Universidad ORT Uruguay

Facultad de Ingeniería

Obligatorio Diseño Aplicaciones 2

Descripción del Diseño

Brahian Peña - 267633

Santiago Panzardi – 182742

2021

https://github.com/ORT-DA2/182742-267633.git

Declaraciones de Autoría

Nosotros, Brahian Peña y Santiago Panzardi, declaramos que el trabajo que se presenta en esta obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos el primer obligatorio de Diseño de Aplicaciones 2;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

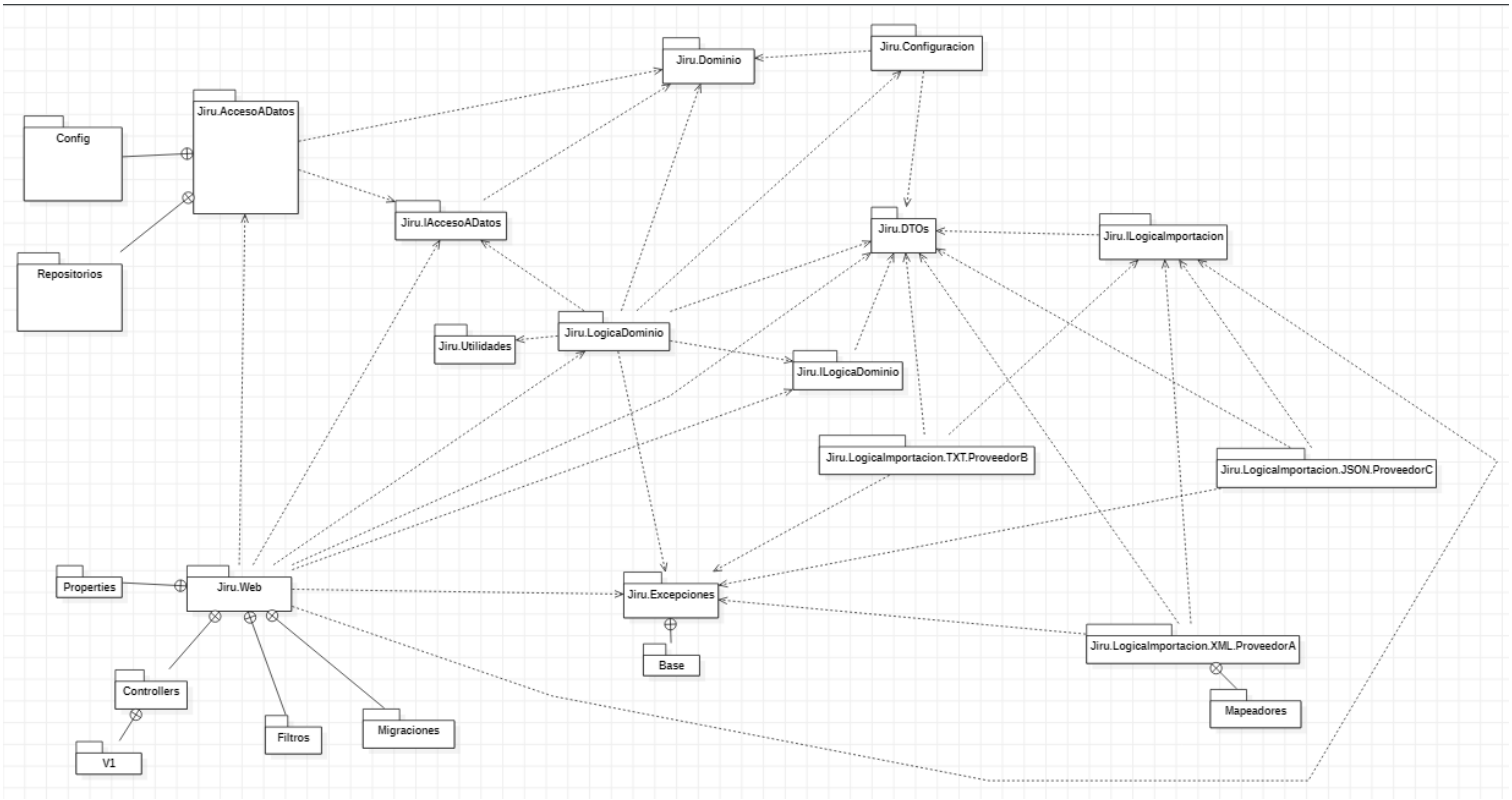
Índice

1. Descripción General del Trabajo

En este documento se describe la solución implementada, la cual es un sistema de gestión de incidencias, principalmente para proyectos de desarrollo de software. La implementación cuenta con la capacidad de persistir información en un motor de base de datos SQL (SQL Server) y operar sobre ella, mediante una capa de servicios que cumple el estándar REST (*Representational State Transfer*), esto permite separar las responsabilidades de la aplicación final llevando la mayor parte de la lógica al *backend*. Estos servicios pueden ser consumidos por terceros utilizando cualquier cliente HTTP que tenga la capacidad de enviar y recibir información en formato JSON. Para que estos servicios puedan ser consumidos, los clientes primeramente deben autenticarse, haciendo esto reciben un token que debe enviar para que el sistema pueda verificar la identidad y sus correspondientes permisos. En nuestro caso, desarrollamos una aplicación web para consumir dichos servicios. Esta es del formato SPA (single page application), y fue escrita utilizando el Framework de Google denominado Angular.

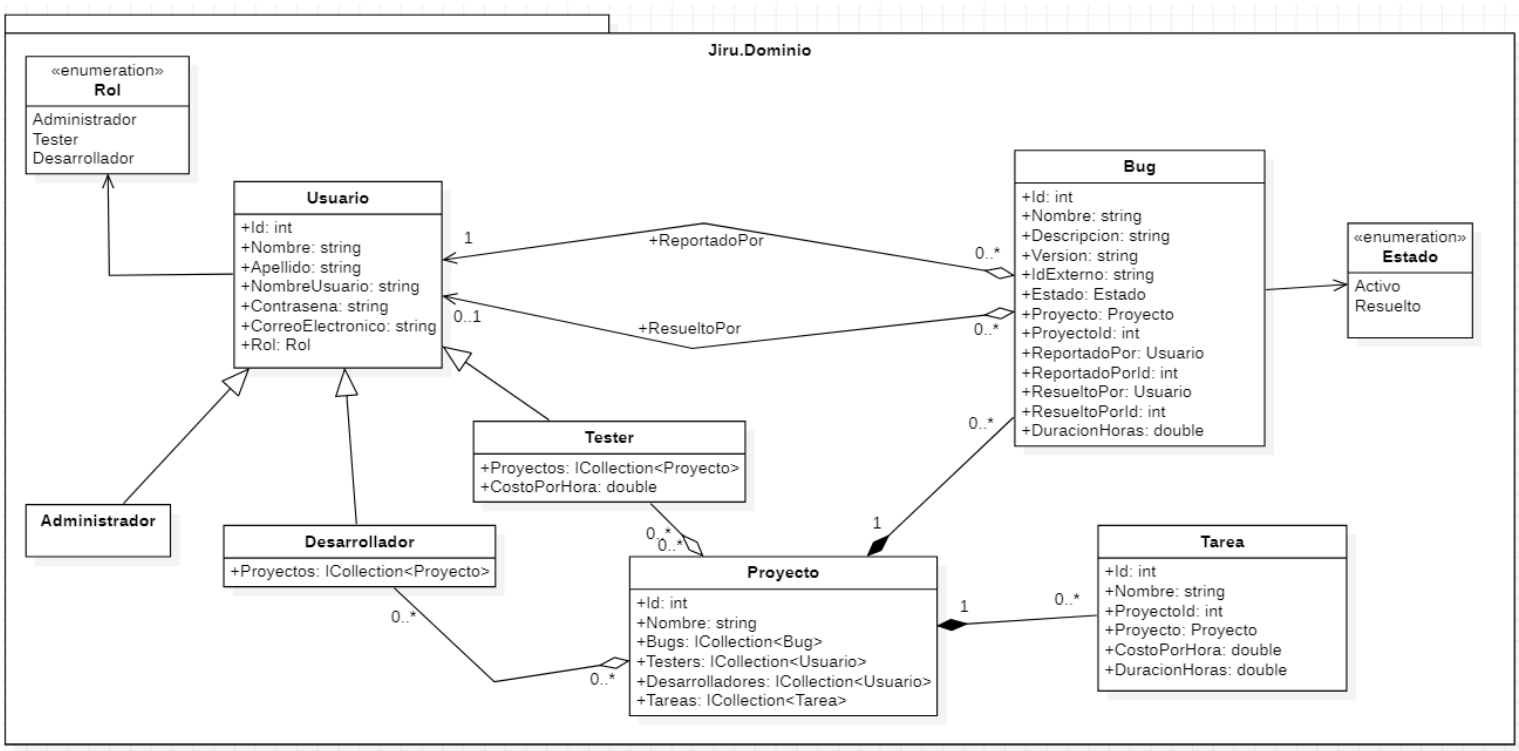
Al momento de escribir este documento, se cree haber implementado todas las funcionalidades solicitadas.

2. Diagrama de Paquetes



3. Diagramas de Clases

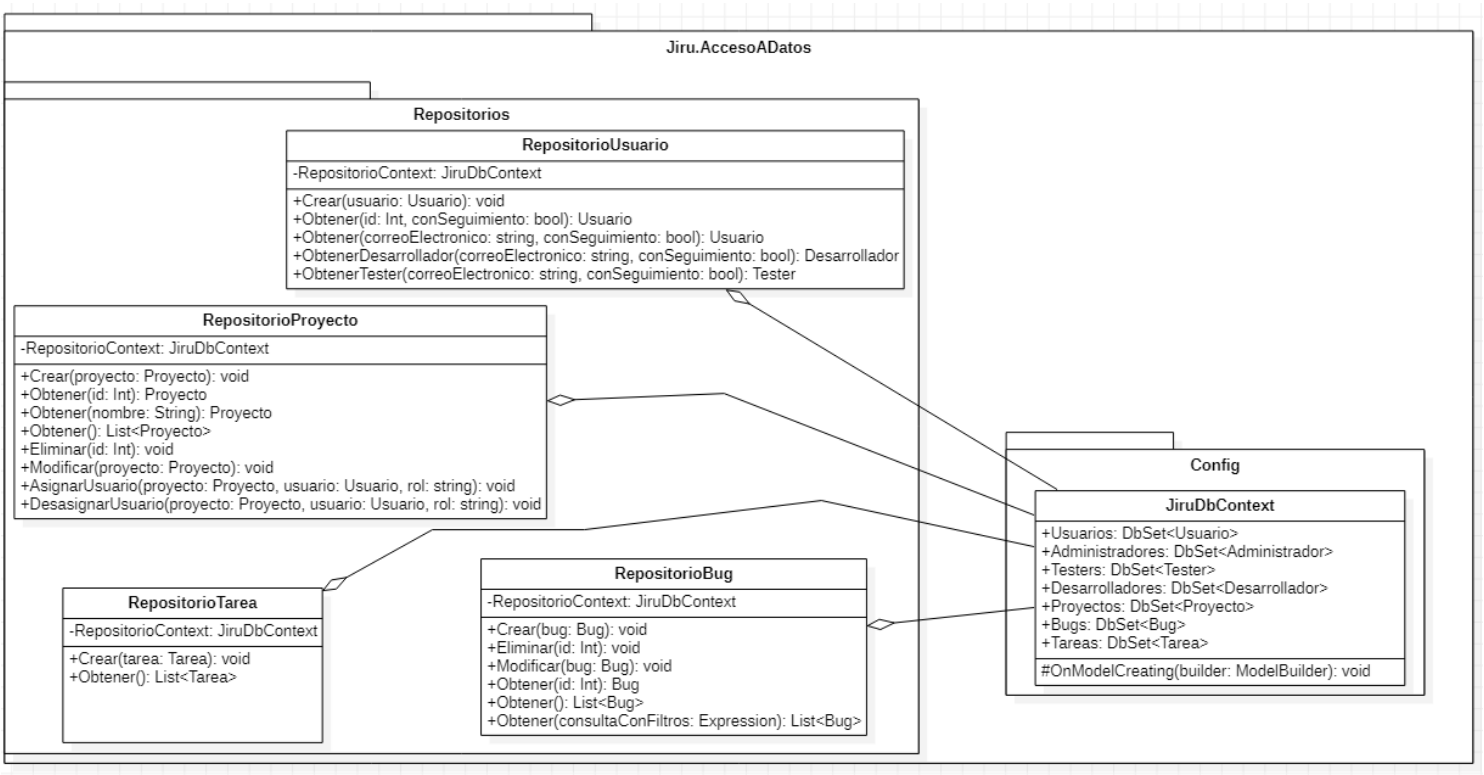
3.1 Jiru.Dominio



Descripción

El paquete *Jiru.Dominio* contiene la definición de las diferentes entidades del sistema. Se puede decir que este paquete es el núcleo de la aplicación, ya que principalmente las clases contenidas aquí son las que van a interactuar entre sí para lograr las funcionalidades requeridas. Están también serán utilizadas por *Entity Framework* para generar la base de datos, ya que se utilizó *Code First.* Se puede ver también que se implementó una herencia para el manejo de usuarios, ya que permite diferenciar que tipos de usuarios tienen listas de proyectos.

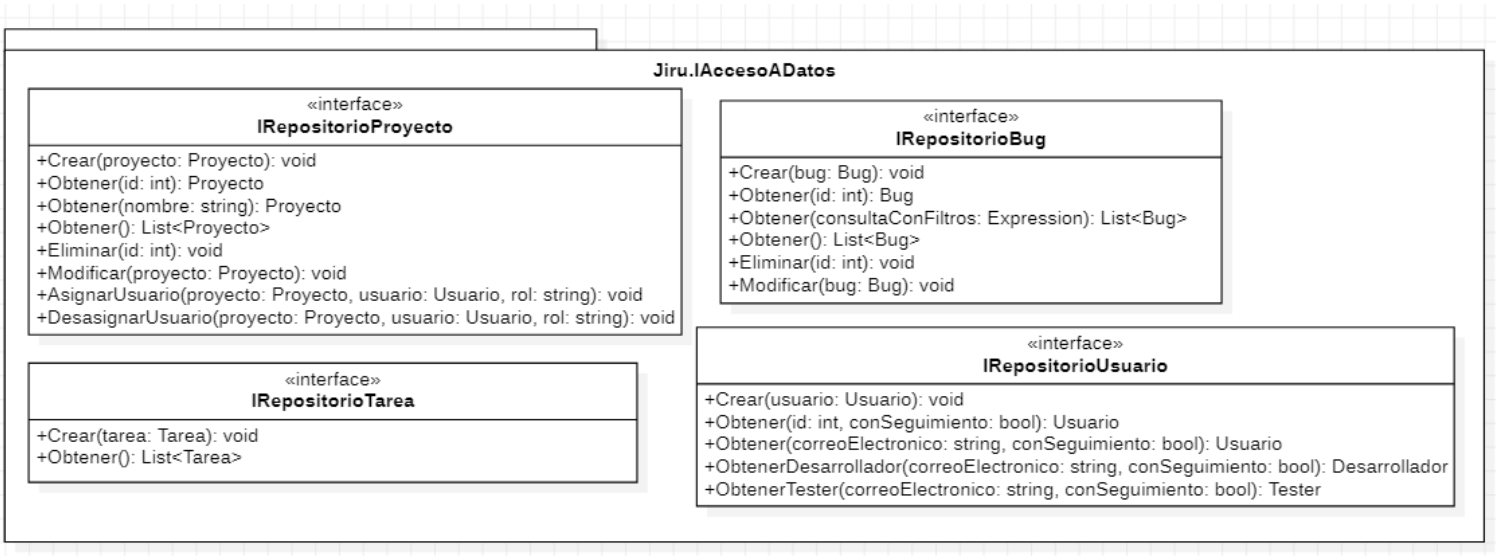
3.2 Jiru.AccesoADatos



Descripción

En el paquete *Jiru.AccesoADatos* encontramos el código necesario para interactuar con la fuente de datos. Si vemos el sub-paquete *Repositorios,* veremos los repositorios de datos de la aplicación. Como el nombre lo indica, estos implementan el patrón repositorio, ya que encapsulan la lógica requerida para acceder a las diferentes fuentes de datos (en este caso SQL Server a través de EF Core). A su vez, tenemos otro sub-paquete denominado *Config,* el cual contiene la clase de configuración requerida por EF, esta se denomina *JiruDbContext*, la cual extiende a *Microsoft.EntityFrameworkCore.DbContext.*

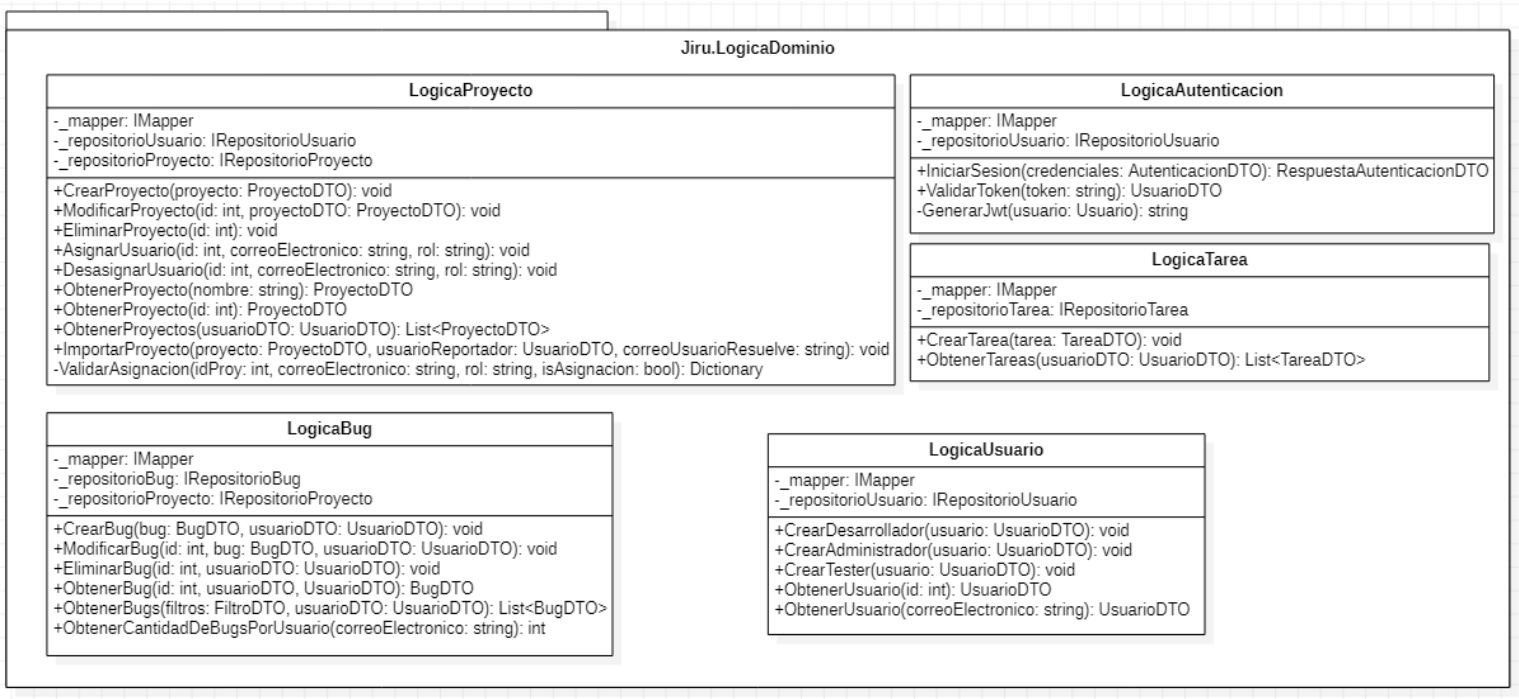
3.3 Jiru.IAccesoADatos



Descripción

En el paquete *Jiru.IAccesoADatos* encontramos las interfaces que son consumidas por las clases del paquete *Jiru.LogicaDominio*. Los repositorios previamente nombrados son los que implementan estas interfaces. Si analizamos el diseño, podemos ver que se cumple el quinto (Inversión de la dependencia) principio de los principios SOLID, ya que tanto los repositorios como las clases del paquete *Jiru.LogicaDominio* dependen de abstracciones y no de implementaciones concretas.

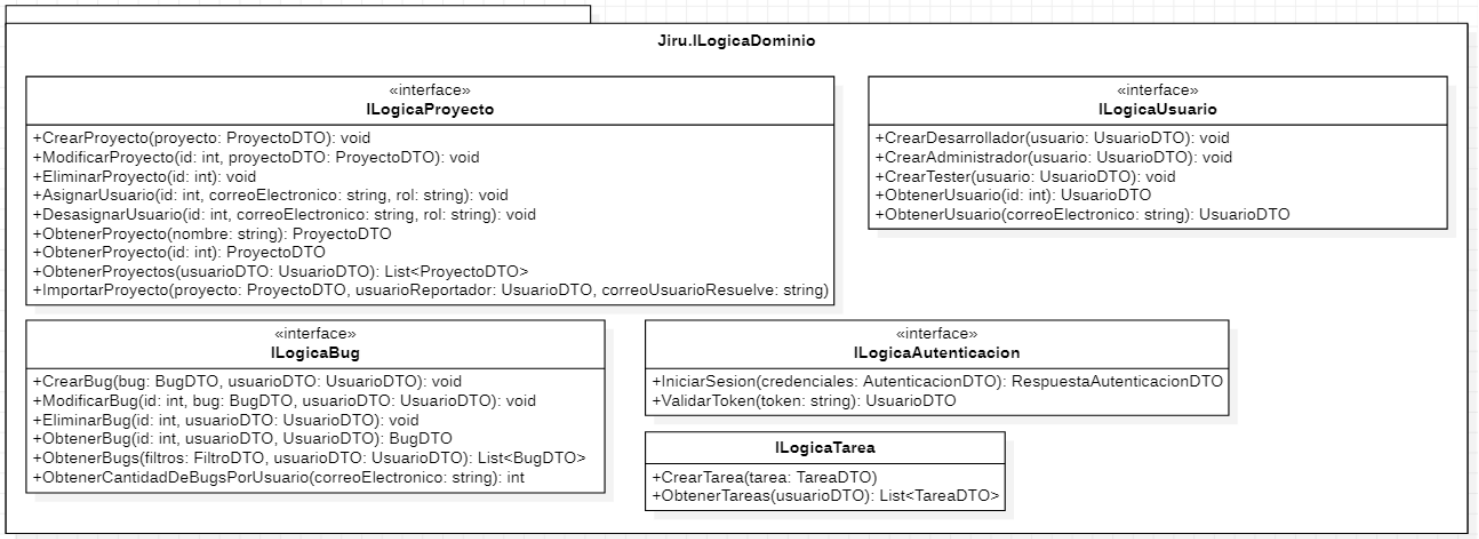
3.4 Jiru.LogicaDominio



Descripción

En el paquete *Jiru.LogicaDominio* encontramos la lógica de la aplicación. Este se podría decir que es el punto intermedio entre los diferentes módulos del sistema. En este paquete se implementan las diferentes operaciones que se pueden realizar con las entidades del sistema, ya sea operaciones básicas (CRUD), como también operaciones de autenticación y asignación. Cabe destacar que todas sus clases consumen al menos una de las interfaces expuestas por el paquete *Jiru.IAccesoADatos*. Todos los métodos declarados en las clases de este paquete reciben y retornan *DataTransferObjects* (aparte de los tipos primitivos de *C#* más *string*).

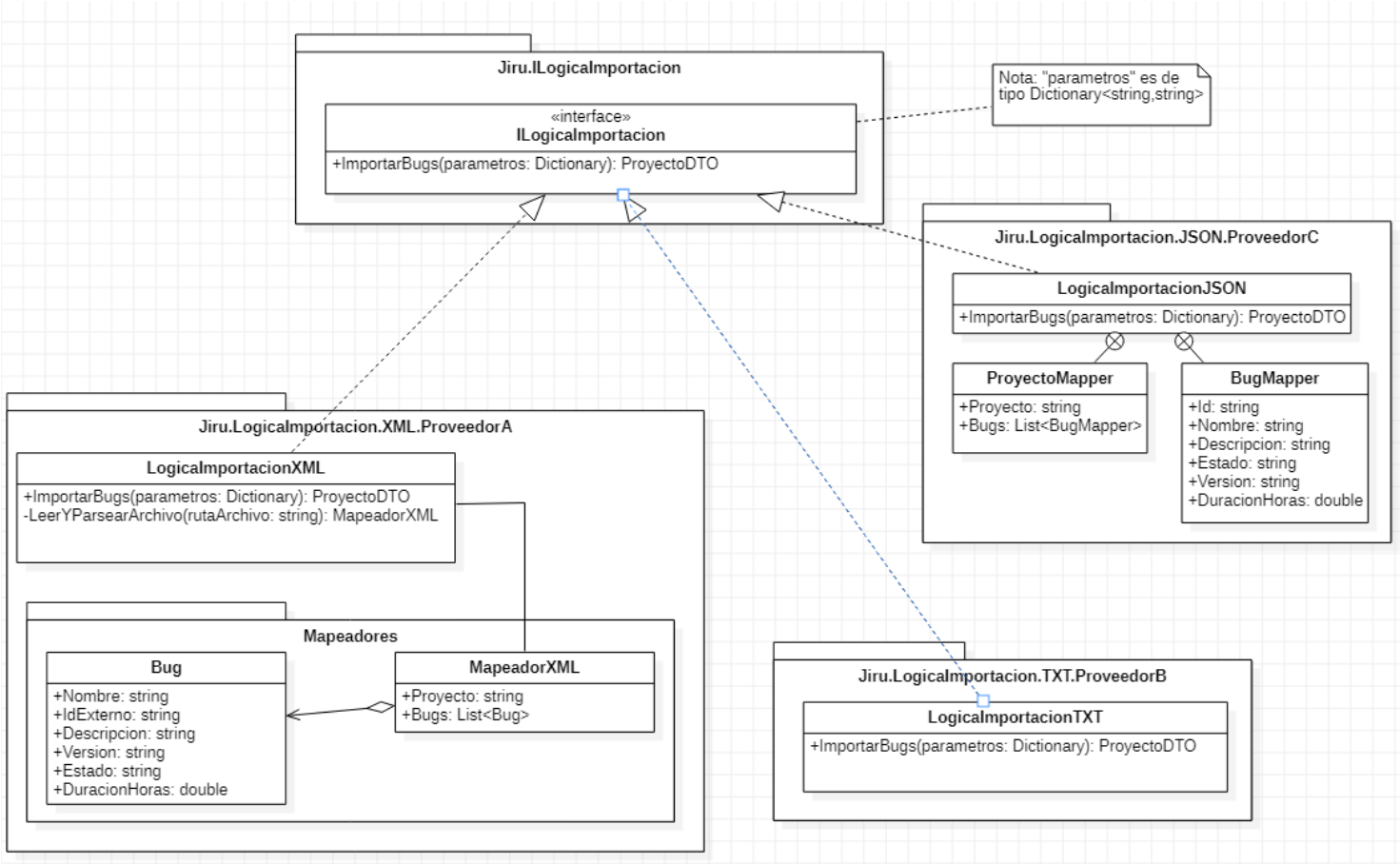
3.5 Jiru.ILogicaDominio



Descripción

En el paquete *Jiru.ILogicaDominio* encontramos las diferentes interfaces que son implementadas por el paquete *Jiru.LogicaDominio.* Estas interfaces son consumidas por las clases del paquete *Jiru.Web* y las diferentes implementaciones de *Jiru.ILogicaImportacion.* Esto nos permite desacoplar los diferentes paquetes, ya que podemos ver que se vuelve a cumplir el principio de inversión de la dependencia. También, en un futuro podríamos agregar una nueva implementación de Jiru.LogicaDominio y los paquetes que consumen estás interfaces no verían afectados.

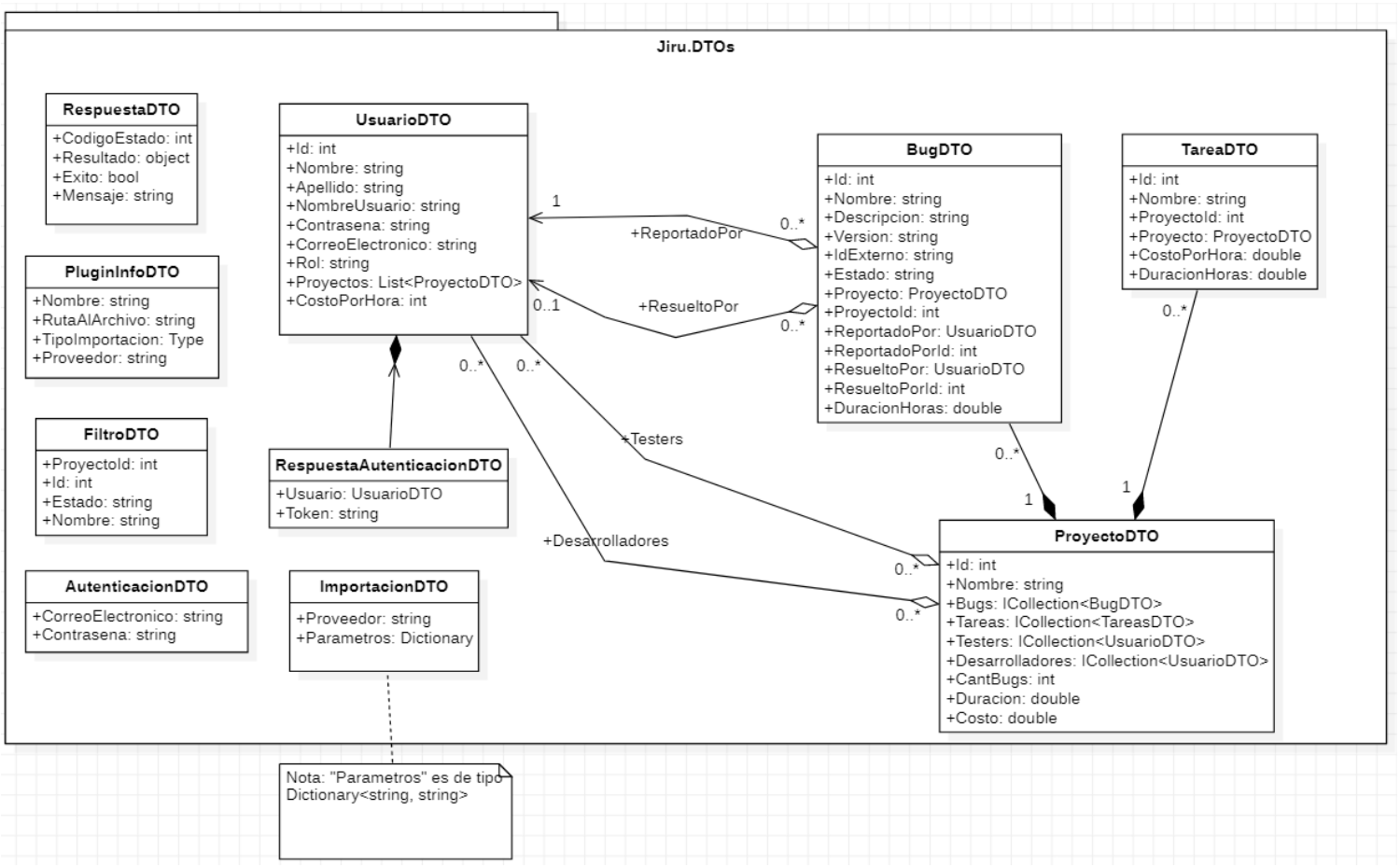
3.6 Jiru.ILogicaImportacion, Jiru.LogicaImportacionXML.ProveedorA y Jiru.LogicaImportacion.TXT.ProveedorB



Descripción

Si analizamos el paquete Jiru.ILogicaImportacion, podemos ver que este solo contiene la interfaz homónima, la cual tiene un método *ImportarBugs* que recibe un diccionario de parámetros. Luego si vamos a los otros tres paquetes, podemos ver que tenemos tres implementaciones diferentes de la interfaz anterior. Lo logrado aquí es una implementación del patrón *estrategia*, ya que se ofrecen tres implementaciones diferentes para el mismo método. La implementación que se desea utilizar se podrá elegir en tiempo de ejecución utilizando una característica de *C#* denominada *Reflection*. El objetivo principal de estas clases son la de ofrecer la capacidad de importar bugs desde fuentes externas, en este caso archivos *XML, JSON* y *TXT*, previamente cargados en el servidor.

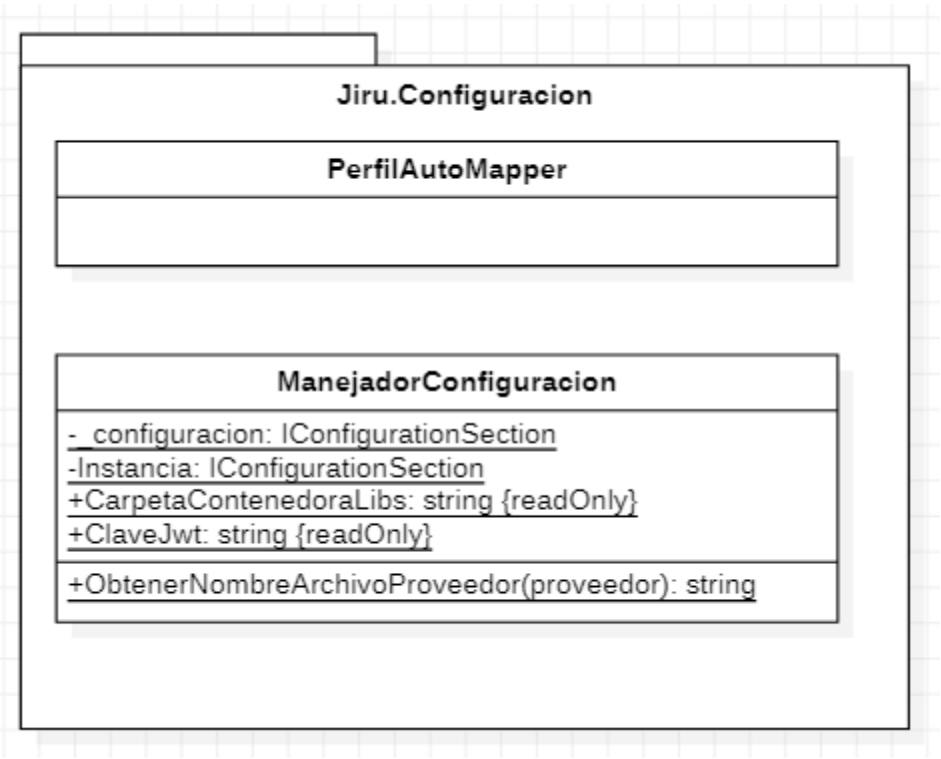
3.7 Jiru.DTOs



Descripción

En este paquete encontramos los *DataTransferObjects,* que son utilizados en diferentes paquetes. Estos permiten transferir y validar información entre capas del sistema de una forma sencilla y rápida. Podemos observar, que algunos son muy similares a las entidades previamente definidas. Estos *DTOs* son los que posteriormente se van a serializar en formato *JSON* (también la operación inversa), para ser enviados mediante *HTTP*.

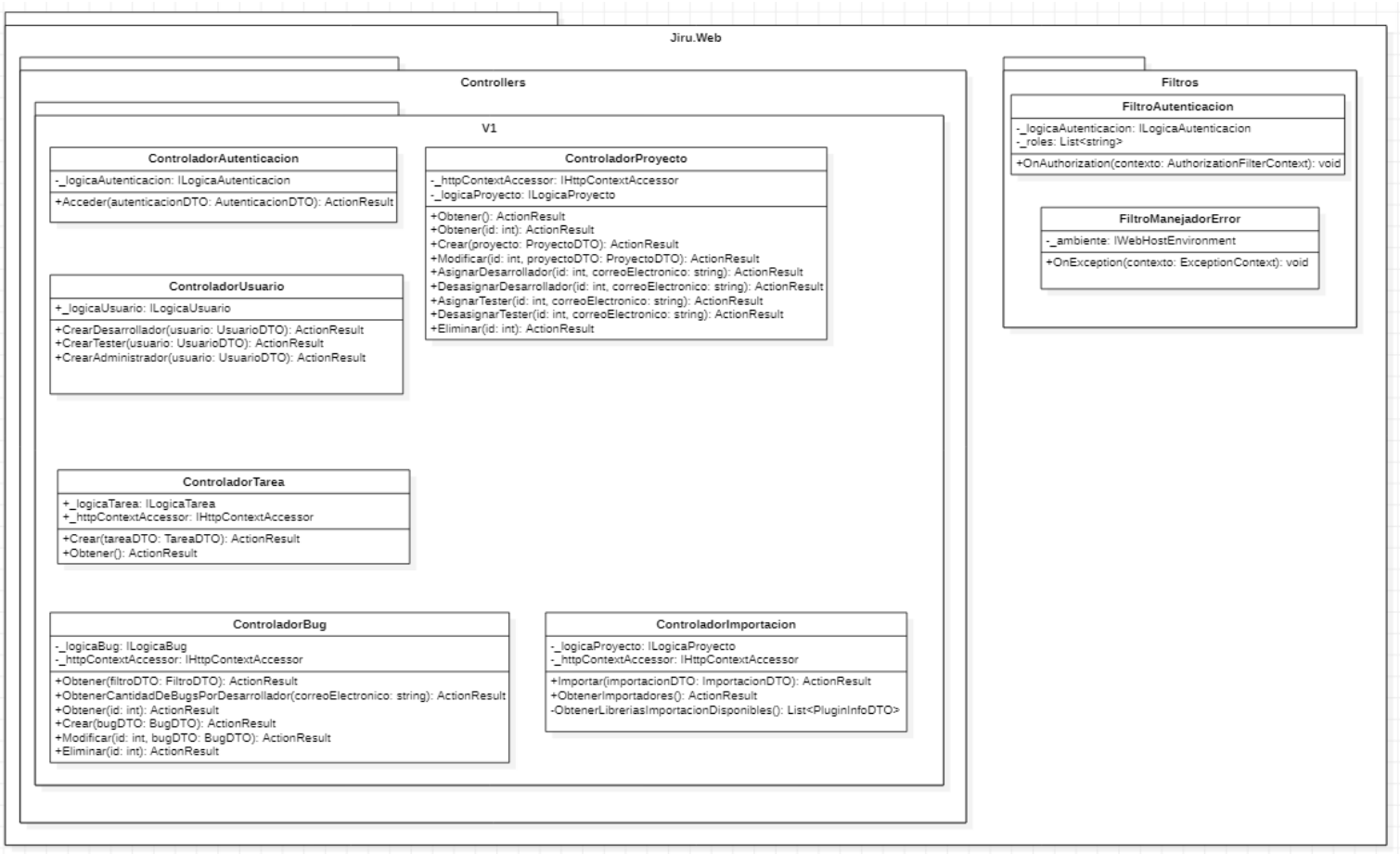
3.8 Jiru.Configuración



Descripción

En este paquete encontramos dos clases relacionadas a la configuración del sistema. La primera es *PerfilAutoMapper*, la cual define los mapeos que se pueden hacer entre *DTOs* y entidades, y viceversa. Luego tenemos la clase *ManejadorConfiguracion*, la cual es una clase estática (también sus métodos), que permite acceder a la configuración del archivo *appsettings.json.* Cabe aclarar que el método *ClaveJwt* debería obtener esa información (la clave con la que se cifran los tokens de autenticación) de otro lugar más seguro como *ParameterStore* de *AWS*, pero para evitar complejidad se decidió almacenar ese dato en el *appsettings.json.*

3.9 Jiru.Web



Descripción

En este paquete encontramos varias cosas, principalmente dos sub-paquetes: *Jiru.Web.Controllers.V1* y *Jiru.Web.Filtros*. El primero contiene la definición de la primera versión de los controladores de la API, los cuales van a ser la puerta de entrada al sistema, ya que exponen servicios que cumplen el estándar REST. Estos controladores consumen las interfaces definidas en el paquete *Jiru.ILogicaDominio.*

Seguido de los controladores, tenemos los filtros, en este caso, uno para manejar las excepciones del sistema y otro para la autenticación.

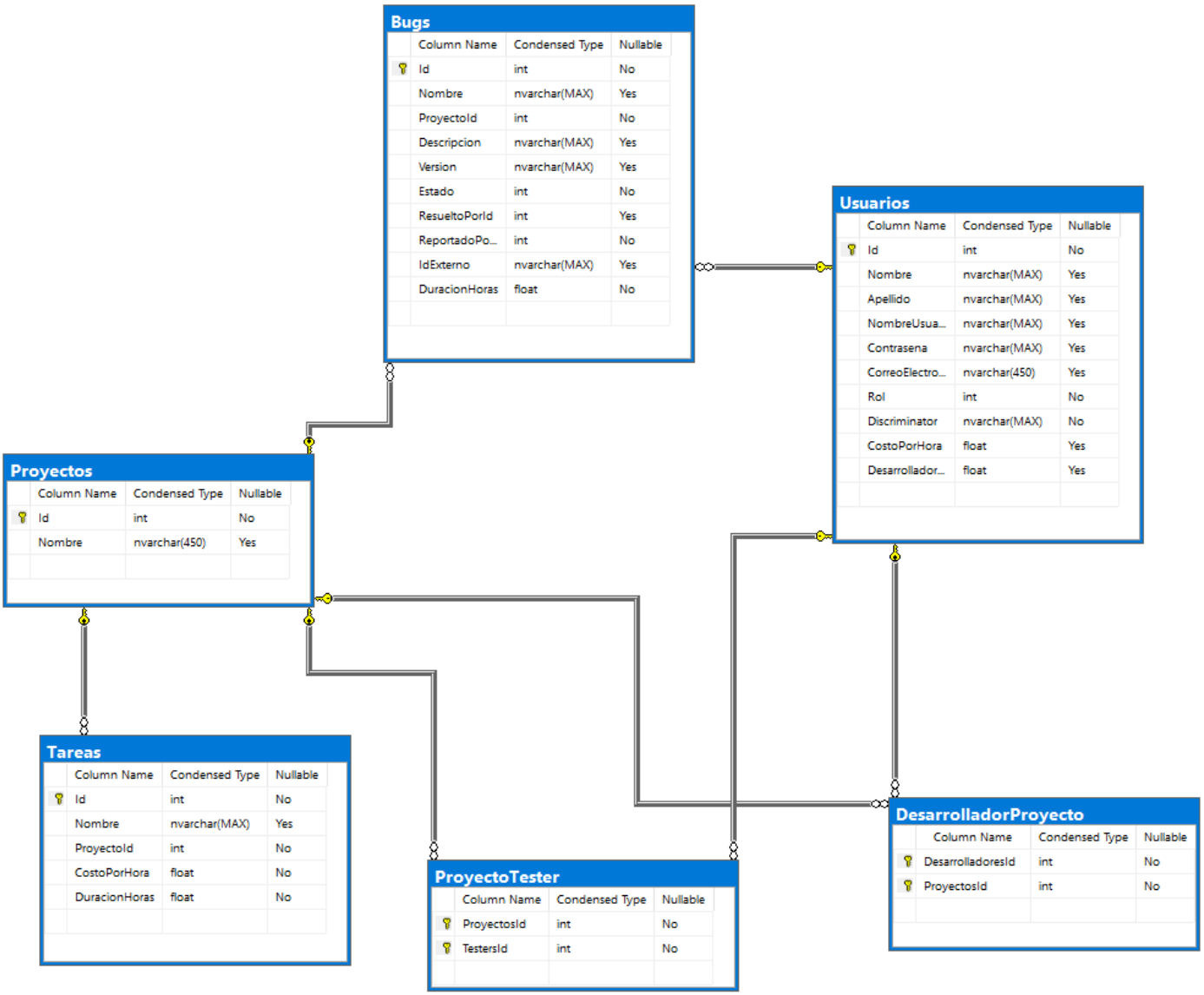
Dentro del paquete Jiru.Web también tenemos el paquete de migraciones que se decidió omitir en el diagrama de clases. También tenemos el directorio *Importaciones,* en el cual se almacenan los archivos con bugs a importar.

3.10 Jiru.Excepciones

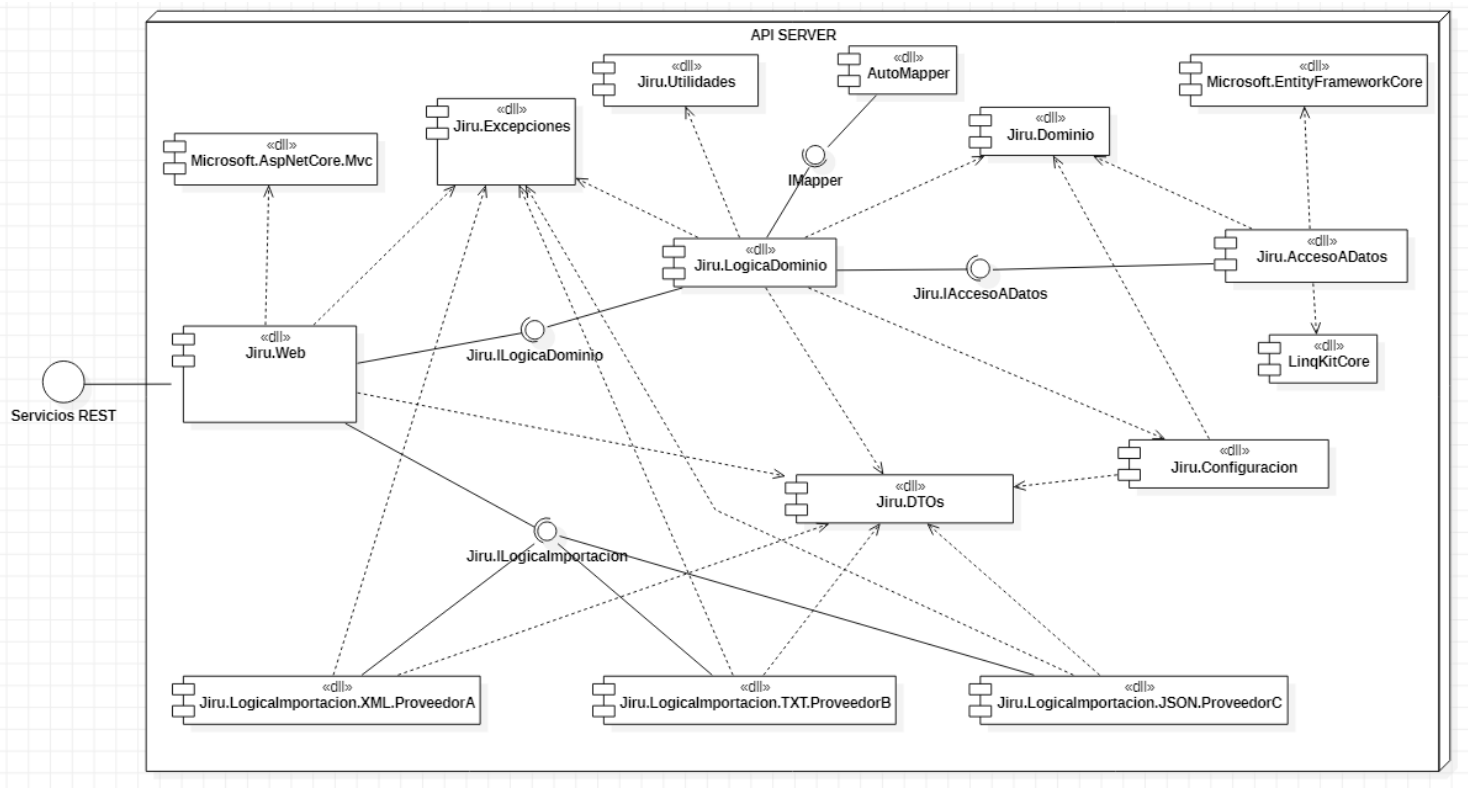
Descripción

En este paquete encontramos todas las excepciones personalizadas, que son lanzadas en el sistema. Se decidió ignorar este diagrama ya que son simplemente clases que extienden *Exception*, y para algunos casos reciben un mensaje en el constructor.

4. Diagrama de Tablas



5. Diagrama de Componentes



Descripción

El principal motivo por el cual se separó la solución en los componentes mostrados es el reusó de funcionalidades y código. Luego está el hecho de que, si el sistema está separado en diferentes componentes, es más mantenible y sencillo de analizar en caso de que algo falle. También si el día de mañana se desea reemplazar o quitar (para usar en otra solución) algún componente, se podría hacer. Todo esto evita que el sistema sea un gran monolito difícil de mantener y depurar. Otra cosa importante es que cada componente está nombrado en base a su responsabilidad, por lo que, por ejemplo, no vamos a encontrar una clase accediendo a la fuente de datos en el componente de configuración. Notar que en el diagrama también se incluyeron las librerías externas de las que depende el sistema.

6. Justificación y Análisis del Diseño

6.1 Estructura de la solución

Como se pudo ver en los diagramas, la solución está separada en diferentes proyectos, estos agrupan funcionalidades relacionadas, que, a la hora de compilarlos, se pueden hacer por separado. Esto nos permite distribuir de mejor forma la aplicación, mejorando la capacidad de reusó de los diferentes módulos.

Otra cosa para destacar es que las interfaces tienen sus propios paquetes, esto nos permite aplicar el principio de inversión de la dependencia, para que los módulos de alto y bajo nivel, solo dependan de interfaces bien definidas y no de implementaciones concretas.

Cada paquete tiene su propia responsabilidad y estás no están mezcladas entre paquetes, como se comentaba en la descripción del diagrama de componentes.

6.2 Dependencia entre paquetes

En el proyectó se utilizó el patrón de inyección de dependencias para evitar el acoplamiento entre clases y paquetes. Se hizo uso de la implementación que ofrece .NET Core, la cual permite inyectar implementaciones de interfaces previamente especificadas, en los constructores de las clases. Esto es una implementación del principio IoC (Inversion of Control), en el que el Framework tiene el control del ciclo de vida de los objetos a inyectar.

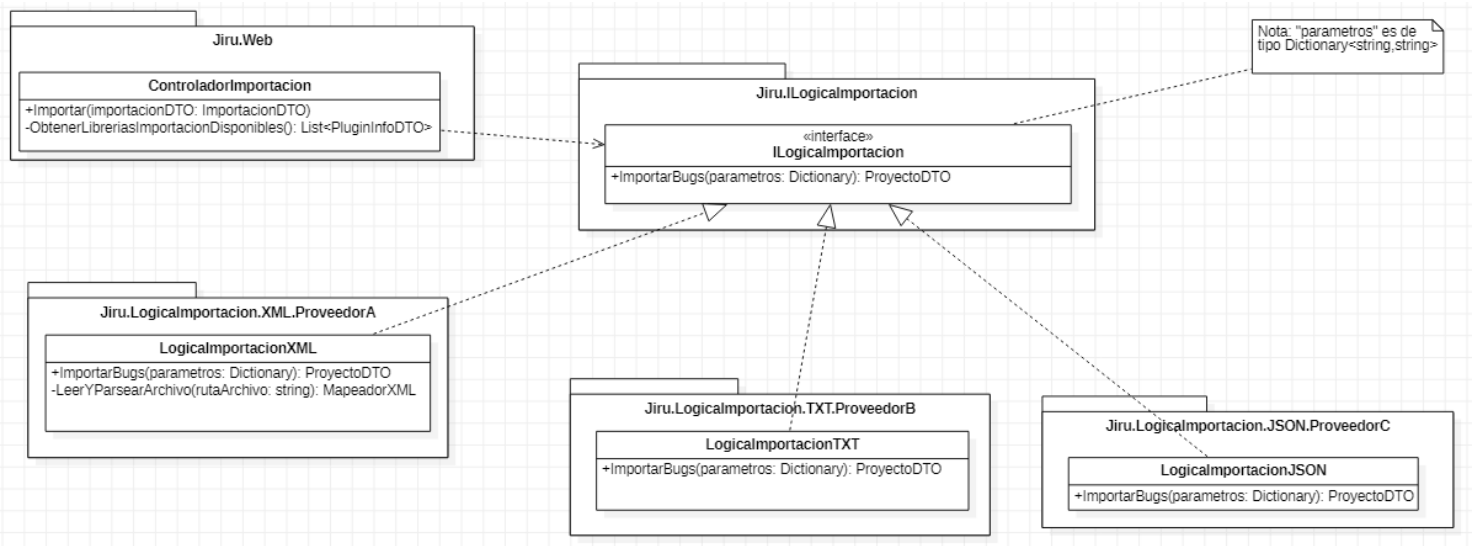
Para evitar que ocurran posibles Memory Leaks si la aplicación tiene mucha carga, se decidió que la instancia del objeto que se inyecta sea la misma que se pasa a los controladores, servicios, etc. dentro del ciclo de vida de la petición HTTP. Esto se logra utilizando el método AddScoped en vez de AddTransient / AddSingleton. Ejemplo en Startup.cs:

```
services.AddScoped<ILogicaAutenticacion, LogicaAutenticacion>();
services.AddScoped<ILogicaBug, LogicaBug>();
services.AddScoped<ILogicaProyecto, LogicaProyecto>();
services.AddScoped<ILogicaUsuario, LogicaUsuario>();

services.AddScoped<IRepositorioBug, RepositorioBug>();
services.AddScoped<IRepositorioProyecto, RepositorioProyecto>();
services.AddScoped<IRepositorioUsuario, RepositorioUsuario>();
```

6.3 Importación de bugs

Para solucionar este problema, se utilizó el patrón estrategia, el cual nos permite ofrecer varias implementaciones de la misma funcionalidad, pudiendo seleccionar una en tiempo de ejecución utilizando *Reflection*.



Esto pone sobre la mesa la capacidad de que, si en el futuro llega un nuevo proveedor con otro tipo de importación, pueda crear su propia implementación de ILogicaImportacion, luego simplemente con compilar la nueva librería y agregarla a la carpeta *C:/libreriasjiru*, la nueva implementación queda disponible para ser utilizada, sin tener que, ni siquiera, reiniciar el servidor web la API. También, como el método *importar* recibe un diccionario de valores *String* (tanto clave, como valor), se puede enviar cualquier tipo de dato en ese formato, desde rutas a archivos, cadenas de conexión, credenciales, *JSONs*, etc. Todo esto mejora la extensibilidad del sistema, y

simplifica el proceso de agregar nuevas implementaciones de importación a los desarrolladores.

6.4 Mecanismos de acceso a datos

A la hora del acceso a datos, se tuvo en cuenta la utilización del patrón repositorio. Así el acceso a datos queda encapsulado en un solo paquete, mejorando la mantenibilidad y seguridad del sistema.

De todas formas, este diseño puede ser mejorado, ya que se puede generar un repositorio base que sea abstracto y después hacer que las diferentes implementaciones del repositorio hereden todos los métodos ya implementados de las funcionalidades básicas (CRUD), para así evitar repetir código.

También si el dominio de la solución se vuelve más complejo, se podría evaluar la implementación del patrón *UnitOfWork* sobre el *DbContext* (que ya de por si cumple con este patrón), para que todos los Repositorios compartan el mismo *DbContext*, así, entre otras cosas, obtener mayor control sobre las transacciones que se comienzan y confirman al operar sobre las diferentes entidades del sistema.

6.5 Manejo de excepciones

Para este caso, se decidió crear excepciones personalizadas, las cuales permiten un manejo de errores simplificado. Luego se creó un filtro manejador de errores, el cual permite capturar excepciones de una forma global, y serializar las mismas en un formato correcto para ser devueltas al cliente que esté consumiendo la API.

Este también evalúa el ambiente en el que se está ejecutando la aplicación antes de retornar la información, si es desarrollo también retorna el StackTrace de la excepción (para mejorar la experiencia de desarrollo). Si la excepción no es una de las excepciones personalizadas, entonces la misma cae en un caso Fallback, así evitamos revelar información interna al usuario sobre el error.

6.6 Autenticación

Para la autenticación, se decidió utilizar JWT ya que es un estándar ampliamente utilizado en la industria. La lógica de generación y autenticación de los tokens se encuentra en el paquete LogicaDominio (se podría mover a su propio proyecto para agregar la capacidad de llevarse la solución completa de autenticación a otra solución de .NET Core).

También se agregó un filtro de autenticación, este evalúa si el usuario tiene acceso a la aplicación antes de ejecutar cualquier operación. En caso de que no tenga acceso, retorna el error correspondiente. Si tiene acceso, agrega el usuario al contexto de la petición HTTP para ser utilizado por la operación que se va a ejecutar.

6.7 Estructura de proyecto web

Se decidió colocar los controladores en un paquete asociado a la versión de la API, para que en un futuro si se desea agregar una nueva versión, no haya conflictos entre controladores con el mismo nombre, y puedan coexistir ambas versiones.

6.8 Análisis de métricas

Para poder obtener las diferentes métricas de diseño, utilizamos la herramienta denominada *NDepend*, la cual permite analizar el código y obtener un reporte de métricas de este. Luego de ejecutar el proceso de análisis con *NDepend* obtuvimos el siguiente reporte:

Assemblies	# lines of code	# IL instruction	# Types	# Abstract Types	# lines of comment	% Comment	% Coverage	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
Jiru.Web v1.0.0.0	857	10810	26	0	17	1.95	-	0	192	0.58	1	0	0
Jiru.LogicaDominio v1.0.0.0	273	2418	5	0	0	0	-	1	96	0.2	0.99	0	0.01
Jiru.DTOs v1.0.0.0	102	397	10	0	0	0	-	22	17	0.9	0.44	0	0.4
Jiru.Dominio v1.0.0.0	79	299	9	0	0	0	-	15	11	1.67	0.42	0	0.41
Jiru.AccesoADatos v1.0.0.0	79	1132	6	0	0	0	-	1	58	1	0.98	0	0.01
Jiru.LogicaImportacion.JSON.ProveedorC v1.0.0.0	29	166	3	0	0	0	-	0	19	1.33	1	0	0
Jiru.LogicaImportacion.XML.ProveedorA v1.0.0.0	28	170	3	0	0	0	-	0	29	1.33	1	0	0
Jiru.Configuracion v1.0.0.0	26	229	2	0	3	10.34	-	3	39	0.5	0.93	0	0.05
Jiru.LogicaImportacion.TXT.ProveedorB v1.0.0.0	17	126	1	0	0	0	-	0	15	1	1	0	0
Jiru.Excepciones v1.0.0.0	13	102	12	0	0	0	-	10	6	0.08	0.38	0	0.44
Jiru.Utilidades v1.0.0.0	4	76	3	0	7	63.64	-	1	22	1	0.96	0	0.03
Jiru.ILogicaDominio v1.0.0.0	0	0	5	5	-	-	-	13	11	0.2	0.46	1	0.32
Jiru.IImportacion v1.0.0.0	0	0	1	1	-	-	-	4	3	1	0.43	1	0.3
Jiru.IAccesoADatos v1.0.0.0	0	0	4	4	-	-	-	10	14	0.25	0.58	1	0.41

Lo primero que podemos deducir a partir de la tabla mostrada, es que se cumple el principio de abstracciones estables, ya que los paquetes más inestables son los más concretos (Ejemplo: Jiru.LogicaDominio - contiene solo clases concretas y la inestabilidad es de 0.99) y los paquetes más estables son los paquetes más abstractos (Ejemplo: Jiru.IlogicaDominio - contiene solo interfaces y la inestabilidad es de 0.46).

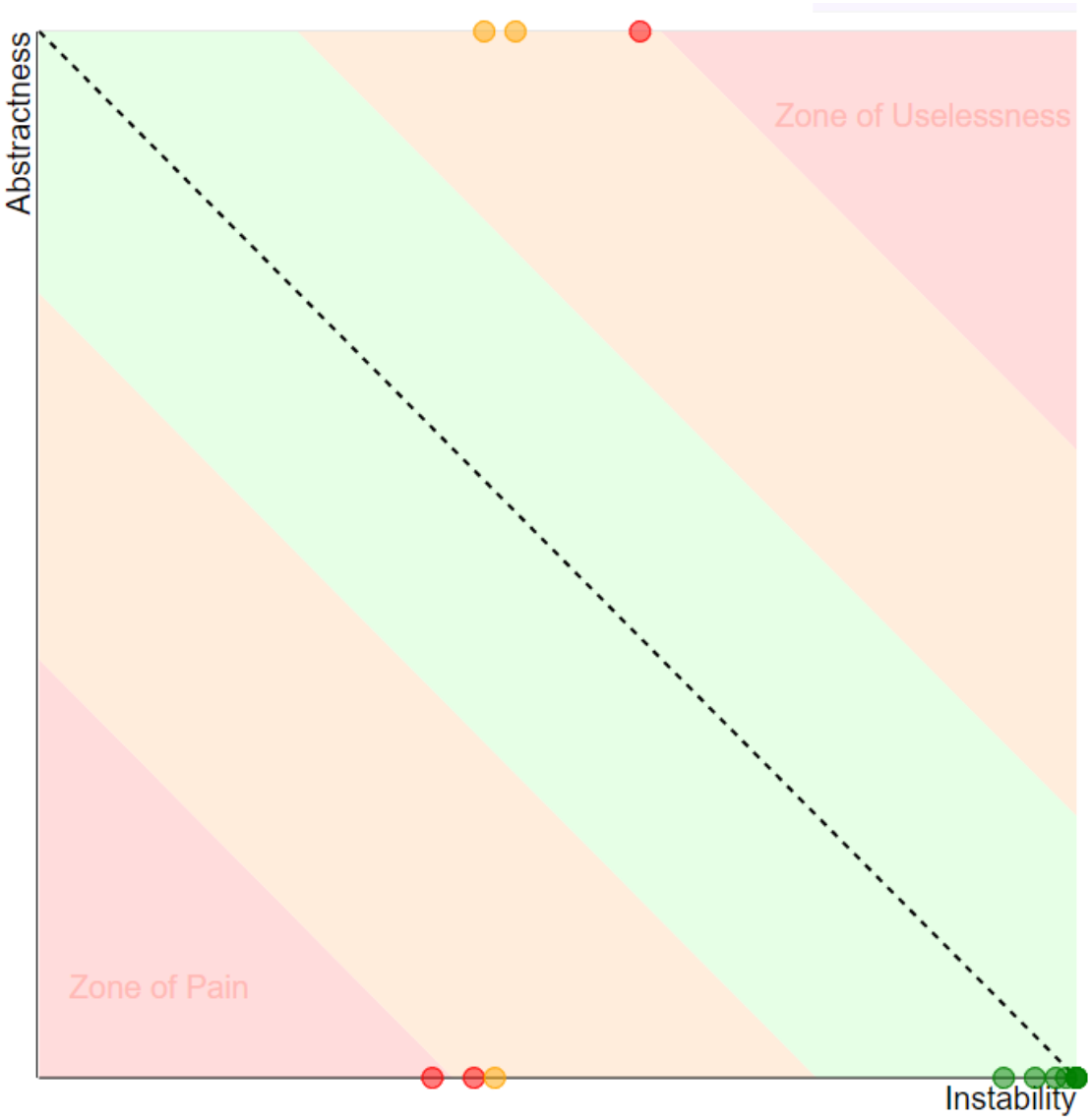
También, en la mayoría de los casos se cumple el principio de dependencias estables, el cual menciona que un paquete debería depender de paquetes que son más estables que él. Si tomamos el ejemplo de Jiru.LogicaDominio, podemos ver que este tiene I=0.99, pero depende de paquetes que tienen I=0.44 (Jiru.DTOs), I=0.46 (Jiru.ILogicaDominio), I=0.38 (Jiru.Excepciones), etc.

Otro principio que se llega a cumplir es el principio de reúso común, esto no se llega a notar del todo bien en las métricas, pero si analizamos podemos ver que el valor del *Afferent Coupling* (Acoplamiento aferente en español) es más alto que el *Efferent*

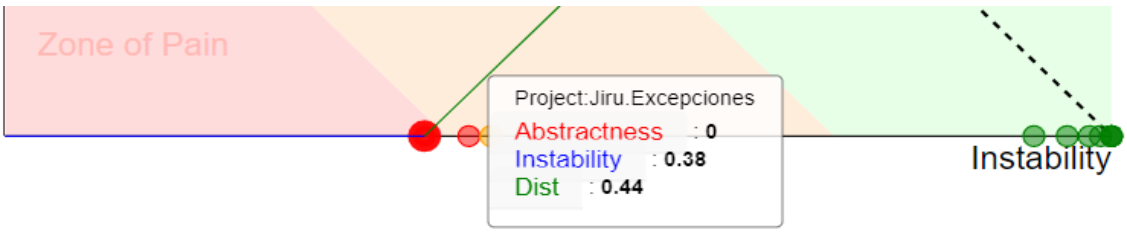
Coupling (Acoplamiento eferente en español*)* en algunos paquetes como Jiru.Excepciones, Jiru.DTOs y Jiru.Dominio. Esto nos puede llegar a indicar que esos paquetes están siendo reutilizados en muchas partes del sistema, ya que muchos paquetes dependen de ellos.

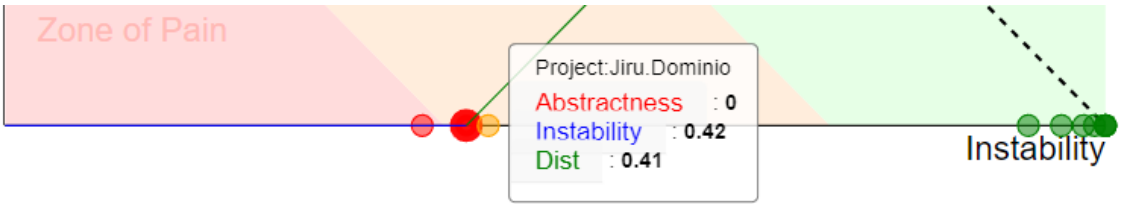
Luego si vemos el caso de la cohesión relacional, podemos ver que muchos paquetes tienen un valor inferior a 1.5, haciendo que no se cumpla el principio de clausura común. Esto se debe a que a la hora de decidir como se iba a separar los componentes del sistema, se tomó la decisión de separarlos por responsabilidad y no por funcionalidad, por ejemplo, todas las clases de la lógica están el paquete de lógica, todas las clases del dominio están el paquete del dominio, todas las excepciones están en el paquete de excepciones, etc. Si hubiéramos creado los componentes en base a funcionalidades, por ejemplo, todo el manejo de bugs en un paquete, todas las clases de ese paquete estarían fuertemente relacionadas, por lo que la cohesión relacional seria más alta, pero quizás otros principios como el de rehúso común se verían afectados de forma negativa.

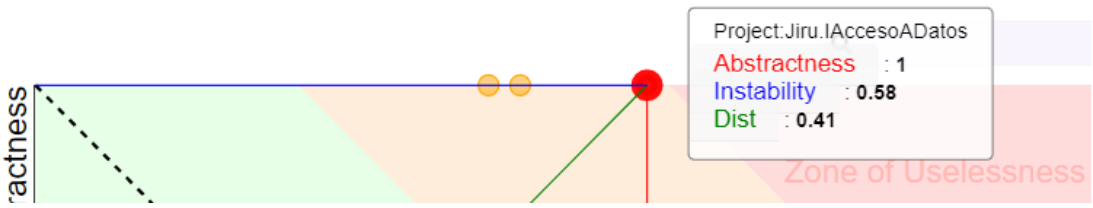
Otro análisis que podemos hacer a partir de las métricas de diseños es el de la distancia y la relación que hay entre inestabilidad y abstracción. Para esto podemos utilizar el grafico siguiente:



Como se puede ver, hay tres paquetes en zona roja, estos son:







Para los casos de los paquetes que están en la zona de dolor, se tendría que evaluar la posibilidad de agregar abstracciones, por ejemplo, que en el paquete Jiru.Dominio, la clase Usuario sea abstracta y así con alguna otra clase. Para el caso de excepciones, esto es más complicado ya que el paquete simplemente contiene excepciones que heredan de Exception. Agregando estas abstracciones, nos ayudaría a mejorar la extensibilidad del paquete y evitar que se produzcan errores en otros paquetes cuando las clases del paquete descrito cambian.

En el caso del paquete que se encuentran en la zona de poca utilidad, se podría ver de mover las interfaces que se encuentran en el a otros paquetes, ya que pocos paquetes dependen de él. Por ejemplo, las interfaces de Jiru.IAccesoADatos, se podrían mover al paquete Jiru.AccesoADatos, y así contener todas las abstracciones y lógica de acceso a datos en un solo paquete.

Luego, si vemos los otros paquetes vemos que la mayoría se encuentran a una distancia prudente de la secuencia principal, incluso habiendo varios encima de esta. Esto nos indica que el balance entre estabilidad y abstracción es bueno, por lo que el diseño de esto se podría llegar a decir que también es bueno.

6.9 Mejoras del diseño

Para esta entrega el principal punto que se cambió a nivel diseño fue la importación de bugs, si bien en la primera entrega se había utilizado el patrón estrategia con *Reflection* para la funcionalidad de importar bugs, en esta entrega nueva se mejoró el hecho de que sea lo más extensible posible, por lo que se quitó la referencia a la lógica de dominio y ahora el método ImportarBugs recibe un Diccionario de Strings, lo que hace posible enviar cualquier tipo de parámetros hacia las librerías externas.

Luego simplemente se agregaron las nuevas funcionalidades requeridas, como el manejo de tareas, que para esto simplemente con crear su entidad, el DTO, su propio repositorio, su interfaz de repositorio, su clase de lógica de dominio, una interfaz de la lógica de dominio y un controlador de la WebApi quedó funcionando correctamente (obviamente se tuvieron que implementar los métodos correspondientes – crear y obtener). Después se agregaron los campos nuevos a los modelos y la lógica correspondiente, como por ejemplo el costo por hora y la duración.

También algunos métodos de clases que fueron creados en la primera entrega cambiaron los parámetros que reciben, para poder simplificar la lógica que ellos contienen.

7. Despliegue de la Aplicación

Para facilitar el proceso de despliegue de la aplicación se decidió agregar soporte para que pueda ser ejecutada en contenedores Linux de Docker, lo cual hoy en día es una de las formas más utilizadas en la industria, ya que brinda mucha flexibilidad a la hora de manejar los recursos y escalar aplicación. Muchos proveedores de servicios en la nube, como por ejemplo AWS, GCP, Azure, ofrecen orquestadores de contenedores, como ECS, Kubernetes, etc, que nos permiten desplegar, escalar y monitorizar nuestra aplicación de forma rápida y resiliente.

En nuestro caso, como se va a hacer una muestra de despliegue de la aplicación localmente, se utilizará Docker Compose como orquestador, y se levantaran tres contenedores diferentes, uno para el servidor de base de datos, otro para el backend y otro para el frontend. Por lo que, si se desea levantar los contenedores de la aplicación utilizando Docker Compose, se deben seguir los siguientes pasos (son los mismos tanto para el front-end como el back-end).

1. Asegurarse de que Docker se encuentra instalado en el sistema y se encuentra actualizado.
2. Abrir una terminal y pararse en la raíz de cada proyecto (el archivo *docker-compose.yml* debe estar en ese directorio).
3. Ejecutar: **docker compose build** desde una terminal.
4. Esperar que termine de construir los contenedores.
5. Cuando termina, estos contenedores estarán listos para ser desplegados al orquestador de preferencia.
6. Si se desea levantar localmente, ejecutar: **docker compose up**
7. Si se desplego localmente, podremos acceder a:
    a. Al servidor de base de datos con los siguientes datos:
        i. Host: 127.0.0.1
        ii. Puerto: 14333
        iii. Usuario: sa

iv. Contraseña: Jiru123_Pass

b. Al servidor de la API:
    i. URL: http://localhost:8000

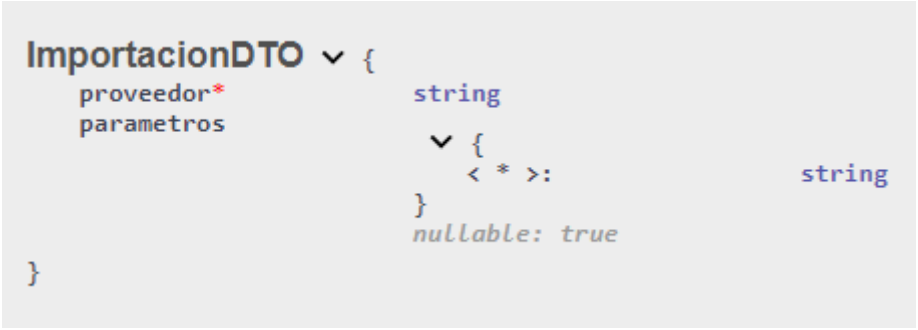
c. Al servidor del Front-end:
    i. URL: http://localhost:8080

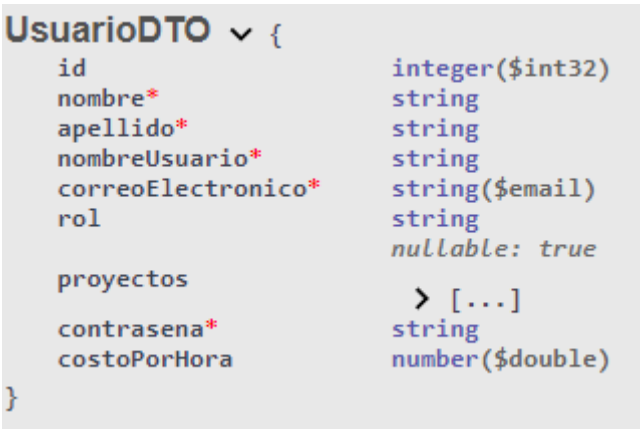
(Notar que se muestran los usuarios y contraseña ya que simplemente son para realizar una prueba local, en el caso de desplegar el contenedor se debería optar por un servicio de base de datos *managed* como AWS RDS, y no levantar la base de datos en un contenedor, ya que se puede llegar a tener complicaciones)

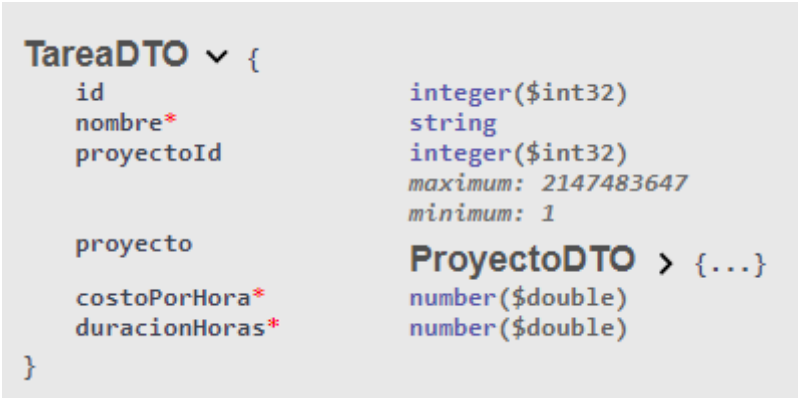
8. Anexo

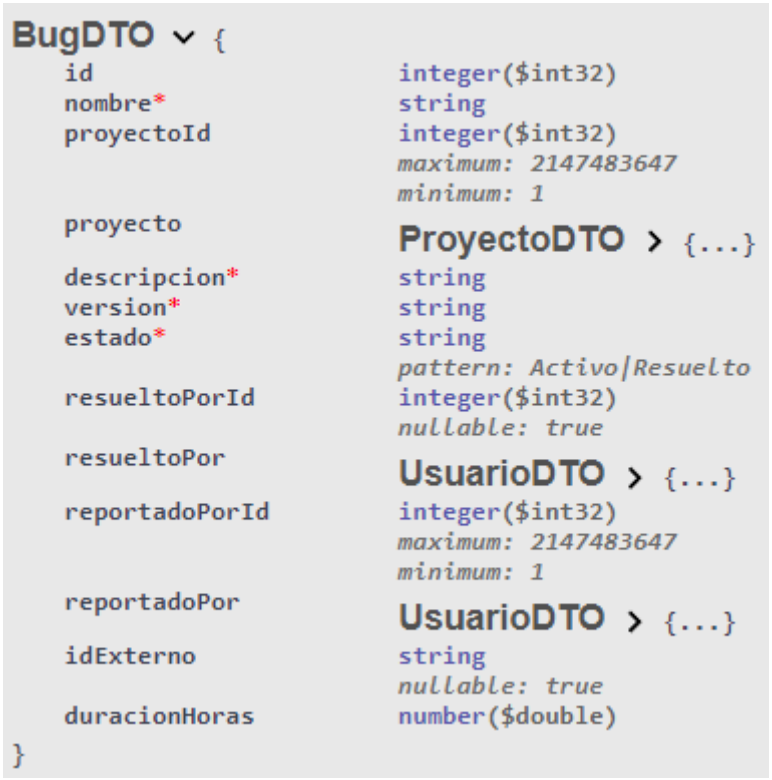
8.1. Especificación de la API

8.1.1. Esquemas nuevos y modificados



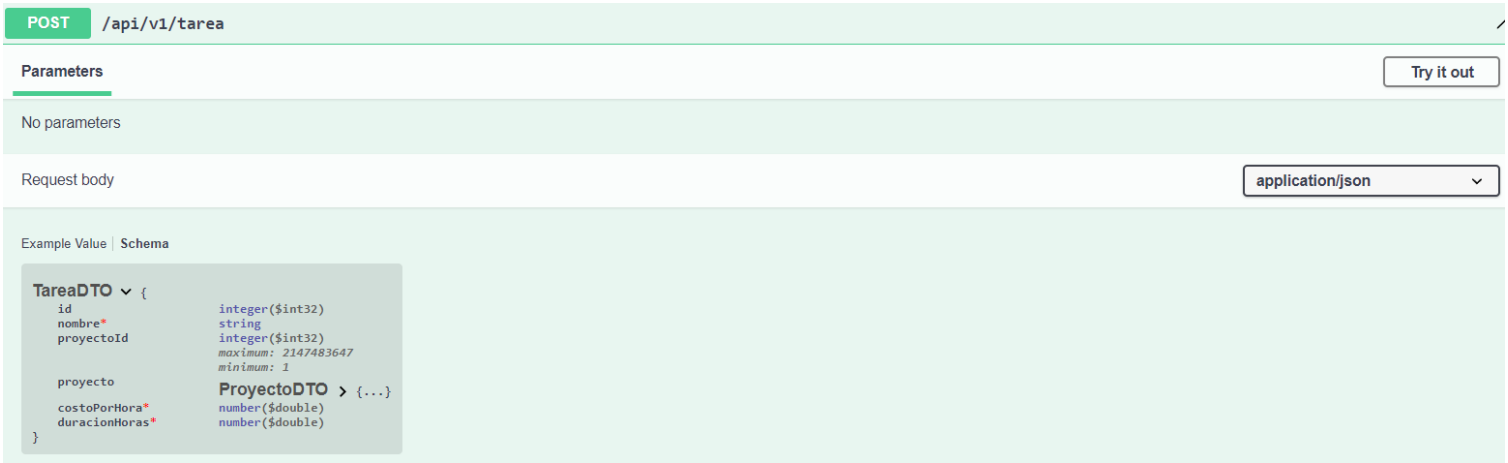






8.1.2. Recursos nuevos

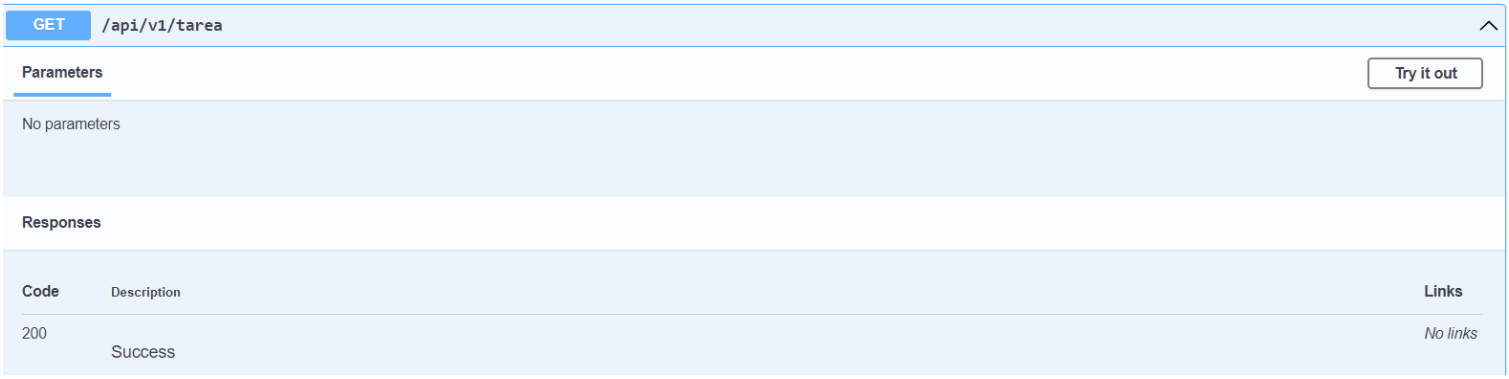
8.1.2.1. Crear Tarea



Respuestas:

- 201 – Se creó la tarea exitosamente.
- 403 – No tienes permiso para realizar esta operación.
- 404 – No existe el proyecto al que se desea asignar la tarea.

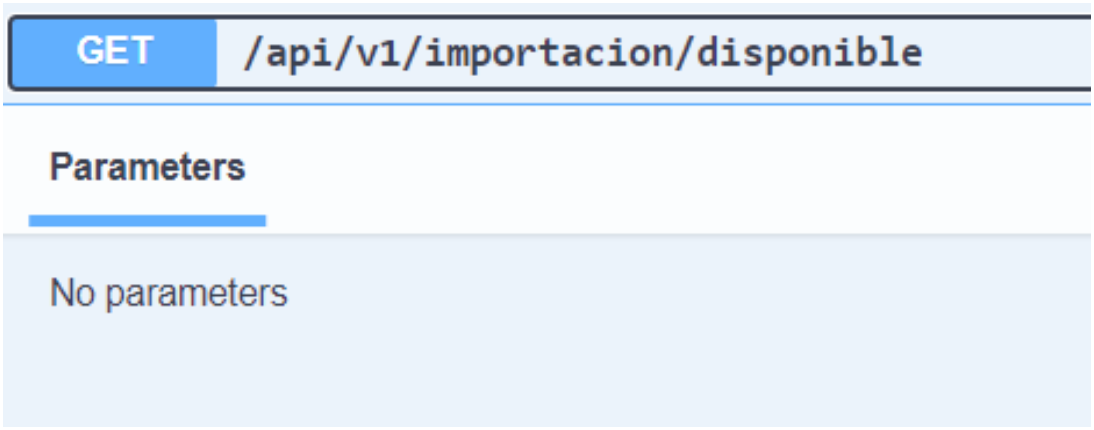
8.1.2.2. Obtener Tareas



Respuestas:

- 200 - Se obtuvieron las tareas exitosamente.

8.1.2.3. Obtener librerías de importación disponibles



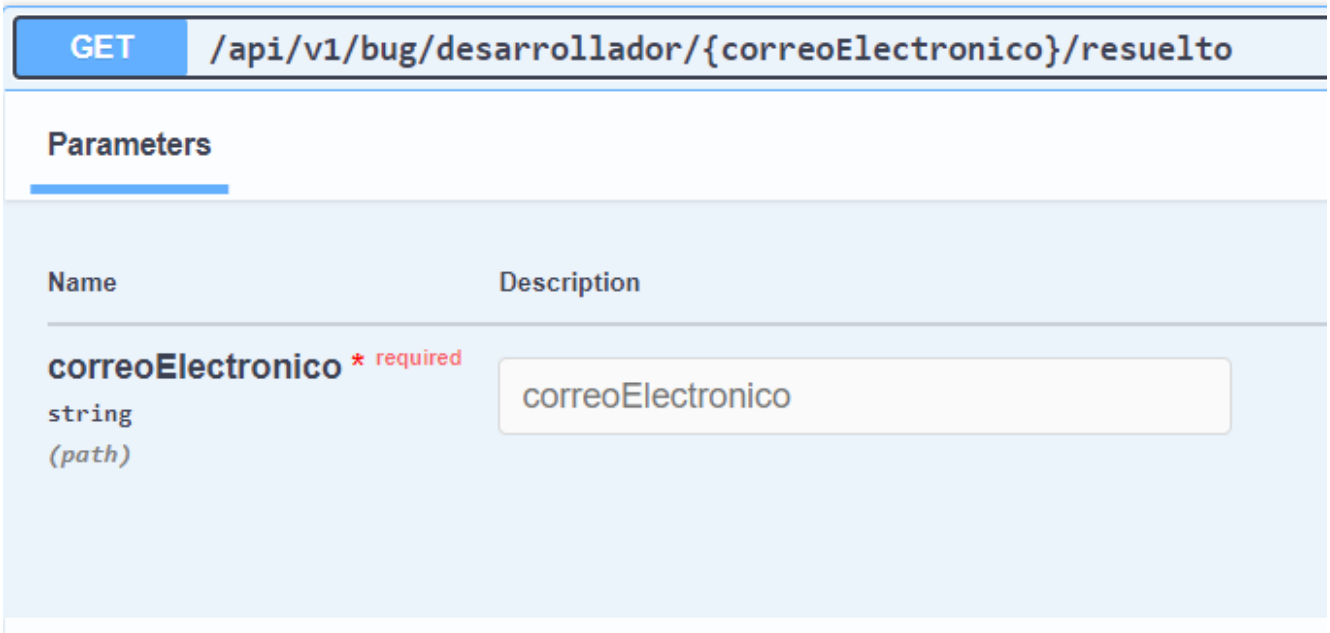
Respuestas:

- 200 - Se obtuvieron los plugin exitosamente.

8.1.3. Recursos modificados

8.1.3.1 - Obtener cantidad de bugs resueltos por desarrollador

Este endpoint anteriormente recibía el id del usuario, pero ahora recibe el correo electrónico.

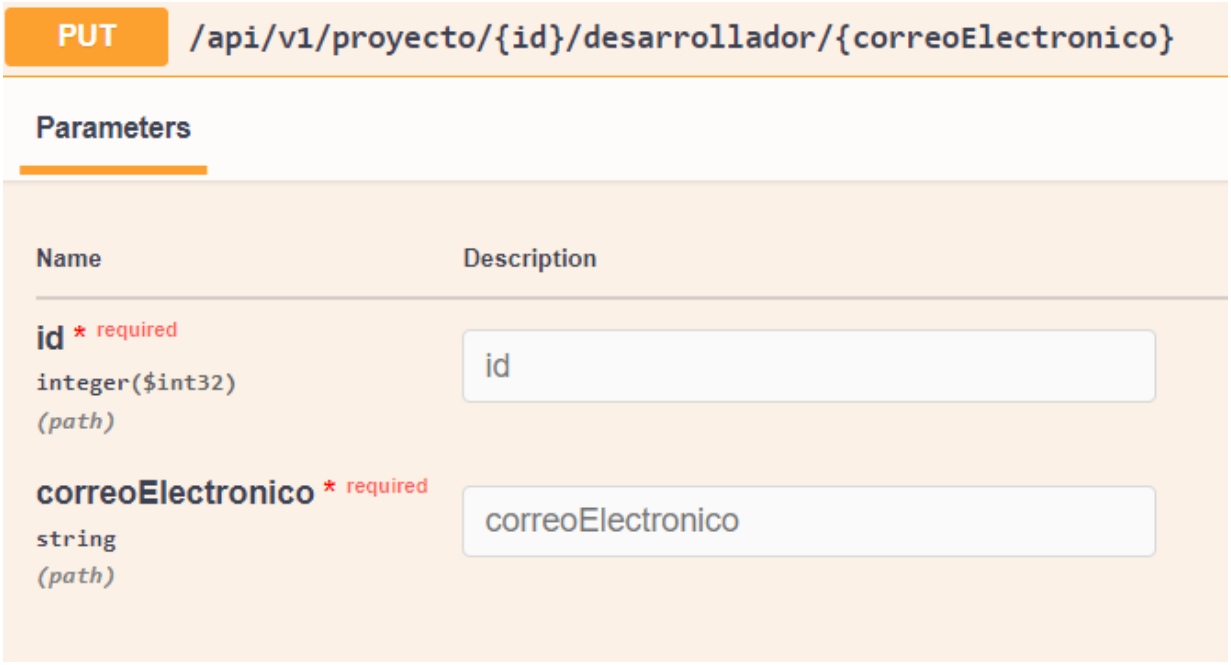


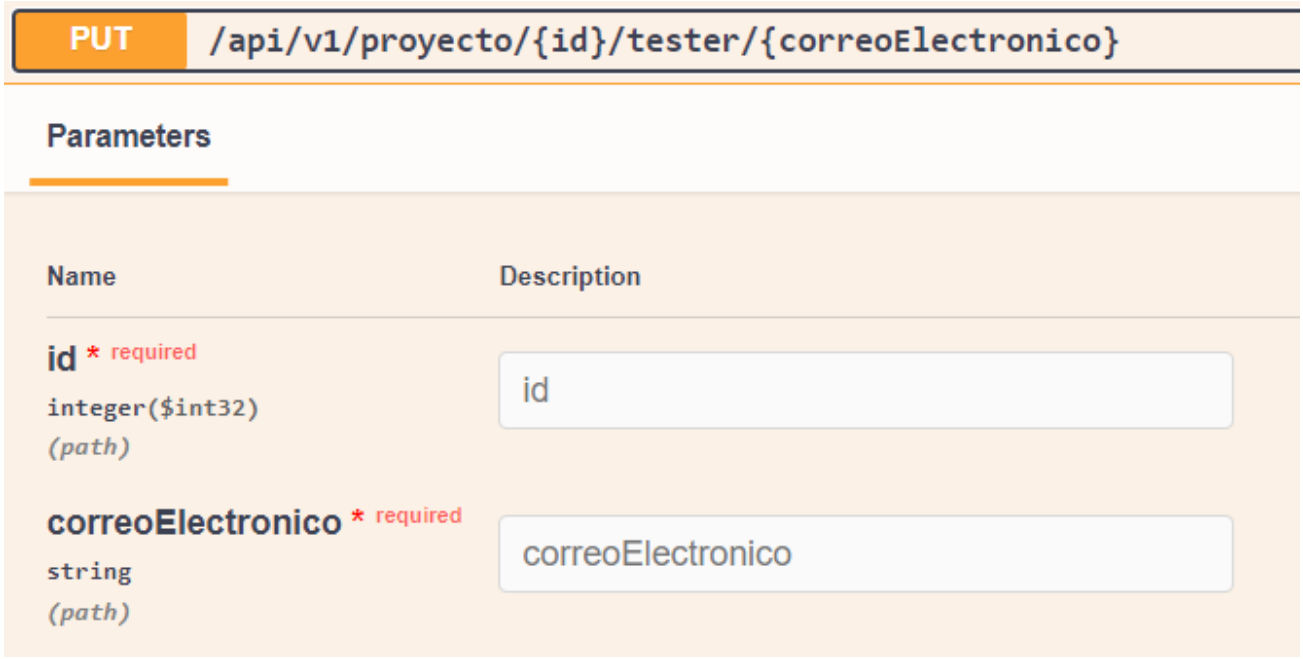
Respuestas:

- 200 – Se obtuvo la cantidad de bugs exitosamente.
- 403 – No tienes permiso para realizar esta operación.

8.1.3.2 – Asignar desarrollador/tester a proyecto

Anteriormente este recurso recibía el id del proyecto y el id del desarrollador, pero ahora recibe el id del proyecto y el correo electrónico del desarrollador.





Respuestas:

- 200 – Se asigno el usuario exitosamente.
- 400 – El rol del usuario no coincide.
- 403 – No tienes permiso para realizar esta operación.
- 404 – El proyecto o el usuario no existen.

8.1.3.3 – Desasignar desarrollador/tester a proyecto

Anteriormente este recurso recibía el id del proyecto y el id del desarrollador, pero ahora recibe el id del proyecto y el correo electrónico del desarrollador.

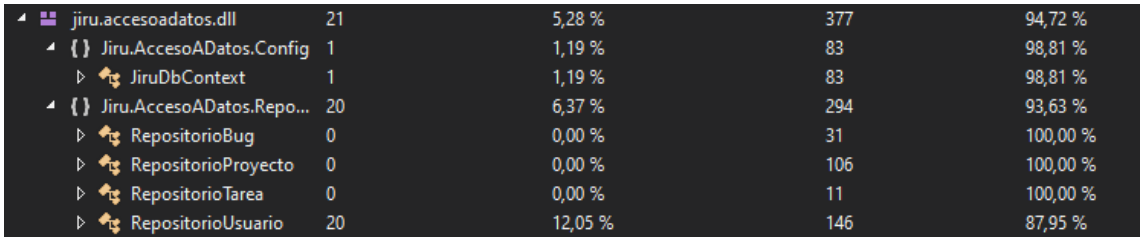
Respuestas:

- 200 – Se desasigno el usuario exitosamente.
- 400 – El rol del usuario no coincide.
- 403 – No tienes permiso para realizar esta operación.
- 404 – El proyecto o el usuario no existen.

8.2 – Informe de las pruebas

En nuestra solución la estrategia de TDD que aplicamos fue *outside - in*, porque comenzamos por un test de aceptación que falle, lo que nos guiaba al bucle interno. En el bucle interno (Red - Green - Refactor) implementamos la lógica de nuestra solución.
Por último, realizamos las interacciones necesarias en este bucle interno hasta conseguir pasar el test de aceptación.

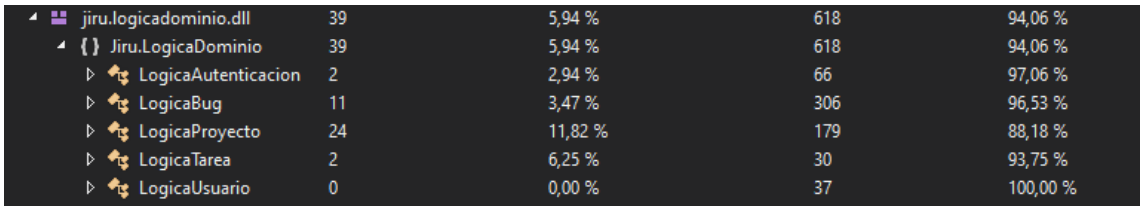
8.2.1 - Prueba de Acceso a datos



jiru.accesoadatos.dll	21	5,28 %	377	94,72 %
Jiru.AccesoADatos.Config	1	1,19 %	83	98,81 %
JiruDbContext	1	1,19 %	83	98,81 %
Jiru.AccesoADatos.Repo...	20	6,37 %	294	93,63 %
RepositorioBug	0	0,00 %	31	100,00 %
RepositorioProyecto	0	0,00 %	106	100,00 %
RepositorioTarea	0	0,00 %	11	100,00 %
RepositorioUsuario	20	12,05 %	146	87,95 %

En el paquete *Jiru.AccesoADatos* tienen un 95% de cobertura, es donde encontramos el código necesario para interactuar con la fuente de datos. Tener este porcentaje de cobertura nos garantiza una alta calidad en el acceso a los datos.

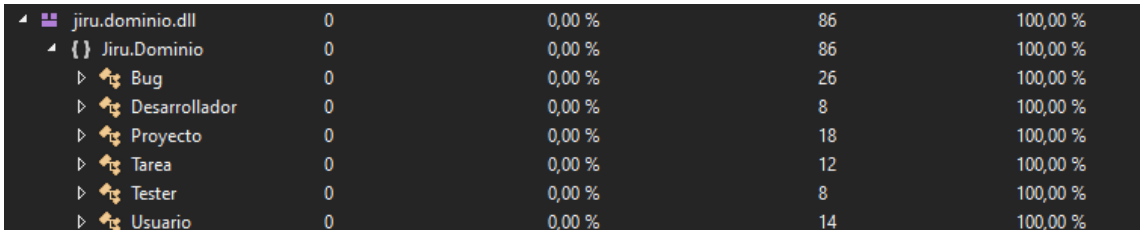
8.2.2 - Pruebas Lógica del dominio



jiru.logicadominio.dll	39	5,94 %	618	94,06 %
Jiru.LogicaDominio	39	5,94 %	618	94,06 %
LogicaAutenticacion	2	2,94 %	66	97,06 %
LogicaBug	11	3,47 %	306	96,53 %
LogicaProyecto	24	11,82 %	179	88,18 %
LogicaTarea	2	6,25 %	30	93,75 %
LogicaUsuario	0	0,00 %	37	100,00 %

En el paquete *Jiru.LogicaDominio* tienen un 94% de cobertura, es donde encontramos la lógica de la aplicación. En este paquete se implementan las diferentes operaciones que se pueden realizar con las entidades del sistema, ya sea operaciones básicas (CRUD), como también operaciones de autenticación y asignación. Nos parece de suma importancia tener niveles tan altos de cobertura de código en clases tan importantes de la solución, de esta manera garantizar una alta calidad.

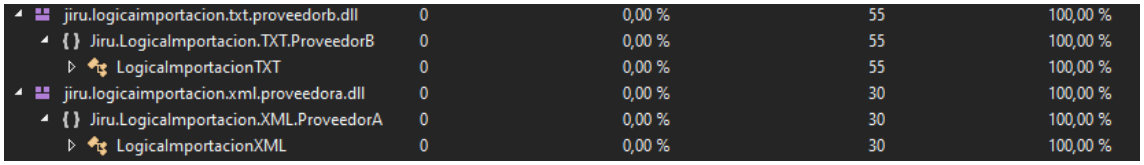
8.2.3 - Pruebas de dominio



jiru.dominio.dll	0	0,00 %	86	100,00 %
Jiru.Dominio	0	0,00 %	86	100,00 %
Bug	0	0,00 %	26	100,00 %
Desarrollador	0	0,00 %	8	100,00 %
Proyecto	0	0,00 %	18	100,00 %
Tarea	0	0,00 %	12	100,00 %
Tester	0	0,00 %	8	100,00 %
Usuario	0	0,00 %	14	100,00 %

El paquete *Jiru.Dominio* tienen un 100% de cobertura y es quien contiene la definición de las diferentes entidades del sistema. Tener este porcentaje de cobertura nos garantiza una alta calidad en el acceso a los datos.

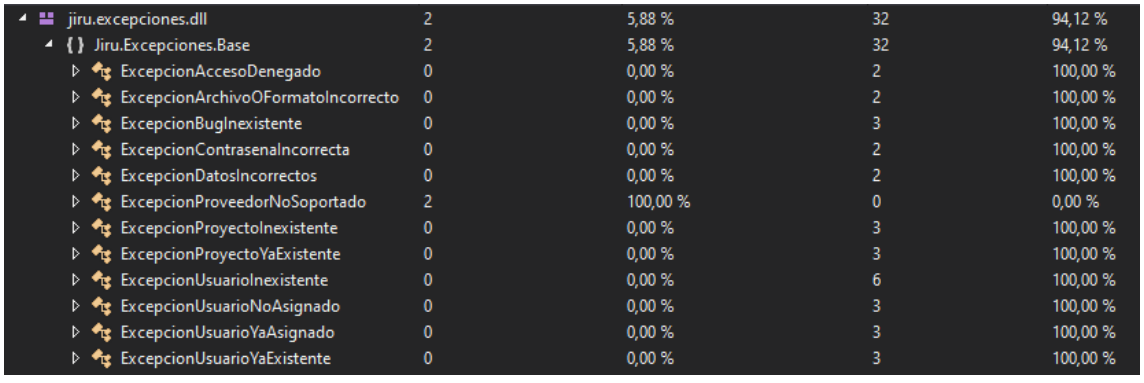
8.2.4 - Pruebas de lógica de importaciones



jiru.logicaimportacion.txt.proveedorb.dll	0	0,00 %	55	100,00 %
Jiru.LogicaImportacion.TXT.ProveedorB	0	0,00 %	55	100,00 %
LogicaImportacionTXT	0	0,00 %	55	100,00 %
jiru.logicaimportacion.xml.proveedora.dll	0	0,00 %	30	100,00 %
Jiru.LogicaImportacion.XML.ProveedorA	0	0,00 %	30	100,00 %
LogicaImportacionXML	0	0,00 %	30	100,00 %

Los paquetes la lógica de importaciones tienen un 100% de cobertura, en estas clases es donde se implementan todos los métodos de la lógica de las importaciones tanto de los archivos TXT como los XML. Tener este porcentaje de cobertura nos garantiza una alta calidad en el acceso a los datos.

8.2.5 - Pruebas de excepciones



jiru.excepciones.dll	2	5,88 %	32	94,12 %
Jiru.Excepciones.Base	2	5,88 %	32	94,12 %
ExcepcionAccesoDenegado	0	0,00 %	2	100,00 %
ExcepcionArchivoOFormatoIncorrecto	0	0,00 %	2	100,00 %
ExcepcionBugInexistente	0	0,00 %	3	100,00 %
ExcepcionContrasenaIncorrecta	0	0,00 %	2	100,00 %
ExcepcionDatosIncorrectos	0	0,00 %	2	100,00 %
ExcepcionProveedorNoSoportado	2	100,00 %	0	0,00 %
ExcepcionProyectoInexistente	0	0,00 %	3	100,00 %
ExcepcionProyectoYaExistente	0	0,00 %	3	100,00 %
ExcepcionUsuarioInexistente	0	0,00 %	6	100,00 %
ExcepcionUsuarioNoAsignado	0	0,00 %	3	100,00 %
ExcepcionUsuarioYaAsignado	0	0,00 %	3	100,00 %
ExcepcionUsuarioYaExistente	0	0,00 %	3	100,00 %

El paquete *Jiru.Excepciones* tienen un 94% de cobertura, en este paquete encontramos todas las excepciones personalizadas, que son lanzadas en el sistema. El nivel de cobertura no es mayor porque no supimos implementar el test de la excepción de proveedor no soportado, pero el resto de las excepciones están al 100% de cobertura.