde devs sea el principal actor.

POST /devs

Ejemplo:

```
{
  "username": "juanperez",
  "name": "Juan",
  "lastname": "Perez",
  "password": "123dr568",
  "email": "juanperez@gmail.com"
}
```

Para postear un usuario es necesario ser admin, por lo tanto se utilizo en la header de la request el token login para verificar esto.

GET /devs/{id}/bugs/quantity

Endpoint nuevo:

GET /devs

Los interesados en conseguir la lista de todos los developers son al admin y los tester, que pueden agregar un dev a un proyecto.

Project:

Project	
GET	/projects
POST	/projects
GET	/projects/{id}
PUT	/projects/{id}
DELETE	/projects/{id}
GET	/projects/name/{name}
GET	/projects/{id}/bugs
GET	/projects/{id}/bugs/quantity
GET	/projects/{id}/devs
GET	/projects/{id}/testers
DELETE	/projects/{idProject}/devs/{idDev}
POST	/projects/{idProject}/devs/{idDev}
DELETE	/projects/{idProject}/testers/{idTester}
POST	/projects/{idProject}/testers/{idTester}

Las acciones que donde si era necesario especificar el proyecto, se decidió incluir en la ruta de projects.

Nuevos endpoints:

GET	/projects/{id}/cost
GET	/projects/{id}/duration

Para el segundo obligatorio se pedia calcular la duracion y costo de cada proyecto. Se obtiene el id del proyecto desde la ruta.

Tester:

Una de las ideas importantes que consideramos para la URI era que fuera consistente. Es decir, que no se tenga que usar un formato de URI para los devs y otro para los testers. Ejemplo, el POST es similar para ambos y sucede lo mismo con el proyecto.

Tester	
POST	/testers
GET	/testers/{idTester}/bugs/status/{filter}
GET	/testers/{idTester}/bugs/name/{filter}
GET	/testers/{idTester}/bugs/project/{filter}

El unico endpoint nuevo es:

GET	/testers
-----	----------

Con el mismo formato que para developers.

Work:

POST	/works
GET	/works
GET	/works/{id}

Ejemplo de una request para Post:

```
{  
  "name": "tarea 6",  
  "time" : 5,  
  "cost": 3,  
  "ProjectId": 3  
}
```

Tambien se incluye el token login en el header para la autorización de quien puede crear un work, en este caso un admin o un tester.

En el Get solo se incluye el token login en la request. Ese token se utiliza para obtener los works necesarios para cada tipo de usuario dependiendo de cual este logueado.

Anexo Cobertura de Código

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.dll	2705	75,75 %	866	24,25 %
Repository	27	3,02 %	866	96,98 %
Repository.Design	10	100,00 %	0	0,00 %
Repository.Migrations	2668	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.dll	156	46,85 %	177	53,15 %
WebApi	69	100,00 %	0	0,00 %
WebApi.Controllers	0	0,00 %	177	100,00 %
WebApi.Filters	87	100,00 %	0	0,00 %

Comparers

En el obligatorio anterior en Dominio incluimos comparadores (ya que consideramos que dominio es el experto sobre si dos clases suyas son o no iguales), que utilizamos para tests y no todas llegaban al 100%

domain.dll	24	12,06 %	175	87,94 %
Domain.Utils	24	19,83 %	97	80,17 %
Domain	0	0,00 %	78	100,00 %

Sin embargo en el obligatorio 2 el coverage mejoró en Dominio

domain.dll	18	5,25 %	325	94,75 %
Domain	2	1,14 %	174	98,86 %
Domain.Utils	16	9,58 %	151	90,42 %

Factory bug parser

En el obligatorio anterior 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 y había quedado sin testear.

Dicho factory es la única clase que no tenía 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 %

En el obligatorio 2 ahora esta 100% testada:

bugparser.dll	0	0,00 %	69	100,00 %
{ } BugParser	0	0,00 %	69	100,00 %
BugModel	0	0,00 %	34	100,00 %
BugParserXML	0	0,00 %	19	100,00 %
BugXML	0	0,00 %	8	100,00 %
BugsXML	0	0,00 %	2	100,00 %
ParserFactory	0	0,00 %	6	100,00 %
GetBugParser(D...	0	0,00 %	6	100,00 %

Custom Bug Importer

No se testea el mensaje de la excepción

custombugimporter.dll	16	19,75 %	65	80,25 %
{ } CustomBugImporter	16	19,75 %	65	80,25 %
CustomBugImporte...	14	17,72 %	65	82,28 %
ImporterManagerEx...	2	100,00 %	0	0,00 %
ImporterManagerException(string)		100,00 %	0	0,00 %

```
1 reference | IvanMonjardin, 3 days ago | 1 author, 1 change
public ImporterManagerException(string message) : base(message) { }
}
```

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

▷	bugparser.dll	0	0,00 %	69	100,00 %
▷	businesslogic.dll	6	1,52 %	390	98,48 %
▲	custombugimporter.dll	16	19,75 %	65	80,25 %
▲	{ } CustomBugImporter	16	19,75 %	65	80,25 %
▷	CustomBugImporte...	14	17,72 %	65	82,28 %
▷	ImporterManagerEx...	2	100,00 %	0	0,00 %
▷	domain.dll	18	5,25 %	325	94,75 %
▷	dto.dll	29	8,76 %	302	91,24 %
▲	repository.dll	2705	75,75 %	866	24,25 %
▷	{ } Repository	27	3,02 %	866	96,98 %
▷	{ } Repository.Design	10	100,00 %	0	0,00 %
▷	{ } Repository.Migrations	2668	100,00 %	0	0,00 %
▷	testbugparser.dll	3	3,30 %	88	96,70 %
▷	testbusinesslogic.dll	108	4,46 %	2313	95,54 %
▷	testcustombugimporter.dll	1	1,02 %	97	98,98 %
▷	testdataaccess.dll	14	0,90 %	1540	99,10 %
▷	testdomain.dll	33	6,52 %	473	93,48 %
▷	testdto.dll	0	0,00 %	294	100,00 %
▷	testwebapi.dll	58	4,36 %	1271	95,64 %
▲	webapi.dll	156	46,85 %	177	53,15 %
▷	{ } WebApi	69	100,00 %	0	0,00 %
▷	{ } WebApi.Controllers	0	0,00 %	177	100,00 %
▷	{ } WebApi.Filters	87	100,00 %	0	0,00 %

Anexo Respaldos Base de Datos

Existen dos respaldos de la base de datos, en la carpeta Database/EmptyDatabase se encuentra el archivo .bak vacío, que incluye un administrador básico para poder utilizar el resto de las funcionalidades, con los siguientes datos y su correspondiente script:

	Id	Username	Name	Lastname	Password	Email
1	1	admin	Juan	Perez	123admin	admin@admin.com

En la carpeta Database/DemoDatabase se encuentra el archivo .bak con datos de prueba para la demo en todas las tablas y su correspondiente script. Los datos fueron generados desde el frontend, mientras eran probados todos los requerimientos de la aplicación.

Tabla de Bugs:

	Id	Name	Description	Version	IsActive	ProjectId	ProjectName	CompletedById	Time
1	1	Get bugs endpoint	Endpoint does not work, possible problem with da...	1.0	0	1	Bug Manager	2	2
2	2	Fields validation	Theres no validation on frontend forms	1.4	0	1	Bug Manager	NULL	4
3	3	Password encryption	Saved passwords should be encrypted	2.7	1	2	Password Manager	NULL	6
4	4	Animation	Animation bug when right click	1.8.7	1	5	Minecraft Mod	NULL	7
5	5	Button redirection	Button in forms does no redirect correctly	2.9.0	0	6	Personal Project	1	1

Tabla de Projects:

	Id	Name
1	1	Bug Manager
2	2	Password Manager
3	4	3D Game
4	5	Minecraft Mod
5	6	Personal Project

Tabla de Works:

	Id	Name	Cost	Time	ProjectId	ProjectName
1	1	Calculate project total cost	6	2	1	Bug Manager
2	2	Assignment controller Web Api	9	3	1	Bug Manager
3	3	Password sharing	4	7	2	Password Manager
4	4	Character Models	5	17	4	3D Game
5	5	Game mechanics	4	20	4	3D Game
6	6	Create modified environment	8	9	5	Minecraft Mod

Anexo Pendientes obligatorio 1

- Login o identificación del usuario que utiliza la web api
- Importación de archivo mediante texto posicional (si se incluyeron factories, interfaces y otras clases que podrán facilitar su desarrollo en un futuro)
- Cantidad de bugs resueltos por un desarrollador, no funciona siempre
- El filtrado de bugs por proyecto, nombre etc. fue implementado en todas las capas pero no logramos que funcione en la capa de data access
- Conseguir la lista de los proyectos de un desarrollador
- La modificación de bugs permite modificar de activo a resuelto, pero consideramos mejor práctica hacer un endpoint más específico para esto.
- Validación del dominio o de gran parte de los datos/ agregar más excepciones. El manejo de excepciones si está

Anexo Decisiones de Diseño obligatorio 1

- Cada interfaz (IBusinessLogic, IDataAccess, etc) van en proyectos a parte y no en una carpeta dentro de la clase concreta. Esto habilita a no depender de la clase concreta en caso que usen el mismo namespace y hace la solución más extensible. Se puede agregar y eliminar clases concretas sin modificar la interfaz tanto de lugar como sus métodos. Lo único que hay que cambiar es el factory de la inyección de dependency en caso que se quieren cambiar las clases concretas.
- Cuando hubo interfaces similares, por ejemplo en IBusinessLogic donde muchas de las clases van a implementar métodos de CRUD, se agruparon los métodos en común

en una interfaz usando Generics (IBusinessLogic<T>) permitiendo que se hagan nuevas interfaces específicas relacionadas a cada clase del dominio. Esto primero era para no repetir código y permitiendo en un futuro se quiere cambiar algún método, por ejemplo el retorno de Delete o similar, se puede hacer fácilmente y no haya que modificar varias clases. Más que nada esto da una coherencia a la hora de utilizar métodos básicos de los recursos. De todos modos, siempre es mejor una interfaz más específica que una generica que después tengan que implementar las clases concretas. Los métodos de bugs no deberían pertenecer a la interfaz de proyecto y vice versa, apuntamos siempre a que las clases usen las interfaces más específicas que puedan, siguiendo los principios de diseño.

- Al igual que no tenemos las interfaces dentro de una clase concreta, tampoco tenemos los test dentro de la clase. Nos pareció mejor tener proyectos separados. Una razón es que así hay responsabilidades claras y no hay un mismo proyecto que hace dos cosas (ejecución y pruebas).
- Se investigó y se decidió que el responsable de hacer, por ejemplo, SaveChanges() a la base de datos es el data Access y no , por ejemplo, la businessLogic. Nosotros vimos cómo su “singular responsibility” administrar todo lo relación al almacenamiento de datos. Similar a como podría suceder en una base de datos no code first donde se le manda consultas SQL posiblemente complejas para que ejecute la BD y retorne algún valor. Esto permitió no incluir operaciones específicas de data access al resto en los proyectos. Si bien a veces podría ser útil ejecutar dos operaciones
- El bug Parser(importación del ob. 1) es el responsable de abrir el archivo y devolverme su lista de bugs del archivo. La idea es que la única responsabilidad de esa clase es devolver una lista de bugs de un path, cómo lo hace no debe importarle a las otras clases. Por lo tanto, no nos pareció bien pasarle un Stream File o hacer parte de lo que se consideraría importar bugs de archivos en otra clase.
- Se utilizaron archivos xml de pruebas para los tests de Bug Parser y métodos de mstest para moverlos a la carpeta de Debug para que estén disponible a la hora de ejecutar los tests. No nos pareció bien ni:
 - Poner el archivo en el código y pasarlo de alguna manera, ya que ocuparía mucho espacio en el código y dificulta la lectura.
 - Armar los archivos durante los tests. Sería poner mucha lógica innecesaria en los tests
 - Utilizar algún tipo de Mock para files. Primero porque creo que no se puede mockear al ser estática y segundo porque, como dijimos antes, la responsabilidad de la clase incluye poder utilizar el archivo que le pasen.
- Entendemos de la letra que si en el futuro se agregan más formas de importar bugs para otras empresas, que van a ser siempre archivos. Y no, por ejemplo, pasarle un binario mediante la web api sin antes guardarlo en un archivo. De todos modos, se diseñó para que se pueda agregar cualquier tipo de archivo en el futuro.
- Utilizamos dos factories de servicios para cada capa de la inyección de dependencias de businesslogic y data access. Poner junto la inyección de dependencias de business logic y de data access, rompía el single-responsibility principle. Tampoco está bueno,

poner todas las capas en una misma clase, de la forma que lo implementamos se podría agregar una tercera capa más fácilmente (sin modificar los archivos ya existentes).

- Dejamos el default de EF sobre la primera Id de las entidades siendo 1. Permitiría diferenciar entre un valor posiblemente válido y el 0 que seguro no es válido.
- En la api de importación de bugs, elegimos que el usuario mande el path del archivo por el header y no por la route, así se soluciona el problema de mandar algunos caracteres no válidos en la URI.

Anexo Decisiones Repositorio

Se establecieron algunos estándares para trabajar en el repositorio, como por ejemplo:

- El nombre de las ramas feature es "feature-NombreDeLaRama". Se utiliza UpperCamelCase
- Los commits se escriben en presente (Ej. "Adds modify button").
- Para arreglar una funcionalidad se crea una rama "fix-NombreDeLaRama"
- Si se generó un problema al mergear a develop, se puede arreglar y crear una rama fix.
- Utilizamos pull request donde la otra persona tiene que aceptar la pull request
- Para hacer que git detecte carpetas vacías se agregó un archivo .gitkeep según la convención
- Los commits automáticos de Github (ej, Merge pull request) no los modificamos
- No se puede mergear a develop directamente sin la aprobación de la otra persona

Anexo Carpeta de Importadores

Para indicar en qué carpetas van los .dll se utiliza el appsetting.json del proyecto WebApi donde CustomImporterFolder tiene el valor de la carpeta, Ejemplo:

```
{
  "ConnectionStrings": {
    "BugManagerDb":
    "Server=.\SQLEXPRESS;Database=BugManagerDb;Trusted_Connection=True;MultipleActiveResultSets=True"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "MySettings": {
    "CustomImporterFolder":
    "C:\\Repositorios\\Disiot-221025-Monjardin-239850\\Backend\\ExtensibleBugImporter\\Importers"
  }
}
```

Nota: Solo pueden haber importadores o otros .dll(aunque no sean importadores) en la carpeta elegida

Anexo Guia de instalación

Para el frontend:

- Ir a la carpeta .Frontend\BugManager partiendo del root del repositorio
- Ejecutar npm install
- Ejecutar ng serve --open para abrir la pagina en una ventana nueva

Para el backend:

- Ir a la carpeta /Backend/
- Abrir la solución Obligatorio.sln
- Correr la solución en VisualStudio o su IDE de preferencia

Para ambos se incluyen releases en la carpeta application. Ambas se pueden usar para crear servidores, por ejemplo, usando IIS. Los puertos por defecto son 80 para el backend y 81 para el front. De todos modos se pueden modificar en el main del dist, buscando “localhost”.

El sistema solo se probó con HTTP y no aseguramos que funcione sin configurar si se usa HTTPS

Para crear de nuevo la releases se utiliza

Backend

- `dotnet build -c Release`
- `dotnet publish -c Release`

Frontend

- `ng build --prod`

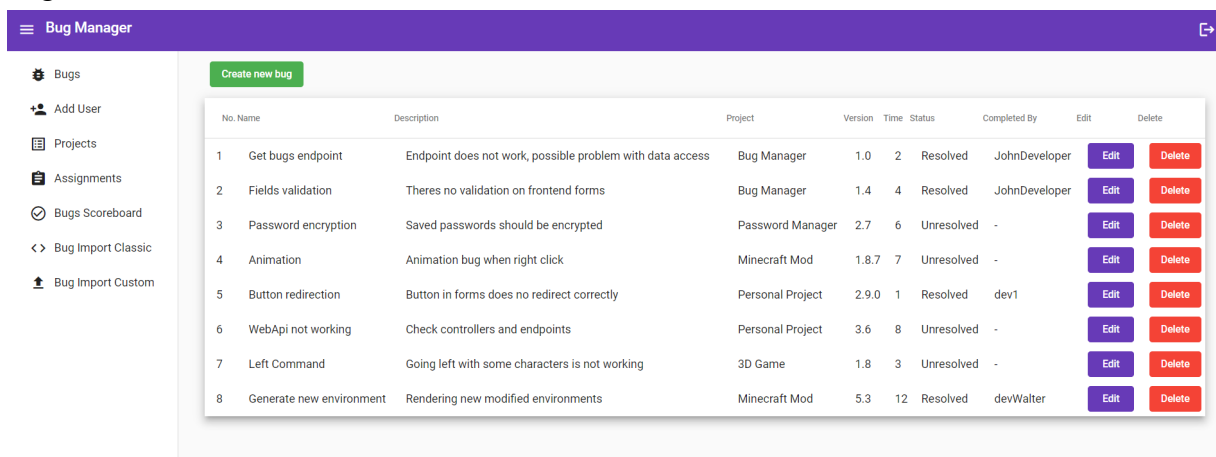
Anexo diagrama de secuencia

El diagrama tiene como objetivo que el que lo vea entienda a alto nivel como se comunica el front con el backend. Se dio prioridad a que se entienda el diagrama y capaz no tanto a que se cumplan todas las reglas de uml principalmente en la parte del front.

Este ejemplo comienza con el usuario recién logueado y comenzando a utilizar la página. Realiza un pedido de Tareas y lo recibe. Se incluye la foto completa en Documentation/Diagrams

Anexo Pantallas de Aplicación

Bugs:



No.	Name	Description	Project	Version	Time	Status	Completed By	Edit	Delete
1	Get bugs endpoint	Endpoint does not work, possible problem with data access	Bug Manager	1.0	2	Resolved	JohnDeveloper	Edit	Delete
2	Fields validation	Theres no validation on frontend forms	Bug Manager	1.4	4	Resolved	JohnDeveloper	Edit	Delete
3	Password encryption	Saved passwords should be encrypted	Password Manager	2.7	6	Unresolved	-	Edit	Delete
4	Animation	Animation bug when right click	Minecraft Mod	1.8.7	7	Unresolved	-	Edit	Delete
5	Button redirection	Button in forms does no redirect correctly	Personal Project	2.9.0	1	Resolved	dev1	Edit	Delete
6	WebApi not working	Check controllers and endpoints	Personal Project	3.6	8	Unresolved	-	Edit	Delete
7	Left Command	Going left with some characters is not working	3D Game	1.8	3	Unresolved	-	Edit	Delete
8	Generate new environment	Rendering new modified environments	Minecraft Mod	5.3	12	Resolved	devWalter	Edit	Delete

Projects:

☰ Bug Manager

⚙️ Bugs

👤 Add User

📅 Projects

📋 Assignments

🏆 Bugs Scoreboard

🔍 Bug Import Classic

📁 Bug Import Custom

Create new project

N°	Name	Total cost	Total duration (hr.)	Total bugs quantity	Testers	Developers	Edit Name	Delete
1	Bug Manager	\$123	11	2	Testers	Developers	Edit	Delete
2	Password Manager	\$28	7	1	Testers	Developers	Edit	Delete
4	3D Game	\$165	37	1	Testers	Developers	Edit	Delete
5	Minecraft Mod	\$276	21	2	Testers	Developers	Edit	Delete
6	Personal Project	\$19	1	2	Testers	Developers	Edit	Delete

Assignments:

☰ Bug Manager

⚙️ Bugs

👤 Add User

📅 Projects

📋 Assignments

🏆 Bugs Scoreboard

🔍 Bug Import Classic

📁 Bug Import Custom

Create new assignment

No.	Name	Cost	Time	Project name
1	Calculate project total cost	6	2	Bug Manager
2	Assignment controller Web Api	9	3	Bug Manager
3	Password sharing	4	7	Password Manager
4	Character Models	5	17	3D Game
5	Game mechanics	4	20	3D Game
6	Create modified environment	8	9	Minecraft Mod

Create User (Admin)

Create new user

Choose role

Admin

Save

BugsScoreboard:

Bug Manager

Bugs

Add User

Projects

Assignments

Bugs Scoreboard

Username	Name	Lastname	Email	Cost	Bugs Resolved
dev1	Agustin	Lopez	alopez@hotmail.com	\$7/hr	1
JohnDeveloper	John	Smith	jony@gmail.com	\$9/hr	2
devWalter	Walter	Carril	walt@adinet.com	\$12/hr	1

Import Bugs (Classic and Custom):

Import Bugs - Classic

Bug importation for xml files

For more importation formats, please go to [Bug Import Custom](#)

Save

Import Bugs - Custom

Please select the importer below

Choose importer *

For the original xml bug importer please go to [Bug Import Classic](#)

Save

Could not find a part of the path 'C:\Repositorios\Disiot-221025-Monjardin-239850\Backend\ExtensibleBugImporter\Importers'.