Universidad ORT Uruguay

Facultad de Ingeniería

Obligatorio 2

Entregado como requisito para la aprobación del curso - Diseño de aplicaciones 2

Javier Olveyra - 231819

Juan Pedro Puig - 281088

Grupo Matutino M6A

Tutores: Francisco Bouza, Juan Irabedra, Santiago Tonarelli

2024

Link al repositorio de github en la organización: <https://github.com/IngSoft-DA2-2023-2/231819_281088>

Índice

[**Descripción general 4**](#_ng771qsksz6i)

[**Correcciones respecto al primer obligatorio 4**](#_1q21uugft0gz)

[**Errores conocidos 5**](#_89ayxwt59ska)

[**Oportunidades de mejora 5**](#_jxda91r1wvu)

[**Vista de implementación 6**](#_5k1ig3alxuk0)

[Diagrama general de paquetes 6](#_s556cx6yrdp9)

[Diagrama de implementación 7](#_rukmkrdxgijo)

[**Vista lógica 8**](#_yuc0im9j3iai)

[ManagementApi 8](#_a9b2vi4e6dqr)

[WebModels 9](#_t4byrwlc7kcb)

[ServiceFactory 9](#_ccfuw4j2jh9d)

[LogicInterfaces 10](#_mco5clfxhdv0)

[BusinessLogic 11](#_oddhh9a4lira)

[Domain 12](#_639lwp65vqm)

[CustomExceptions 13](#_ezi9vc9aht0n)

[RepositoryInterfaces 13](#_ng7r3ooja03g)

[DataAccess 14](#_8ejmuuxs7m7l)

[**Vista de procesos 15**](#_saf8zkgaw989)

[Diagramas de interacción 15](#_bixq8mingz7a)

[**Modelo de tablas de la estructura de la base de datos 15**](#_m8jfj2anjxyo)

[**Jerarquía de herencias 17**](#_jbeirra9wl2f)

[**Inyección de dependencias 17**](#_pomtnd9cfh3n)

[**Patrones 17**](#_gqevbu47f8yg)

[**Métricas de diseño 18**](#_q1h4ww67l39b)

[**Principios de diseño 21**](#_rzhyrtyj9ta8)

[SOLID 21](#_o8tgpe5dbgqv)

[GRASP 22](#_ko9eb0s2yhc0)

[**Decisiones propias de diseño 22**](#_awdgkgw9mxvo)

[**Supuestos 22**](#_k8ou3ya9xhjz)

[**Mecanismos utilizados para permitir la extensibilidad solicitada en la funcionalidad de este nuevo obligatorio 23**](#_5qya6i173s2)

[Instructivo de importación de edificios mediante una dll: 25](#_fgib8a7v9oxq)

[**Clean code 27**](#_e10g3u84h5xz)

[**TDD 27**](#_xxhsqjir43qf)

[**Reporte de herramienta de cobertura y análisis del resultado 28**](#_1xx7oz9if0o)

[**GitFlow 30**](#_ecdzh6mpkq0u)

[**Especificación de la API 31**](#_fm21x631ylvg)

[**Manejo de excepciones 31**](#_133fumkwc2gn)

[**Mecanismo de acceso a datos 32**](#_frpsdhbk4nq2)

[**Anexo 33**](#_1da26mzhwncp)

[Diagrama de WebModels 33](#_mt3ggt87m7me)

[Diagrama de BusinessLogic 34](#_peos126dxhkt)

[Diagrama de DataAccess 35](#_gvylicu5kq70)

[Diagrama de secuencia de GetReport 36](#_hl5s5gz0y9bh)

[Diagrama de secuencia de UpdateInvitationStatus 37](#_rlqvihndvut1)

[Diagrama de secuencia de CreateConstructorCompanyBuilding 39](#_1spsenlfwziy)

# Descripción general

BuildingBoss es una aplicación web que cuenta con arquitectura cliente-servidor en la que los usuarios pueden utilizar el front-end, implementado con Angular. Este se comunica con la WebAPI implementada en la entrega anterior, y ajustada en esta entrega para admitir las nuevas funcionalidades, para realizar las operaciones elegidas por el usuario.

El objetivo de la solución es permitir alta, baja y modificación de edificios, alta, baja y modificación de administradores, administradores de empresas constructoras, encargados y empleados de mantenimiento, asignación de tareas de mantenimiento para apartamentos a empleados y generación de reportes.

Existen cuatro roles de usuarios en el sistema: Administrador, Administrador de empresa constructora, Gerente (Manager) y Empleado de mantenimiento. Cada uno tiene diferentes funcionalidades.

El administrador puede crear otros usuarios administradores (todos con las mismas funcionalidades), crear categorías para las solicitudes de los apartamentos o invitar a nuevos usuarios que no se encuentren en el sistema (se dicta a partir del email, no pueden existir dos usuarios con el mismo email). Las invitaciones pueden ser aceptadas o rechazadas. Si se rechazan o se vencen, el administrador puede borrarlas. Si se aceptan se crea un nuevo usuario con los datos de la invitación, el rol de manager o administrador de empresa constructora dependiendo el rol elegido al momento de crear la invitación y una contraseña predeterminada.

El administrador de empresas constructora puede crear o unirse a una empresa constructora y modificar su nombre. También puede crear, listar y modificar edificios que pasarán a estar asociados a la empresa constructora a la que dicho administrador está asociado. Además, puede extender la aplicación importando archivos que a través de un importador elegido cree los edificios, apartamentos, managers y dueños de apartamentos especificados en dicho archivo. Para mayor control, el administrador puede utilizar los importadores ya ingresados (json y xml) o también ingresar un importador al sistema que tome el formato elegido y mapee dichos datos a los DTOs provistos por la aplicación.

En cuanto al manager, para esta nueva versión ya no puede crear edificios, pero sí modificar sus gastos comunes y los empleados de mantenimiento asociados al mismo. Además puede crear empleados de mantenimiento y solicitudes para los apartamentos de los edificios a los que es asignado por un administrador de empresa constructora. Puede editar los apartamentos y sus solicitudes de sus edificios, y puede asignar solicitudes a los empleados de mantenimiento. También puede visualizar un reporte de los estados de sus solicitudes por apartamento o empleado de mantenimiento.

Por último, el empleado de mantenimiento puede ver las solicitudes que son asignadas por un manager. El empleado puede cambiar el estado de una solicitud entre pendiente, en progreso o completado.

# Correcciones respecto al primer obligatorio

La corrección más significativa que realizamos respecto a la entrega anterior fue la implementación de una clase Report que simplifica el algoritmo de creación de reportes y permite mayor legibilidad al momento de ver el código.

# Errores conocidos

Durante las pruebas y revisiones de código que se hicieron del sistema, se encontró el siguiente error:

Cuando se elimina un Flat y no quedan Flats en el sistema cuyo dueño sea el dueño del Flat que se eliminó, el dueño no se elimina de la tabla People, por lo que ésta crece indiscriminadamente. Consideramos que no es un error muy grave ya que se podría querer que se almacene los datos para algún uso futuro, pero para esto como quedan sin referencia en la base de datos se debería poder buscar por dueño de Flat o por mail entre los mismos. Si se quisiera eliminar los dueños, lo que habría que hacer sería recorrer todos los Flats cada vez que se elimina un Flat para verificar que el dueño del Flat no esté asociado a otro Flat y poder eliminarlo.

En general, no es un problema que no afecta los propósitos principales del sistema, que no fue solucionado por falta de tiempo y por priorizar hacer una buena documentación y pruebas sobre el sistema.

# Oportunidades de mejora

Como marcamos en la entrega anterior, una oportunidad de mejora se encuentra en el momento de crear un flat. Como los flats son creados por el edificio, estos se crean con un dueño con campos vacíos. Esto genera que la tabla People se llene de dueños vacíos, que sólo obtendrán datos cuando se actualice su información. Igualmente se borra al momento de eliminar el building o modificar el dueño de un flat. Pero ocupan espacio sin sentido en la base de datos mientras no son modificados ni eliminados. Una forma de mejorar dicho problema sería que la propiedad de un flat que refiere al dueño pueda ser nula, asignándole o creando uno al momento de modificación.

Otra oportunidad de mejora respecta al diseño de clases. Al finalizar la implementación de la solución para esta nueva entrega nos dimos cuenta que la lógica de los edificios dentro del paquete BusinessLogic quedó extensa de más. Una buena solución sería separar ciertos métodos de la misma en una lógica de flats y otra de edificios, y que la de edificios conozca a la de flats.

Otra oportunidad de mejora es tener la posibilidad de eliminar los importadores de edificios registrados en el sistema. En esta entrega no se pueden eliminar, por lo que si en algún momento se desea deja de usar no se pueden eliminar de la base de datos por lo que se debe mantener el archivo en la carpeta Importers para que no falle en tiempo de ejecución.

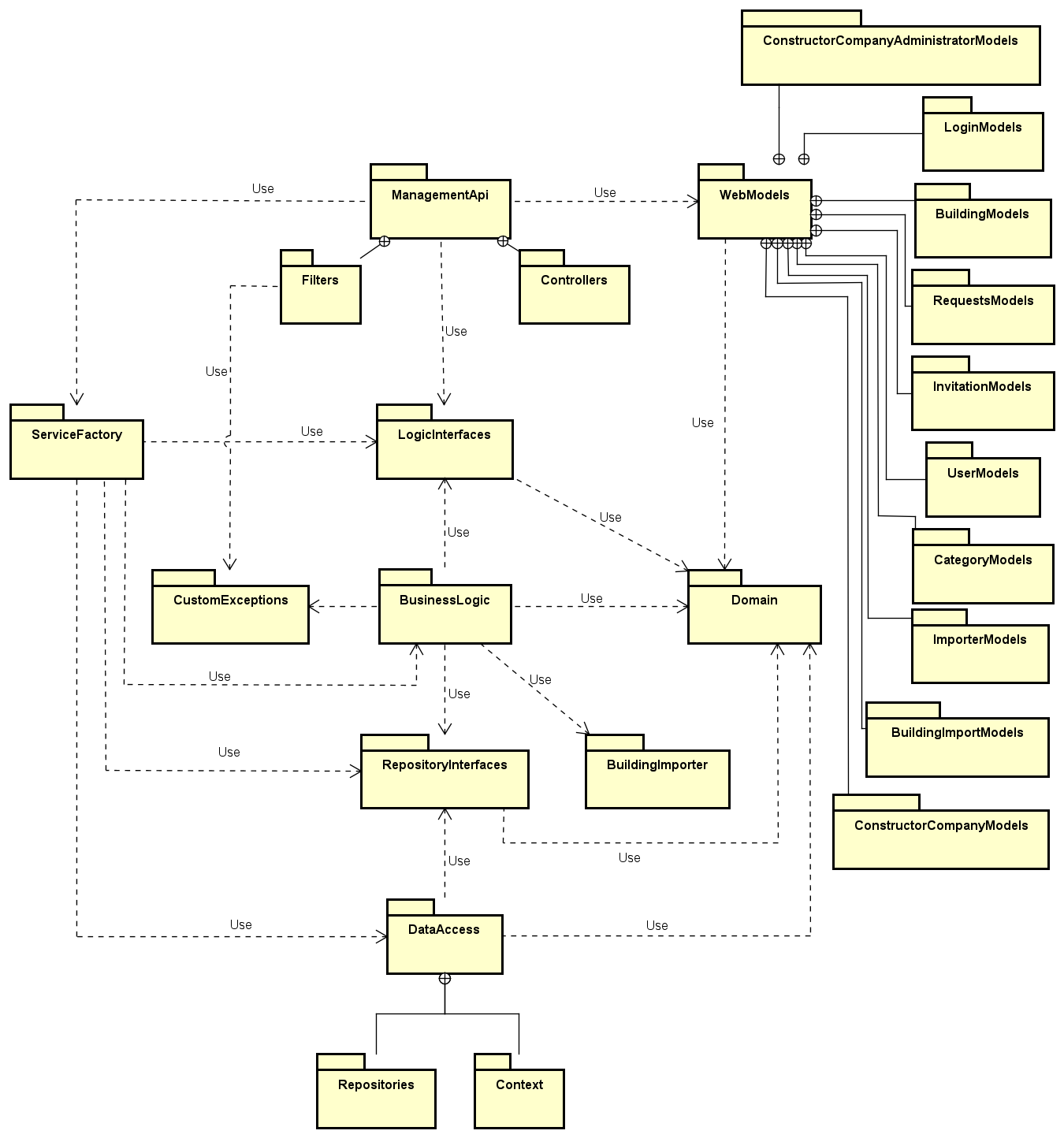
También con respecto a los importadores, se podría mejorar la respuesta que devuelva la API al momento de la importación. Se podría devolver una respuesta por cada intento de creación de edificio, para así poder ver el resultado de la ejecución viendo cuál se creó y cuál falló. Al momento, si uno falla, los que se crearon previamente quedarán en base de datos mientras que el que falló y los posteriores no serán agregados.

# Vista de implementación

Al igual que en la entrega anterior, implementamos la documentación de la solución en base al modelo 4+1 visto en clase. A continuación se detallan 3 de las 5 vistas de dicho modelo comenzando con la de mayor nivel para ir “bajando”.

En esta vista se describen los componentes estáticos como dinámicos implementados en el proyecto, realizamos dos diagramas en esta vista: general de paquetes, de componentes. Es una vista a mayor nivel.

## Diagrama general de paquetes



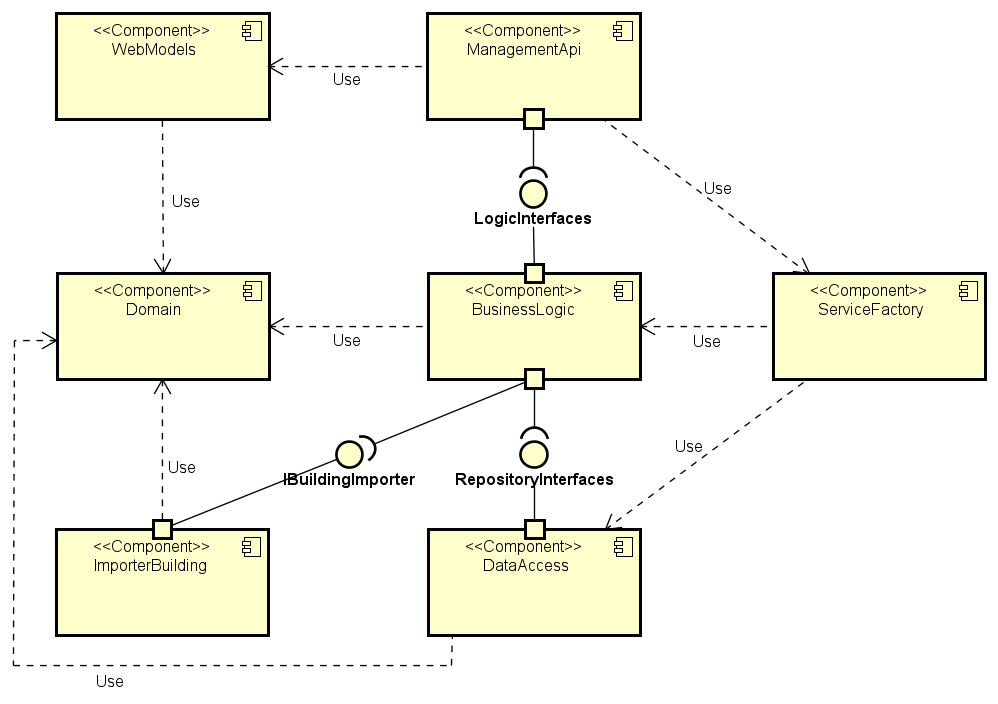
El diagrama general de paquetes que muestra las relaciones entre ellos y los subpaquetes, anidados con el conector de nesting. Podemos ver como las interfaces funcionan como separadores de niveles conociendolas tanto los paquetes como las que las implementan.

En esta nueva entrega se agregan dentro de WebModels los paquetes necesarios para los modelos de entrada y salida para las empresas constructoras, los administradores de empresas constructoras, los importadores y las importaciones. También se añadió el BuildingImporter encargado de agregar nuevas dlls al sistema para poder ser utilizadas.

## Diagrama de implementación

Para el diagrama de componentes utilizamos la mayor implementación posible de UML2 visto en clase. En este caso se ven claramente definidos los niveles y capas que tiene la aplicación. Al igual que en la entrega anterior, la única dependencia que hace más “ruido” es la de ManagementApi con ServiceFactory ya que ManagementApi está dependiendo de un paquete de más bajo nivel. Sin embargo, es un menor precio a pagar ya que la contracara implicaría mayor acoplamiento y menor cohesión.

Además, en .NET el proyecto que ejecuta el Program.cs se encuentra dentro de ManagementApi y este es la clase que contiene el Main y debe conocer la inyección de dependencias y el DbContext.

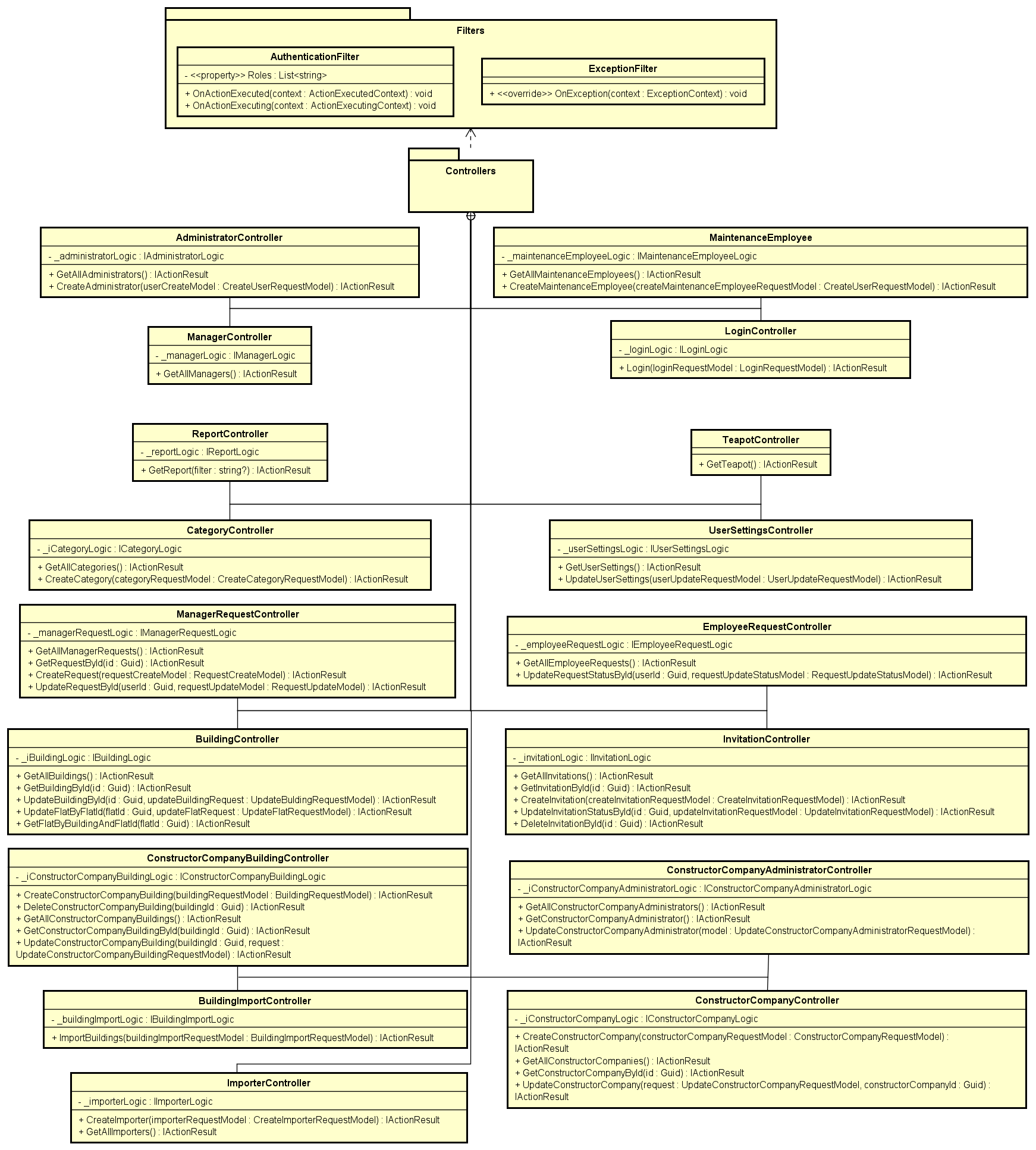


Para esta nueva entrega, agregamos a nivel de componentes el ImporterBuilding que contiene la interfaz y los DTOs a ser utilizados por terceros para implementar el importador que luego será agregado al sistema. Este además expone IBuildingImporter que es utilizado por BusinessLogic para asegurarse que las dlls a correr la implementen y poder ejecutar el método requerido para así obtener la lista de buildings dentro del archivo a importar.

# Vista lógica

Continuamos con la vista lógica que describe los elementos brevemente, sus responsabilidades y cómo se relacionan entre ellos.

## ManagementApi



El proyecto de ManagementApi maneja las responsabilidades de una API, como su nombre lo indica. Esta se encarga de la comunicación con lo externo a la solución, tanto para mandar como recibir. Recibe las request y las deriva a la parte de la lógica de negocio que se encarga de la funcionalidad pedida. Separa las clases del dominio utilizadas en la parte “interna” de la aplicación del “afuera” transformando esas clases en modelos que representan a las mismas dependiendo de la response adecuada.

En el diagrama vemos dos claros subpaquetes dentro del ManagementApi los cuales separan a los filtros, encargados de la autenticación y autorización del usuario así como de la comunicación a través de códigos HTTP al usuario en caso de errores, de los controladores. Estos últimos son los que para cada recurso permiten diferentes verbos HTTP en las request dependiendo las funcionalidades.

Si bien en el diagrama no se ve representado directamente, todas las clases del subpaquete Controllers tienen una dependencia con cada clase del subpaquete Filters. Estas se obvian ya que para la representación del diagrama lo hacían más engorroso y empeoraban la visibilidad de las clases. A su vez, existe la dependencia de Controllers a Filters.

Para esta nueva entrega se agregó el controlador de ConstructorCompanyBuilding ya que las responsabilidades de los buildings se separaron según rol de usuario y por repetición de verbos HTTP debimos separarlo en dos controladores, el anterior se mantuvo para los managers. También se agregaron controladores para los administradores de empresas constructoras, las empresas constructoras mismas, los importadores y las importaciones de edificios.

## WebModels

WebModels es un paquete el cual tiene como responsabilidad albergar los diferentes modelos que utiliza la API tanto de entrada como salida. Estos se separan del proyecto de API por dos razones: no es la responsabilidad de la API contener estas clases, y además permite la separación de capas ya que las clases del dominio son utilizadas en la parte “interna” de la aplicación.

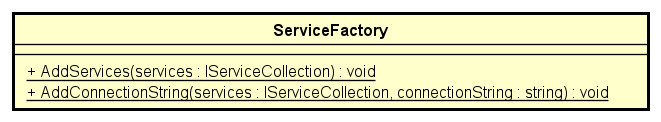
Ver en anexo el [Diagrama de WebModels](#_mt3ggt87m7me).

En el diagrama podemos ver los diferentes subpaquetes ordenados por recurso para que cada uno de estos utilice únicamente el de él. En la mayoría de los subpaquetes existen casi las misma cantidad de modelos utilizados para entrada que verbo HTTP que recibe body en la request del recurso, pero no es así con los de salida, ya que la mayoría de las response utilizan el mismo formato retornando el objeto casi entero.

Se agregaron nuevos modelos de entrada y salida para cada controlador mencionado anteriormente con la excepción de las importaciones de edificios ya que al tratarse de un post retorna la lista de edificios creados que utiliza los modelos de salida del BuildingModels.

## ServiceFactory

El paquete de ServiceFactory contiene una única clase con dos métodos. Este paquete es un resultado de la inyección de dependencias. Se responsabiliza de la inicialización de cada interfaz con la correspondiente clase que implementa dicha interfaz, así al referirse a un método de la interfaz se deriva en la implementación del método en la clase. Además, se encarga de la inicialización del DbContext con el BuildingBossContext, la cual hereda de la anterior, y utiliza el connection string definido para el Sql server.



Es uno de los dos proyectos del cual depende ManagementApi ya que lo utiliza para levantar la aplicación. A su vez, es el único proyecto que no cambió de la primera a la segunda entrega ya que se agregaron AddScoped dentro del método AddServices para que se inyectarán las nuevas clases al ser necesarias, pero esto no se ve reflejado en el diagrama.

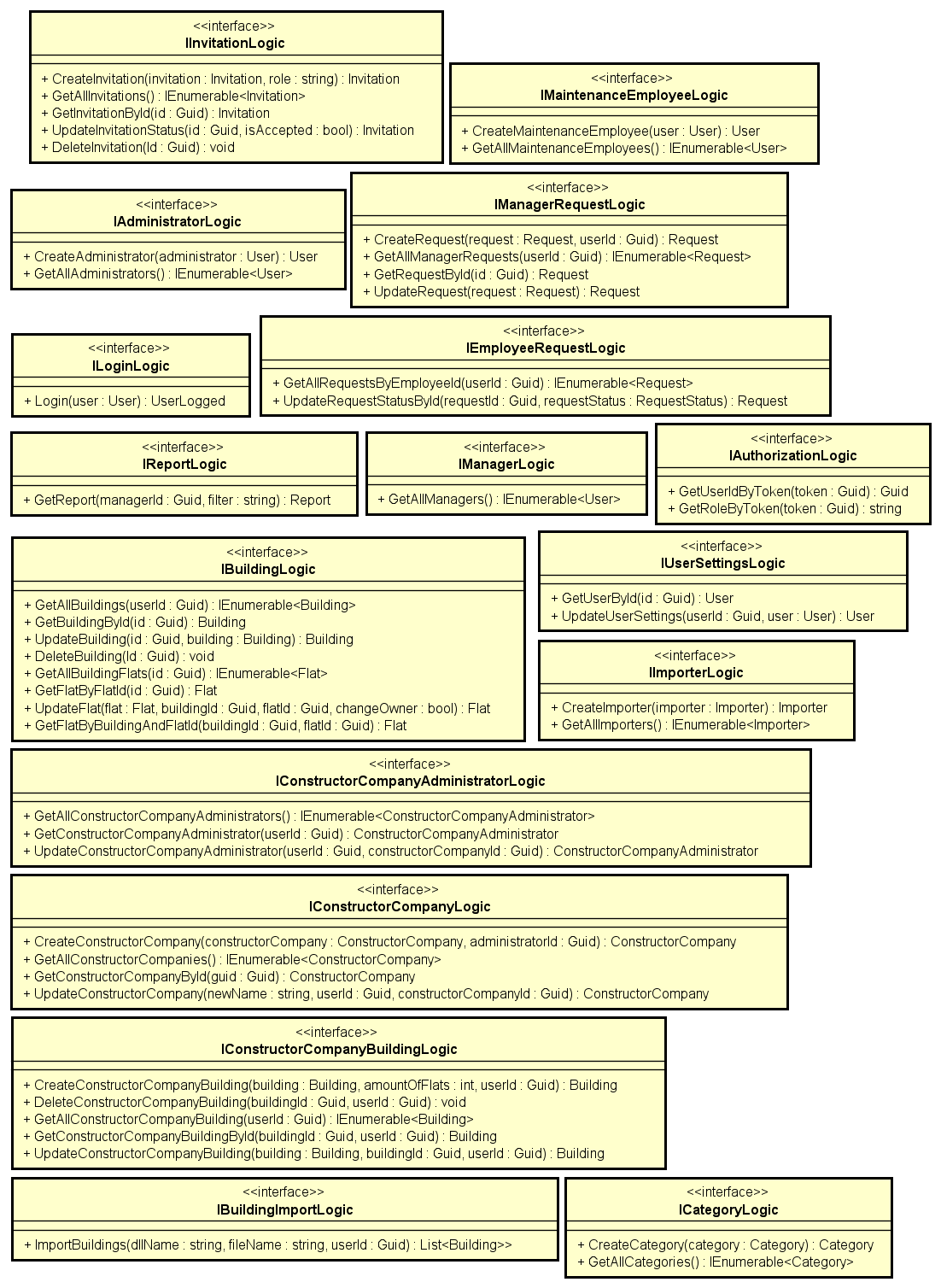
## LogicInterfaces

LogicInterfaces es el paquete responsable de almacenar y definir las interfaces que va a utilizar el proyecto de Api para comunicarse con la lógica del negocio y conocer los métodos que ésta expone sin conocerla directamente, lo que permite manejar un nivel de abstracción y hacer posible las diferentes capas en la aplicación. El proyecto ManagementApi puede conocer los métodos a utilizar sin conocer cómo están implementados.

Este tipo de proyectos cumple con Open-Closed y Liskov de los SOLID, ya que al estar separadas las capas por interfaces y no tener dependencias directas, los cambios tienen menor impacto.

Vemos que hay una interfaz por recurso exponiendo todos los métodos que dicho recurso puede utilizar, pero no vemos ni donde son implementados ni cómo. Estos no se relacionan entre sí ya que no tiene sentido alguno que lo hagan, su única función es la especificada anteriormente.

Se agrega para esta nueva entrega una interfaz por recurso definido anteriormente.



## BusinessLogic

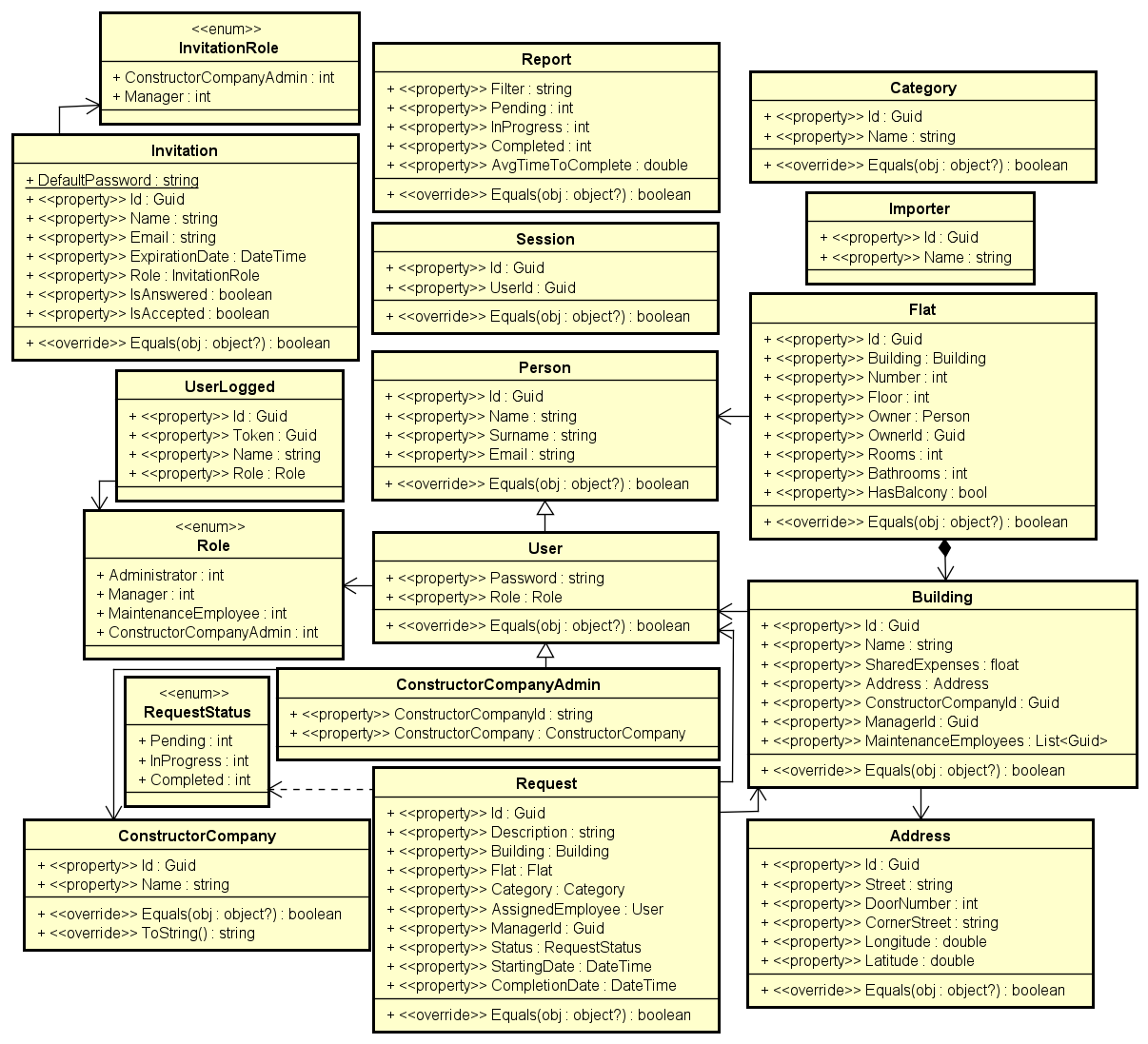
En cuanto a la BusinessLogic, es el paquete responsable de contener la lógica del negocio como menciona su nombre. Este implementa las interfaces de la LogicInterfaces definidas anteriormente. Sin embargo, podemos notar que no tenemos una clase por recurso como en los paquetes anteriores sino que se separan por responsabilidades del dominio.

Además de implementar la LogicInterfaces, utiliza la RepositoryInterfaces, por lo que se encuentra en una capa media. No conoce lo que tiene a un más alto nivel, ni a un menor nivel. El paquete conoce que métodos debe implementar y cuáles puede utilizar nada más.

Ver en anexo el [diagrama de la BusinessLogic](#_peos126dxhkt).

Para esta nueva implementación se añadieron diferentes lógicas como son las de ConstructorCompanyAdministrator, ConstructorCompany, Importer, BuildingImports. A diferencia de las anteriores, la nueva interfaz ConstructorCompanyBuilding es implementada por BuildingLogic ya que mantiene la lógica del obligatorio anterior únicamente con diferentes roles por lo que ese cambio solo impactó al controlador, otra ventaja de la separación de capas.

## Domain



Este paquete contiene las clases del dominio. Las cuales se utilizan en toda la parte “interna” de la aplicación. Debido a lo mencionado anteriormente en BusinessLogic, estas clases quedan con dominio casi raquítico ya que la forma de relacionarse lógicamente entre ellas ya no es su responsabilidad.

Podemos ver en el diagrama varias clases nuevas como ConstructorCompany

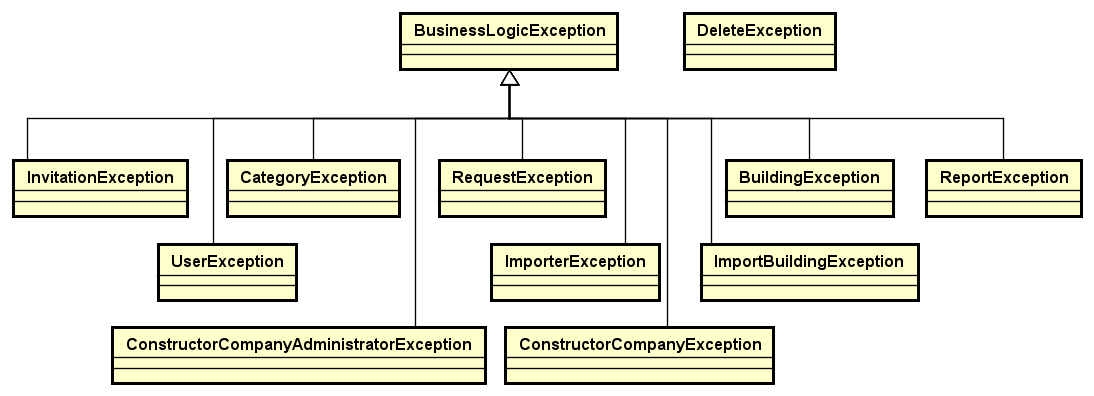
Administrator que hereda de User. Se agrega un enum para controlar el rol a crear por invitación. Se agregan las clases UserLogged para retornar los elementos necesarios al controlador al momento de hacer Login, ConstructorCompany e Importer.

Por último, se agregan relaciones entre clases dependiendo de la mejor opción de lo necesario para mostrar en la aplicación web (front-end).

Podemos ver que las clases igualmente se relacionan entre ellas, ya sea por herencia o asociación más que nada. Sin embargo, estas relaciones no poseen tanta lógica como en el BusinessLogic.

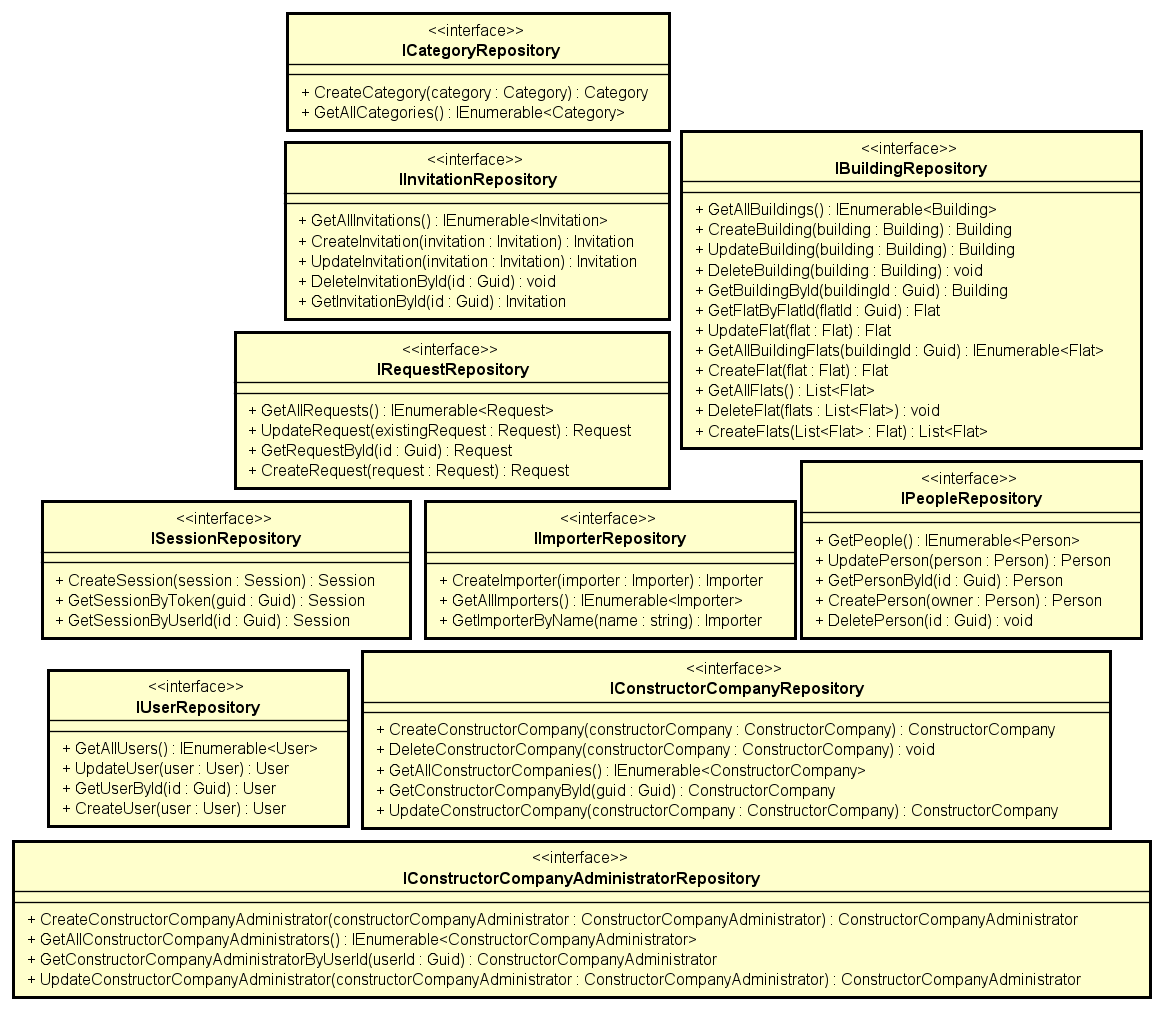
## CustomExceptions

El paquete CustomExceptions contiene las excepciones personalizadas de la aplicación. La mayoría son excepciones de la BusinessLogic, excepto por una que se trata de la excepción largada para hacer idempotente al delete. Más adelante se explica en más detalle el manejo de excepciones. Sin embargo, una clara oportunidad de mejora es la implementación de subpaquetes dentro de CustomExceptions para una mayor organización y especificación cuando ciertas clases usen el paquete. En este momento, para todos los paquetes que usen a este, sus clases podrán utilizar cualquiera de las excepciones, cosa que podemos no querer.



## RepositoryInterfaces

El RepositoryInterfaces es el paquete responsable de las interfaces que separan a la BusinessLogic de los repositorios y el acceso a base de datos. Nuevamente vemos el nivel de abstracción que este genera formando así un nivel más. Al igual que con ManagementApi, BusinessLogic conoce los métodos que puede utilizar pero no quienes lo implementan ni cómo.



Al igual que LogicInterfaces, este proyecto al crear abstracciones permite menor impacto al realizar cambios y mayor cumplimiento de los SOLID mencionados previamente.

## DataAccess

Por último, está el DataAccess encargado de albergar el DbContext y los repositorios que se relacionan con el mismo y sus DbSet. Al igual que el BusinessLogic, implementa los métodos de la interfaz que lo separa de un nivel más alto, y ejecuta cuando es llamado, sin conocer quien lo llamó.

Podemos ver en el diagrama como la mayoría de las clases tienen los mismos métodos: GetAll, GetById, Update y Delete. En los repositorios se busca que estos sean lo más tontos posibles ya que la lógica se encuentra en otro lugar. Estos deben ser “puertas” de entrada y salida de datos. Todos los repositorios realizan acciones sobre DbSets ubicados en el contexto, por lo que el subpaquete Repositories debe conocer a Context.

Ver en anexo el [diagrama de DataAccess](#_gvylicu5kq70).

Creamos un repositorio por DbSet con la excepción de los flats que se encuentran dentro del BuildingRepository ya que tienen una relación de vida con los buildings.

# Vista de procesos

Por último, está la vista de procesos ya que la de despliegue y casos de uso aún no las vimos. En esta vista se describen las secuencias y performance de los métodos y cómo interactúan con otros. Realizamos tres diagramas de secuencia a continuación.

## Diagramas de interacción

Realizamos tres diagramas de secuencia: en el primero se muestra la secuencia de creación de reporte, el segundo muestra la secuencia de actualización del estado de una invitación para aceptarla, mientras que el último caso es el de la creación de un building. Los tres diagramas quedan un poco chicos ya que tienen bastantes pasos con clases de nombres largos por lo que los adjuntamos en la carpeta de documentación.

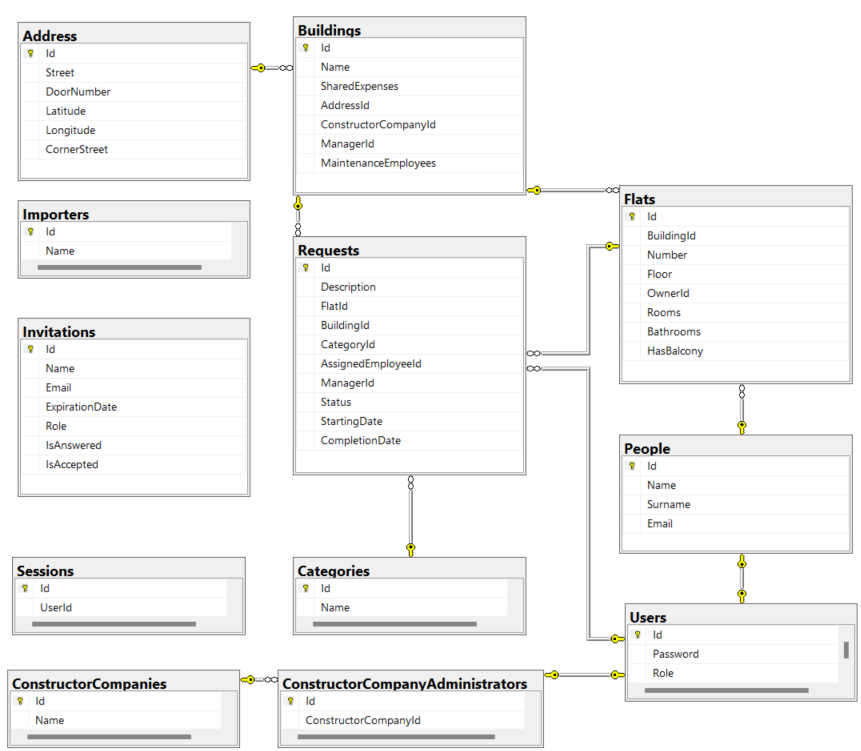
En el [primer diagrama de secuencia](#_hl5s5gz0y9bh) mantuvimos el diagrama de la entrega anterior de los reportes para building que implementa el Template Method, pero lo actualizamos con los cambios hechos sobre la clase reporte mencionada anteriormente que simplifica su ejecución. Agregamos nombres que describen la ejecución ya que utilizamos ++ sobre enteros que no son tan descriptivos en el diagrama de secuencia.

En el [segundo diagrama de secuencia](#_rlqvihndvut1) tomamos una funcionalidad que altere más de un DbSet en base de datos actualizando la invitación y creando un usuario nuevo.

Por último, en el [tercer diagrama de secuencia](#_1spsenlfwziy) tomamos otra funcionalidad que altere a más de un DbSet ya que la creación de un building crea sus flats con sus owners vacíos. Cambiamos la eliminación de un building de la entrega anterior por la creación de un building ya que se tornó muy extensa la eliminación con nombres de clases largos que no permiten la buena legibilidad.

# Modelo de tablas de la estructura de la base de datos

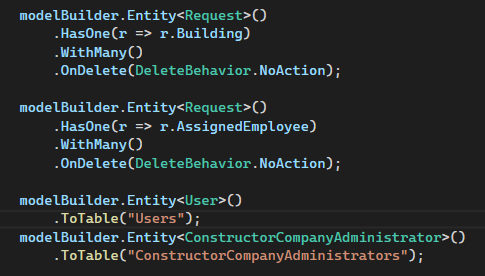
A continuación se presenta el diagrama sobre el modelo de tablas generado por Entity Framework, utilizando el método Code-First, que dado el código, genera el modelo de base de datos:



Al generar el modelo, debimos especificar algunos aspectos con FluentApi, para evitar comportamientos inadecuados o problemas en el modelo de la Base de datos.

En primer lugar se debió especificar que cuando se eliminen Requests no se eliminen ni el edificio que tiene asociado ni el Empleado de mantenimiento que tiene asignado.

En segundo lugar se debió explicitar que se modele la herencia People->User->ConstructorCompanyAdministrator separando en tablas distintas cada entidad:



# Jerarquía de herencias

En la primera entrega se utilizó herencia en dos lugares: el primero es en el Domain con User que hereda de Person, ya que conceptualmente son casi iguales, nos ayuda a no repetir código. El segundo lugar es en el CustomExceptions, utilizamos herencia en las excepciones largadas en BusinessLogic. Estas tienen nombre de la clase en donde es largada y heredan de BusinessLogicException que hereda de Exception. Utilizamos esta herencia para un manejo de errores más sencillo preguntando si el tipo de la excepción corresponde con BusinessLogicException, y que permita además agregar diferentes nuevas clases en la BusinessLogic y lo único que cambia es en una nueva excepción en CustomExceptions que herede de BusinessLogicException.

En esta segunda entrega, se agregó una herencia en el Domain, con ConstructorCompanyAdministrator heredando de User. Para esto se utilizó el mismo criterio que con Person y User, dado que un ConstructorCompanyAdministrator es en definitiva un User que puede (o no) tener una ConstructorCompany asociada.  
 Aunque conceptualmente son entidades distintas, comparten la mayoría de los atributos (todos menos la ConstructorCompany) y usa las mismas validaciones al momento de la creación.

También, se agregaron nuevas excepciones, que heredan de BusinessLogicException, una para cada una de las nuevas clases de la lógica que se debieron agregar para implementar las nuevas funcionalidades.

# Inyección de dependencias

La inyección de dependencias es un patrón en el cual un objeto o función recibe a otro del cual depende. Tiene también como objetivo separar a quien depende de la creación del objeto dependido.

Esto lo vemos claramente en Service Factory. Es el proyecto que inyecta, para cada instancia de una interfaz, una instancia de la clase que implementa dicha interfaz. Por lo que abstrae la creación de los objetos, que son usados por otras clases, de las clases que los usan. Esto tiene como resultado mayor separación de niveles de abstracción que resulta en un menor acoplamiento.

Este cumple con el patrón GRASP de fabricación pura, ya que asigna un conjunto altamente cohesivo de responsabilidades a una clase artificial (que no representa nada del dominio del problema) de forma de dar soporte a la alta cohesión y bajo acoplamiento.

# Patrones

El patrón más claro de ver es el Facade, este es un patrón estructural que funciona como fachada, como dice el nombre. Es el paquete ManagementApi quien cumple con dicho patrón ya que al ser un punto de entrada y salida de la aplicación lo que está afuera no conoce lo que se encuentra dentro ni su implementación y lo de adentro no conoce lo que está afuera ni quien lo llama.

En la lógica de los reportes se implementa el patrón Template Method, ya que se trata de un algoritmo en el que tiene que cambiar el campo por lo que se filtra, se trata de un patrón de comportamiento.

La implementación sucede en RequestReport, esta es la clase abstracta que define el template method (GenerateReport). En él se definen los pasos. El primero (FilterRequests) es un método abstracto que las clases que heredan de esta deben sobreescribir, en él se les da valor a las properties que se van a utilizar en los pasos siguientes. Se separan los campos únicos por los que se van a ordenar y filtrar las requests y se separan estas últimas por esos filtros.

El siguiente paso no es sobreescribible ya que se trata siempre de separar las requests por su estado y de calcular el tiempo promedio de las completadas para luego ser retornadas.

Luego existen dos clases que heredan de RequestReport: una es para los edificios y otra para los empleados de mantenimiento. En ambas se implementa el FilterRequests, filtrando por su conveniencia.

Utilizar un patrón como este nos permite que si en el futuro queremos realizar otro reporte sobre las requests, implementarlo significaría realizar una clase con un método únicamente. Por lo que promueve Open-Closed de SOLID.

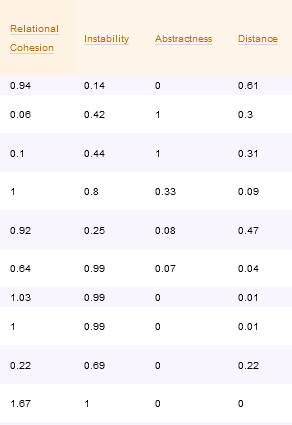
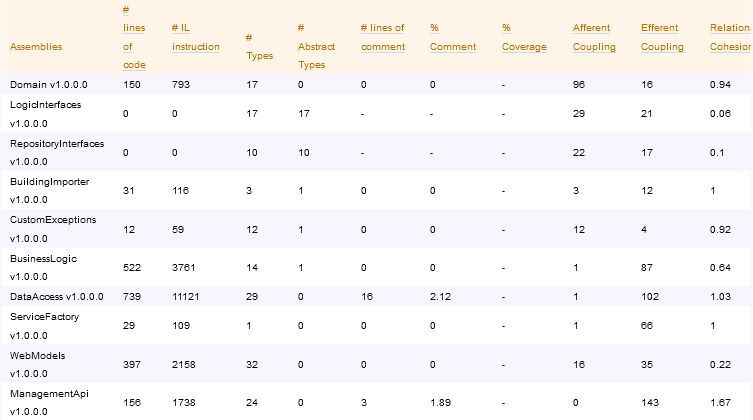
# Métricas de diseño

Una métrica es una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado. Es una forma de medición y escala definidas para realizar mediciones de uno a varios atributos.

En el curso nos centramos en las métricas de inestabilidad (I), abstracción (A), cohesión relacional (H) y la distancia (D o distancia normalizada D’).

La inestabilidad se encuentra como métrica dentro de los principios de dependencias de paquetes referente al Stable Dependencies Principle (SDP) y está relacionada con la cantidad de trabajo requerido para realizar un cambio. Se calcula a partir de las relaciones aferentes (clases fuera del paquete que dependen de alguna clase dentro del paquete) y eferentes (clases fuera del paquete que son dependidas por clases dentro del paquete). Su ecuación es ***I = Ce/(Ca+Ce)*** y se encuentra en el rango de 0 a 1 siendo 1 lo más inestable posible. Podemos ver que el paquete más inestable es de la solución es ManagementApi lo cual tiene sentido ya que todas sus dependencias son eferentes. Le sigue el resto de los paquetes con la excepción de los dos paquetes de interfaces y el de excepciones que son los más estables.

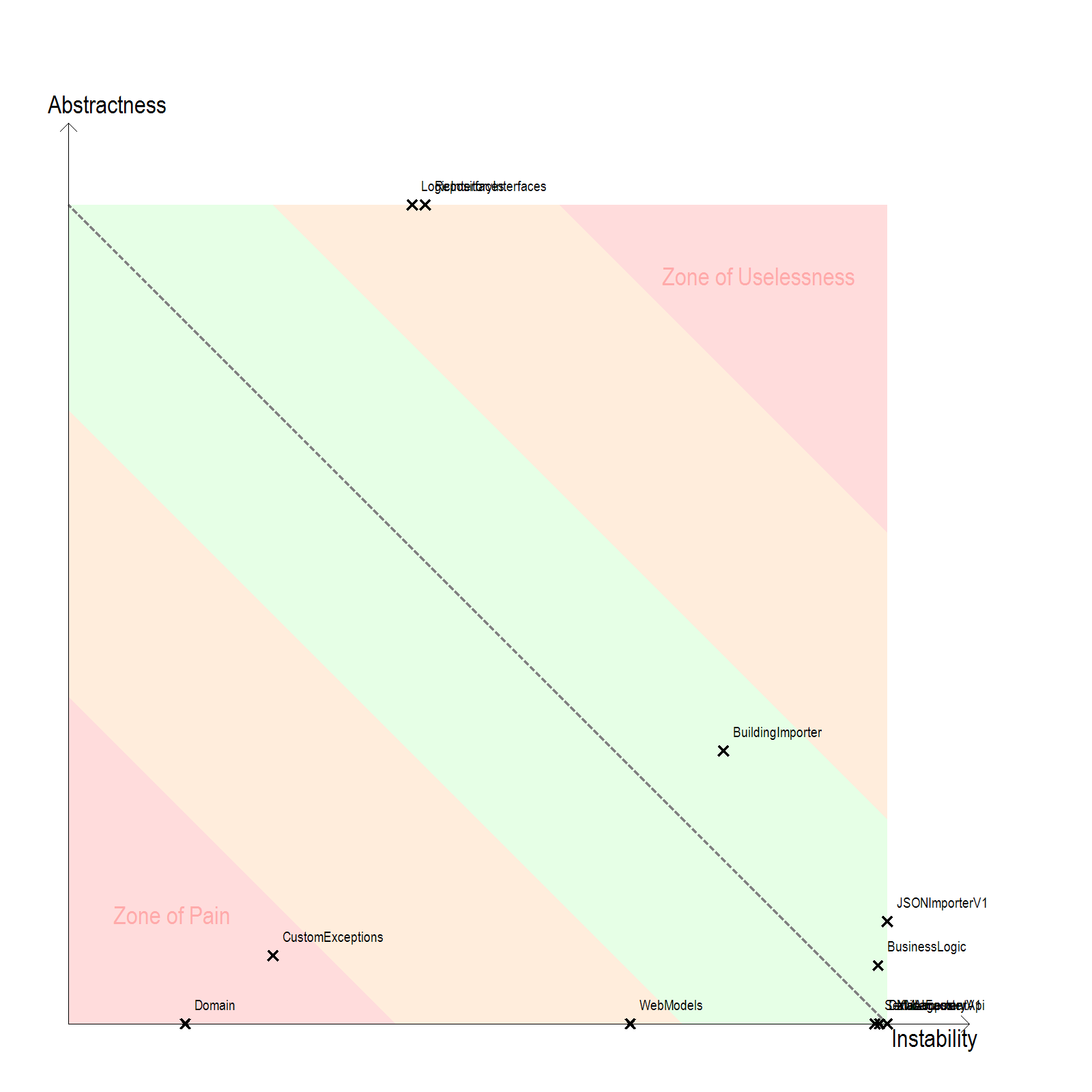
La abstracción también se encuentra dentro de los principios de dependencias de paquetes referente a Stable Abstraction Principle (SAP) que dice que los paquetes estables deberían ser abstractos. Se calcula a partir de las clases abstractas o interfaces en el paquete (Na) y las totales (Nc). Su ecuación es ***A = Na/Nc*** y también varía entre 0 y 1 siendo 1 lo más abstracta posible. Podemos ver que los paquetes más abstractos son LogicInterfaces y RepositoryInterfaces ya que se trata de dos paquetes que contienen únicamente interfaces.



La tercera métrica se trata de la Cohesión relacional (H) que mide la relación entre las clases de un paquete. Para calcularla se necesitan la cantidad de relaciones internas (R), cantidad de clases e interfaces dentro del paquete (N). Su ecuación es la siguiente ***H = (R+1)/N***. Es considerada buena cohesión relacional cuando se encuentra entre 1,5 y 4. En la tabla podemos ver que el único paquete que se encuentra en el rango es ManagementApi ya que todos utilizan el AuthenticationFilter. Sin embargo, el resto no se encuentra dentro del rango especificado ya que al favorecer DIP y priorizar que el diseño de la aplicación fuera más mantenible resulta en un menor acoplamiento dentro de los paquetes, en cada capa se separan las clases de forma que las responsabilidades de las mismas se adecúen a las responsabilidades de la capa lo que resulta en menor relación dentro de cada paquete. Esto resulta en un H que se acerca más a uno que lo ideal.

A partir de las dos primeras métricas vistas podemos calcular la última que se pedía para el proyecto. La distancia se trata de una métrica que relaciona la inestabilidad con la abstracción de los paquetes. Se calcula de la siguiente manera que es la forma en que se muestra en la tabla y esta varía entre 0 y 0,707, pero también existe la normalizada (D’) que es solo el valor absoluto y varía entre 0 y 1, siendo en ambas lo más ideal lo más cercano a 0 posible. Una vez obtenidas las distancias podemos realizar la gráfica a continuación.

Los puntos ideales son las dos esquinas dentro de la zona verde y los peores son la zone of useless que no le encuentra sentido al paquete y la zone of pain que significa que dicho paquete se encuentra muy acoplado.



Podemos ver en la gráfica que la mayoría se encuentra en la inestabilidad y poco abstracto lo cual es un punto ideal. Vemos como BusinessLogic debido al patrón utilizado en los reportes contiene una clase abstracta. También BuildingImporter se ubica dentro de la zona verde pero más abstracta ya que es el paquete que muestra los dtos e interfaces a ser utilizadas por las dlls, lo que significa que un tercio de sus clases son abstractas pero no depende de ningún otro paquete.

Sin embargo hay excepciones como por ejemplo WebModels que lo ubica por fuera de la zona verde. Esta zona va hasta el 0,2 y dicho paquete tiene 0,22 de distancia lo cual sigue siendo bueno, permitimos hasta 0,25 como bueno y hasta 0,3 como permisible (visto en clase) pero con una búsqueda de una mejor solución. Tiene sentido el 0,22 del paquete ya que es totalmente concreto pero solo depende de domain.

En el límite de lo permisible se encuentran los dos paquetes de interfaces ya que estos también dependen de Domain. Sin embargo como se encuentran entre dos capas es necesario que estas conozcan el dominio de al menos una de esas capas. En el caso de RepositoryInterfaces es el mismo dominio para las dos capas a las que relaciona.

Por último está CustomException y Domain que se ubican en la zone of pain ya que son clases concretas que no dependen de ningún otro paquete. Son necesarios ya que permiten a los otros paquetes tener menos acoplamiento y mayor cohesión, permitiendo separación de capas y abstracción en cuanto a la implementación de cada capa.

# Principios de diseño

## SOLID

La aplicación se divide en capas y estas se comunican entre ellas mediante interfaces. Esto favorece el Dependency Inversion Principle, ya que de esta manera, se consigue que los módulos de alto nivel (las capas de niveles superiores) no dependan de módulos de bajo nivel (capas de niveles inferiores) sino de abstracciones de los mismos (interfaces).

A su vez, sigue el principio LSP (Liskov Substitution Principle), porque permite utilizar las interfaces como si fueran las clases que las implementan. Se aprovechó en cada una de las capas, utilizando las interfaces de los módulos de bajo nivel para poder usar sus métodos cumpliendo con Interface Segregation y bajando el acoplamiento. También fue muy útil a la hora de hacer los tests unitarios de una capa, ya que, en caso de necesitar capas inferiores, se puede llamar a la interfaz, moqueando el comportamiento de la capa inferior, pudiendo así obtener la salida deseada, sólo habiendo implementado la interfaz con la firma de los métodos.

El favorecimiento de estos principios, también causó que se respetara el Open-Closed principle, ya que mientras se mantenga la implementación de las interfaces, se puede hacer cambios en los diferentes niveles sin afectarse entre sí. Esto refiere al bajo acoplamiento de la solución.

Respecto a las responsabilidades, se intentó diseñar la aplicación de modo que estuvieran bien repartidas, para lograr una alta cohesión en la solución y respetar el Single Responsability Principle. Cada una de las capas de la aplicación tiene distintas responsabilidades: la capa de la API tiene la responsabilidad de dialogar con el usuario, la de la Business Logic, la de hacer las validaciones lógicas para la lógica de negocio a representar y la capa de acceso a datos la del acceso a la Base de datos para escribir u obtener información.

Dentro de cada capa se dividen las responsabilidades para lograr una buena cohesión, dividendo en distintos controladores (uno para cada recurso) la capa de la API, en distintas clases de lógica según las validaciones a realizar y en distintos repositorios para el manejo de las entidades en la base de datos.

De esta forma, se logra tener menos motivos de cambio alrededor del código. En caso de querer cambiar una funcionalidad, al tener una alta cohesión se podrá cambiar en los lugares donde se impacte específicamente.

## GRASP

Como se dijo anteriormente, el sistema presenta alta cohesión y favorece el principio de variaciones protegidas, porque gracias a las interfaces se protegen las distintas capas de los cambios que se puedan producir en las otras.

Exceptuando por las clases de Domain y Service Factory, no hay dependencias entre capas. Si bien los paquetes Domain y Service Factory están más acoplados, es un menor precio a pagar comparando otras soluciones que podrían generar acoplamiento entre capas, como en obligatorios de semestres anteriores.

Finalmente, como mencionamos antes, por el uso de la Service Factory explicada en la sección de Inyección de dependencias, también se utiliza el principio de Fabricación pura.

Las clases están poco relacionadas entre sí resultando en un bajo acoplamiento.

# Decisiones propias de diseño

Una de las decisiones de diseño tomadas se puede ver en el update de una invitación ya que para realizar un PUT al recurso debe pasarse un body que contiene solo un parámetro de tipo bool. Si se envía un body vacío, .NET parsea el booleano a false por lo que se niega la invitación sin haber pasado dicho booleano. La solucíón encontrada fue permitir que ese booleano contenga valor nulo y evitar el parseo de .NET. Conocemos por ser hablado en clase que no se deben enviar nulos entre capas, en este caso decidimos realizarlo de todas formas por lo explicado anteriormente. Una vez que recibe el valor en la capa lógica, esta chequea que no se trate de un null, si es de esta manera se larga una excepción con el mensaje correspondiente.

Otro lugar en el que se encuentra un nulo y se transmite entre capas es al momento de crear un building. Se crea el building sin managerId y se transmite al repositorio que agrega el campo del building como nulo. Tomamos esta decisión ya que simplifica la complejidad del algoritmo debido a que la otra forma que encontramos significaría ingresar a la base de datos un manager vacío y relacionarlo con el building pero que luego se debe eliminar al momento de la edición del building ya que la única forma de creación de managers es a través de una invitación.

Otra decisión de diseño es al momento de realizar un PUT a un building para editarlo se debe ingresar en todos los request la lista de empleados de mantenimiento deseada, tanto los que se agregan en esa request como los que ya pertenecían anteriormente, ya que esta se sobreescribe en cada update del building.

Cuando se invita a un administrador de empresa constructora, puede ver las empresas constructoras del sistema y seleccionar una de ellas para que sea su empresa constructora. También puede crear una empresa constructora que se le asignará como su empresa constructora.

# Supuestos

A continuación pasamos a nombrar los supuestos realizados por el equipo en base a los especificado por el cliente:

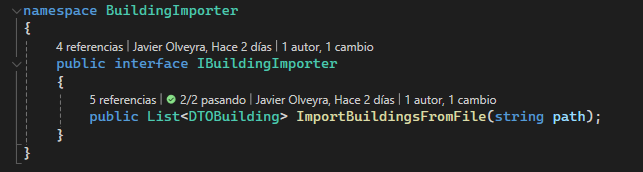
* No se pueden eliminar ni crear flats por fuera de la creación y eliminación de buildings.
* No se pueden crear invitaciones que venzan en menos de 24hs. Si el valor enviado por el modelo de entrada es menor a 1 se toma como un día desde el momento de la creación de la invitación.
* No puede haber dos empresas constructoras con mismo nombre en el sistema.
* Un administrador de empresa constructora no se puede cambiar de empresa constructora.
* Ninguno de los edificios incluidos en un archivo que se va a agregar por reflection debe ya estar en el sistema.
* Si se produce un error lógico en la importación de un edificio quedarán agregados todos los edificios previos al error y en pantalla se le mostrará al usuario el mensaje de error.

# Mecanismos utilizados para permitir la extensibilidad solicitada en la funcionalidad de este nuevo obligatorio

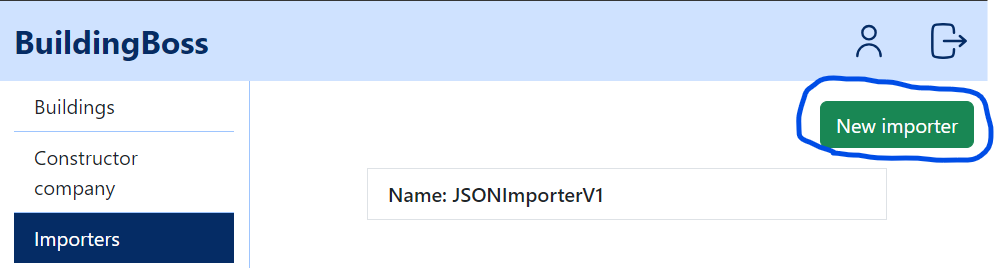
Para permitir la extensibilidad en la funcionalidad de la aplicación se utilizó la técnica de Reflection para poder agregar dlls con importadores que permitan importar edificios en cualquier formato.

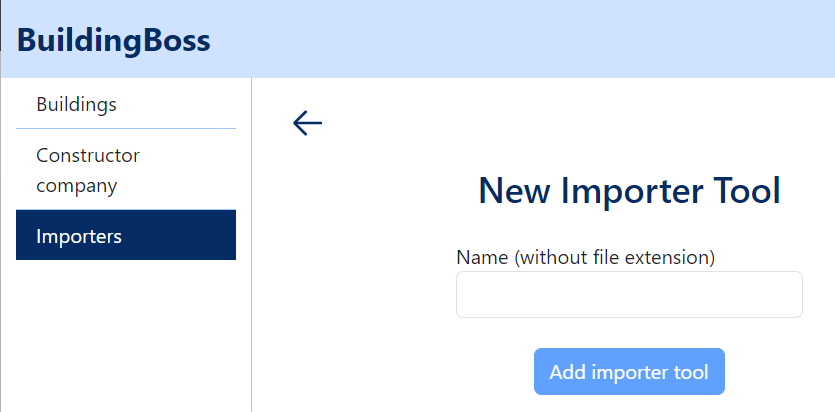
Para esto, se implementó:

1. Una interfaz con el método que deben implementar dichas dlls:

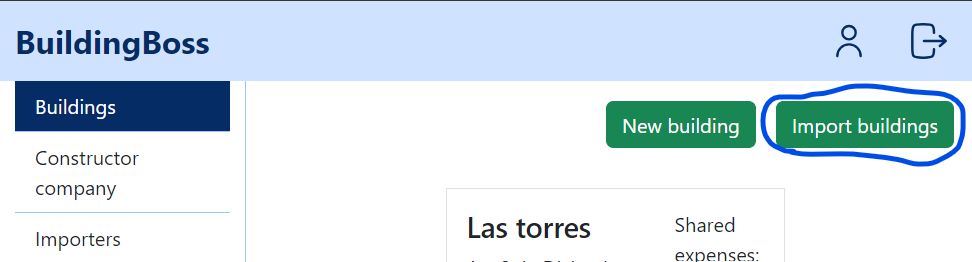


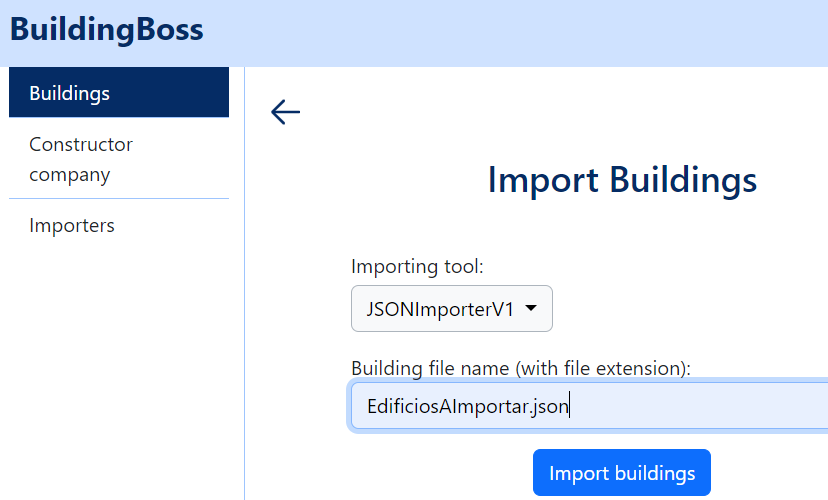
1. Tipos DTOBuilding y DTOFlat, que son los tipos que nuestra aplicación puede interpretar y transformar a la clase Building y Flat del Domain respectivamente.
2. A nivel de API:
   1. Se implementó el controlador ImportersController que permite agregar nuevos importadores (nuevas dlls), para que el usuario pueda, en tiempo de ejecución de la aplicación, luego de dejar las dlls que implementen el método ImportBuildingsFromFile en la carpeta Importers del servidor, agregar en el sistema las rutas a estas dlls para que el sistema pueda invocar la importación de edificios usándolas.
   2. Se implementó el controlador BuildingImports que permite importar edificios llamando a un método de la lógica que llama a la dll cuya ruta se pasa por parámetro, invoca el método ImportBuildingsFromFile que implementa dicha dll, transforma los DTOBuildings en Buildings y los agrega en el sistema.
3. En la interfaz de usuario se agregó para los administradores de empresas constructoras:
   1. Página para agregar las rutas a los nuevos importadores.





* 1. Página para importar edificios que permite seleccionar el importador a utilizar de los que el sistema conoce, especificar el nombre del archivo a especificar, e importar los edificios del archivo.





De esta forma, si se quiere agregar una nueva forma de importar edificios, lo que se debe hacer es lo siguiente:

## Instructivo de importación de edificios mediante una dll:

1) Dejar la dll (que implementa la interfaz mencionada anteriormente) del nuevo importador en la carpeta Importers del servidor. Para que la importación funcione, se debe dejar, además de las dlls, todas las dlls que necesiten los importadores para funcionar.

2) Usando el endpoint api/v2/importers agregar en el sistema el nombre de la dll. El sistema registra los importers con su nombre (el nombre del archivo sin el .dll).

3) Dejar el archivo de edificios a importar con el formato que interpreta la dll dentro de la carpeta BuildingFiles del servidor.

4) Usando el endpoint api/v2/buildingImports, especificar el nombre del archivo (con la extensión) para solicitarle al sistema que importe el archivo. En la página Buildings se podrá ver los nuevos edificios importados.

En esta entrega se incluyen dos .dlls: JSONImporterV1.dll y XMLImporterV1.dll, que pueden agregarse al sistema para, en el caso de JSONImporterV1.dll importar los edificios con el formato del archivo de ejemplo enviado en el foro, y en el caso de XMLImporterV1.dll importar los edificios con el siguiente formato:

<?xml *version*="1.0" *encoding*="UTF-8" ?>

<root>

<edificios>

<nombre>Nombre Ejemplo</nombre>

<direccion>

<calle\_principal>CallePrincipalEjemplo</calle\_principal>

<numero\_puerta>1234</numero\_puerta>

<calle\_secundaria>CalleSecundariaEjemplo</calle\_secundaria>

</direccion>

<encargado />

<gps>

<latitud>-0.176</latitud>

<longitud>-78.48</longitud>

</gps>

<gastos\_comunes>5000</gastos\_comunes>

<departamentos>

<piso>1</piso>

<numero\_puerta>101</numero\_puerta>

<habitaciones>3</habitaciones>

<conTerraza>false</conTerraza>

<baños>2</baños>

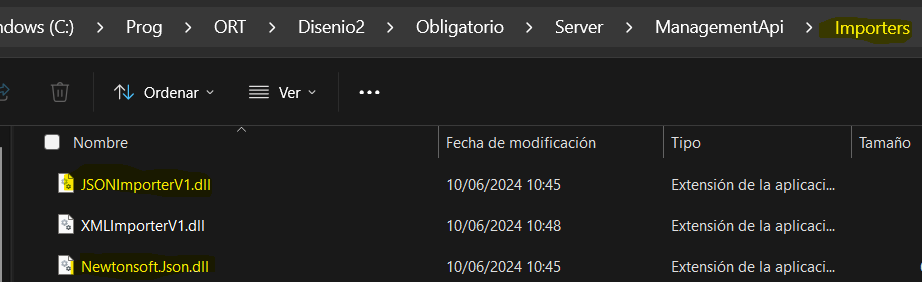
<propietarioEmail>email.ejemplo@mail.com</propietarioEmail>

</departamentos>

</edificios>

</root>

Notar que para el caso de la .dll de JSONImporterV1 necesita la .dll de terceros Newtonsoft.json para funcionar, por lo que, como dice en el punto 1 del **Instructivo de importación de edificios mediante una dll** (más arriba), debe tener dicha dll dentro de la carpeta Importers para funcionar:

  
De lo contrario se dará un error en tiempo de ejecución.

Resumen de las mejoras al diseño

Implementando las nuevas funcionalidades nos encontramos con que nuestra solución admitía la aplicación de los cambios sin repercusión en lo que ya teníamos. Es por esto que no debimos hacer cambios significativos en el diseño de la aplicación para incluirlas.

Los aspectos que cambiamos fueron:

1. Mejoramos la extensibilidad de la aplicación al implementar la técnica de reflection vista en clase ya que permite al usuario extender la aplicación para agregar nuevos edificios.
2. Mejoramos el rendimiento de Entity Framework ya que la tabla de People ya no guarda los datos del User.

Entity Framework representa las herencias automáticamente dejando en la misma tabla los datos de la clase padre, la clase hija y el resto de las subclases, usando un atributo Discriminador que indica de qué tipo es la entidad.

Para el obligatorio 1 se optó por dejar de esta manera la representación de las clases, esto debido a que se tenía una sóla clase hija que tenía sólo dos atributos más que la clase padre.

Para este segundo obligatorio, se pidió que hubiese otro tipo de usuario, los administradores de empresas constructoras, que debían guardar un dato más que los que guarda la clase User.

Para modelar esto, la mejor solución nos pareció partir la tabla de People en tres tablas distintas: People, User y ConstructorCompanyAdministrator, para así evitar seguir acumulando datos nulos en nuestra base de datos y facilitar el modelo en caso de que se tengan que agregar nuevas herencias en un futuro.

Esto mejora la búsqueda de los datos en cada tabla ya que no debe iterar por datos que no le sirve buscar (por ejemplo no iterar todos los dueños de flats en caso de estar buscando un user).

1. Relaciones en el Domain y webmodels: en el Domain y en los WebModels, en el obligatorio 1, en el caso de una relación de asociación entre dos entidades, normalmente se guardaba el id de la clase asociada como property de la clase relacionada.   
   Haciendo la aplicación cliente, nos dimos cuenta que en general era mucho más directo guardar el id y toda la entidad de la clase asociada, para evitar tener que hacer más de una llamada al servidor para obtener todos los datos de un objeto (primera llamada para obtener los datos del objeto y más llamadas para obtener los datos de sus clases asociadas) y para que así nos facilitara el manejo de la información a nivel de cliente.
2. Mejoramos la mantenibilidad al seguir agregando funcionalidades respetando las abstracciones y capas de separación implementadas a través de interfaces.

# Clean code

Al igual que en la primera entrega buscamos seguir las guías de clean code.

Buscamos hacer código sencillo y evitar la repetición de código, y tratar de seguir lo más posible las recomendaciones de la metodología.

Por nombrar un ejemplo: utilizamos el patrón template method para la generación de los reportes y así evitar la repetición de código. Con el mismo motivo, se utilizó herencia en User, que hereda de Person y herencia en ConstructorCompanyAdministrator, que hereda de User.

Hicimos hincapié en que los métodos tengan nombres descriptivos, en intentar minimizar los números mágicos y que distintos métodos se pasen nulos, e intentamos tener métodos de largo acotado, para hacer el código más legible.

# TDD

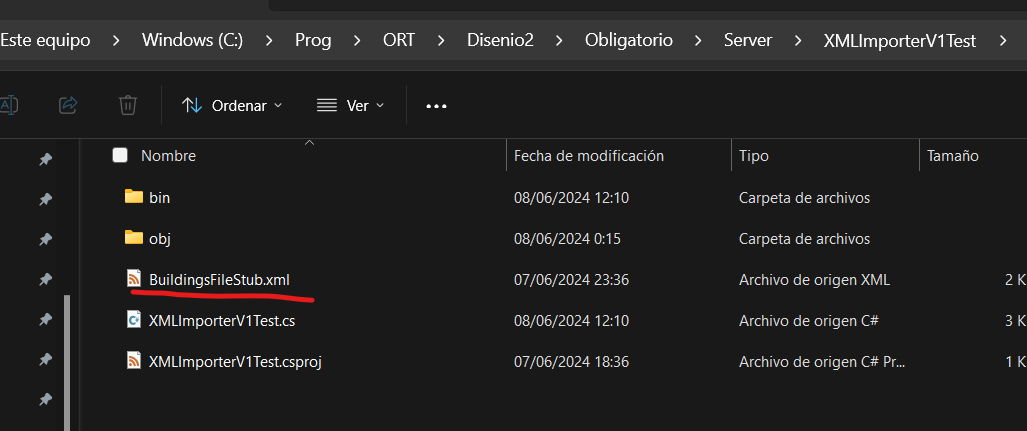
Al igual que como explicamos para el obligatorio 1, el código de este segundo obligatorio se escribió utilizando TDD utilizando Moq para realizar el Mocking de las capas inferiores.

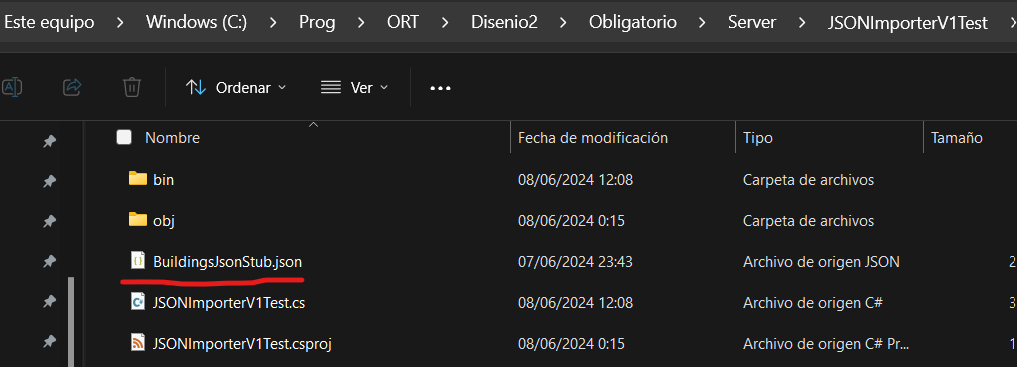
Aparte del uso de TDD para los mismos puntos que se usó en el primer obligatorio, se pidió el uso de TDD para la implementación de los importadores de edificios implementados.

Allí, no fue posible usar la técnica de Mocking porque no era posible mockear el resultado de la interpretación de los resultados, se obtenía un error en tiempo de ejecución.

Lo que se hizo para poder hacer la prueba del caso de la importación fue hacerla con un stub, un archivo xml o json según el importador utilizado, para el que sabíamos cuál sería la lista de DTOBuildings resultado de su importación.

Dichos stubs, a nivel de los archivos de la solución, quedaron en la carpeta del proyecto de pruebas, y deben ubicarse en esa ubicación para que los tests funcionen:

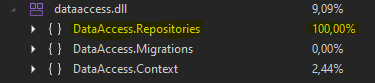
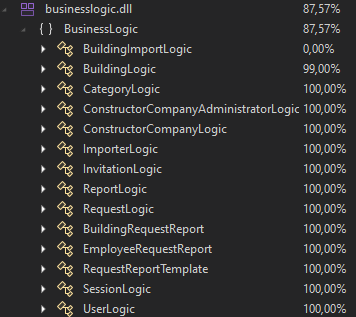
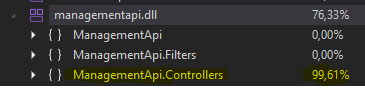




# Reporte de herramienta de cobertura y análisis del resultado

Para analizar el resultado de la cobertura de código, se utiliza la herramienta de análisis de Visual Studio Enterprise, y se obtienen los siguientes porcentajes de los resultados:

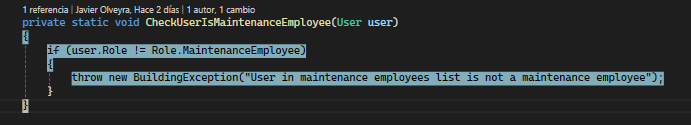
A continuación se puede ver la cobertura de código de los repositorios de acceso a datos y de los controladores de la API:

Como se puede ver, los proyectos BusinessLogic, DataAccess.Repositories y ManagementApi tienen un porcentaje mayor a 99% de cobertura de código, ya que son aquellos donde se implementó la funcionalidad del sistema.

BuildingImportLogic tiene un 0% de cobertura dado a que se corresponde a la clase de la lógica de importación de edificios, para la que no se pidió la implementación con TDD.

Algunos casos en que no se obtuvo 100%, por ejemplo en BuildingLogic, pueden explicarse por bugs en la herramienta de análisis de cobertura, por ejemplo:



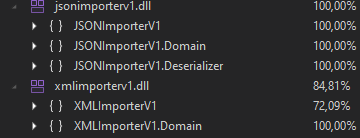
Se puede ver que la última llave está marcada en rojo, por lo que fue tomada como no testeada en la herramienta de análisis, y sin embargo el método CheckUserIsMaintenanceEmployee fue probado.

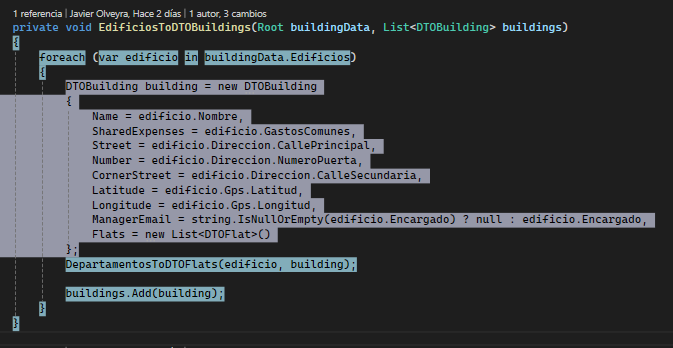
En domain y webmodels no se obtiene tan buen resultado, dado que, en caso de que un atributo no necesitara participar de un test por alguna funcionalidad, no se lo probó específicamente para evitar tener pruebas con muchos asserts innecesarios:

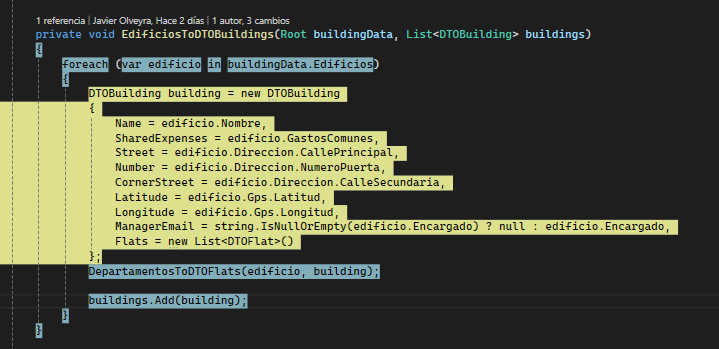




También se utilizó TDD para la implementación de los importadores, aunque no logramos explicar por qué en el caso del importador de XML quedan líneas no cubiertas, aunque en la ejecución del test sí se ejecuta:





En la ejecución del test sí podemos ver que se prueba:

# GitFlow

En cuanto al gitflow, al inicio del proyecto seguimos una estructura de una rama por clase nueva a implementar o fix (implementación a mejorar) en el servidor (back-end) siguiendo la línea de la primera entrega.

En cuanto al front-end como no utilizamos TDD nos designamos una rama llamada Feat-ClientApp en la que mergeamos los cambios hechos al front a medida que los implementábamos. En cuanto a nuevas páginas, creamos una rama nueva por página con el nombre del recurso a implementar (ej: Feat-RequestsPage) y realizamos un commit por implementación de archivo (html o ts).

Ya sobre el final del proyecto, al igual que en la primera entrega, la mayoría de los commits son en base a errores que fuimos encontrando, por lo que si eran básicos los hacíamos en la rama dev para evitar tener muchas ramas con un commit solo y si llegaban a tener algún nivel de complejidad creabamos una rama nueva desde dev.

Decidimos tomar estas decisiones ya que en clase hablamos con los profesores sobre la libertad de las ramas aplicando siempre un trade off que nos resultara eficaz.

# Especificación de la API

Al igual que en la entrega anterior adjuntamos la especificación de la API a un [google sheets](https://docs.google.com/spreadsheets/d/12I_sLp9G_cNsD_9T8nYkWSBXXE1se051/edit#gid=1365849256). En este caso no adjuntamos imágenes ya que la tabla se tornó muy extensa para el límite de cantidad de páginas provistas por la letra.

En la tabla se encuentran destacados con verde los nuevos endpoints de la API correspondiendo a las nuevas funcionalidades pedida por la letra. Además como mencionamos anteriormente se modificaron modelos de entrada y salida de los endpoints que ya existían en la entrega anterior según las modificaciones realizadas.

También adjuntaremos el swagger que se genera al correr el Visual Studio en la parte de documentación junto con el postman.

# Manejo de excepciones

En cuanto al manejo de errores implementamos el ExceptionFilter visto en la clase de tecnología. Este se encarga de cambiar el contexto de la request si existe una excepción que viaja sin ser manejada por nosotros (los programadores). Una vez cambiado el contexto, chequeamos qué tipo de excepción es y enviamos el correspondiente código de respuesta HTTP con el mensaje de la excepción que se largó.

El paquete Filters usa CustomExceptions para conocer las excepciones largadas en la BusinessLogic. Ya que casi todas las excepciones largadas en la BusinessLogic corresponden al código de estado 400 realizamos una herencia de las excepciones largadas según la clase de tipo BusinessLogicException abstracta y así chequear si es de este tipo enviar un 400 Bad Request, ayuda a nuevas implementaciones de errores dentro de BusinesssLogic como fue el caso de las nuevas clases de la lógica para esta entrega, ya que no afecta al ExceptionFilter, el cambio es menor.

Otro caso sería en el que se consulta por un id que no existe en la base de datos, el repositorio correspondiente en este caso largaría una excepción de tipo ArgumentException. El ExceptionFilter para la misma retorna 404 NotFound, ya que se está intentando acceder a un elemento no existente.

El tercer caso sería también al intentar realizar una acción en un elemento no existente en la base de datos, solo que esa acción es el delete. Debido a que este verbo en HTTP es de buena práctica que sea idempotente, debemos largar otra excepción diferente al ArgumentException ya que sino la response tendría código de estado 404. Por lo que utilizamos otra excepción diferente personalizada a la que llamamos DeleteException y es la única en nuestro proyecto CustomExceptions que no hereda de BusinessLogic. Para este caso el ExceptionFilter sobreescribe el resultado del contexto a un ObjectResult con estado 204 NoContent, lo que significa que no realizó acción pero cumple con la request pedida, que era que ese elemento no se encuentre más en la base de datos.

Por último, quedan los errores que no son lógicos. Estos pueden suceder por varias razones pero no controlamos nosotros cuando se largan. A estos el ExceptionFilter larga un 500 InternalError, ya que es un código que le comunica a quien hace la request que falló internamente la aplicación, no por algo que haya realizado el usuario.

# Mecanismo de acceso a datos

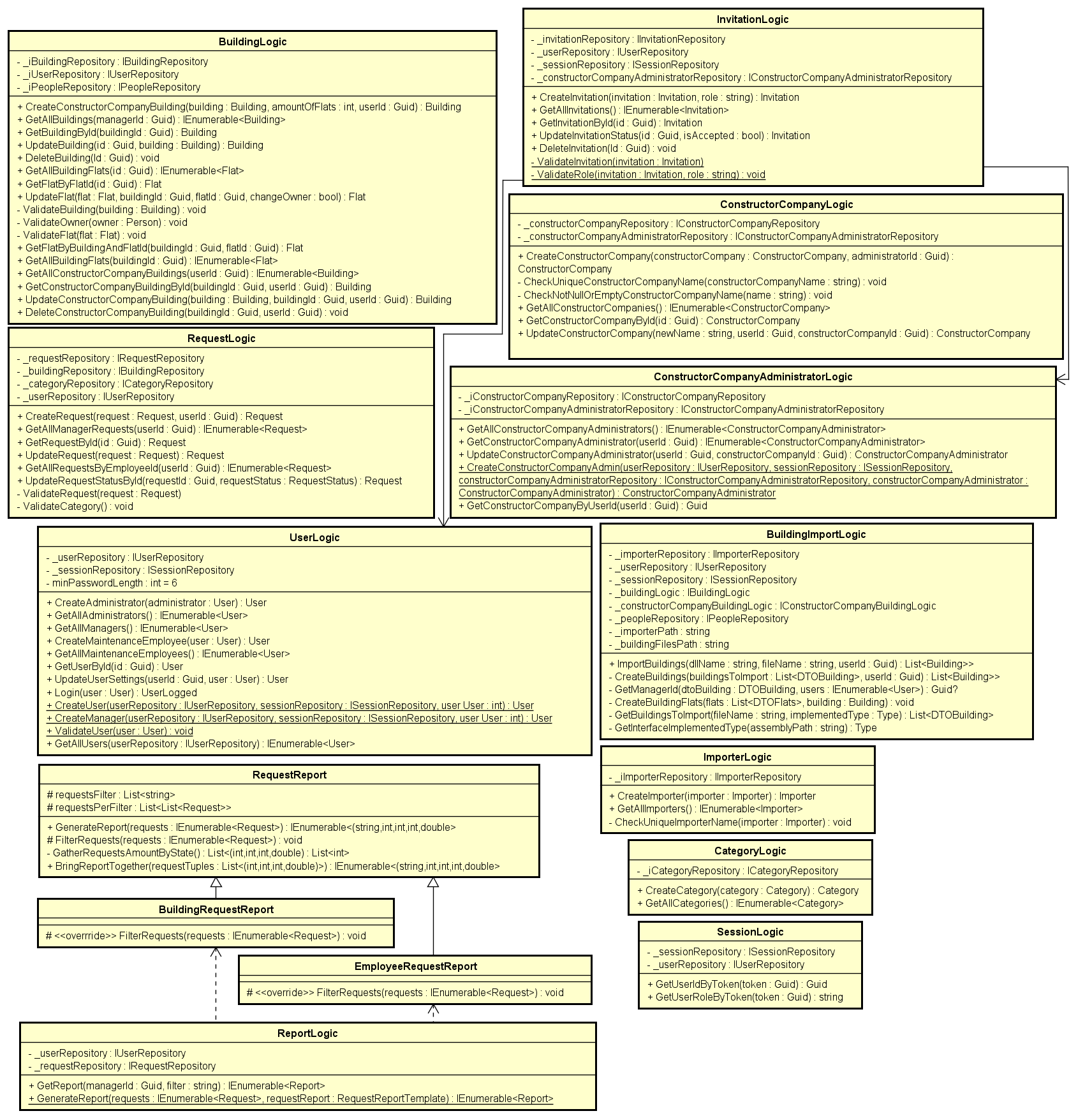
En el proyecto utilizamos Entity Framework para modelar la base de datos. Este requiere tener un contexto que herede de la clase DbContext, proporcionada por el framework, en la que se definen los DbSets a modelar. Además en esta clase se pueden sobreescribir métodos para mejor modelo de las entidades, uno de estos métodos es OnModelCreating por ejemplo.

Una vez creado el contexto de la base de datos creamos repositorios, que como especificamos antes en la sección del proyecto DataAccess, los utilizamos con la funcionalidad más básica posible para que sean “puertas” de entrada y salida de datos , mientras que la lógica se ubica en otra parte del proyecto. Creamos uno por DbSet en el que se encarga de las operaciones del mismo, con la excepción de los building y flats que están altamente relacionados.

# Anexo

## Diagrama de WebModels

## Diagrama de BusinessLogic



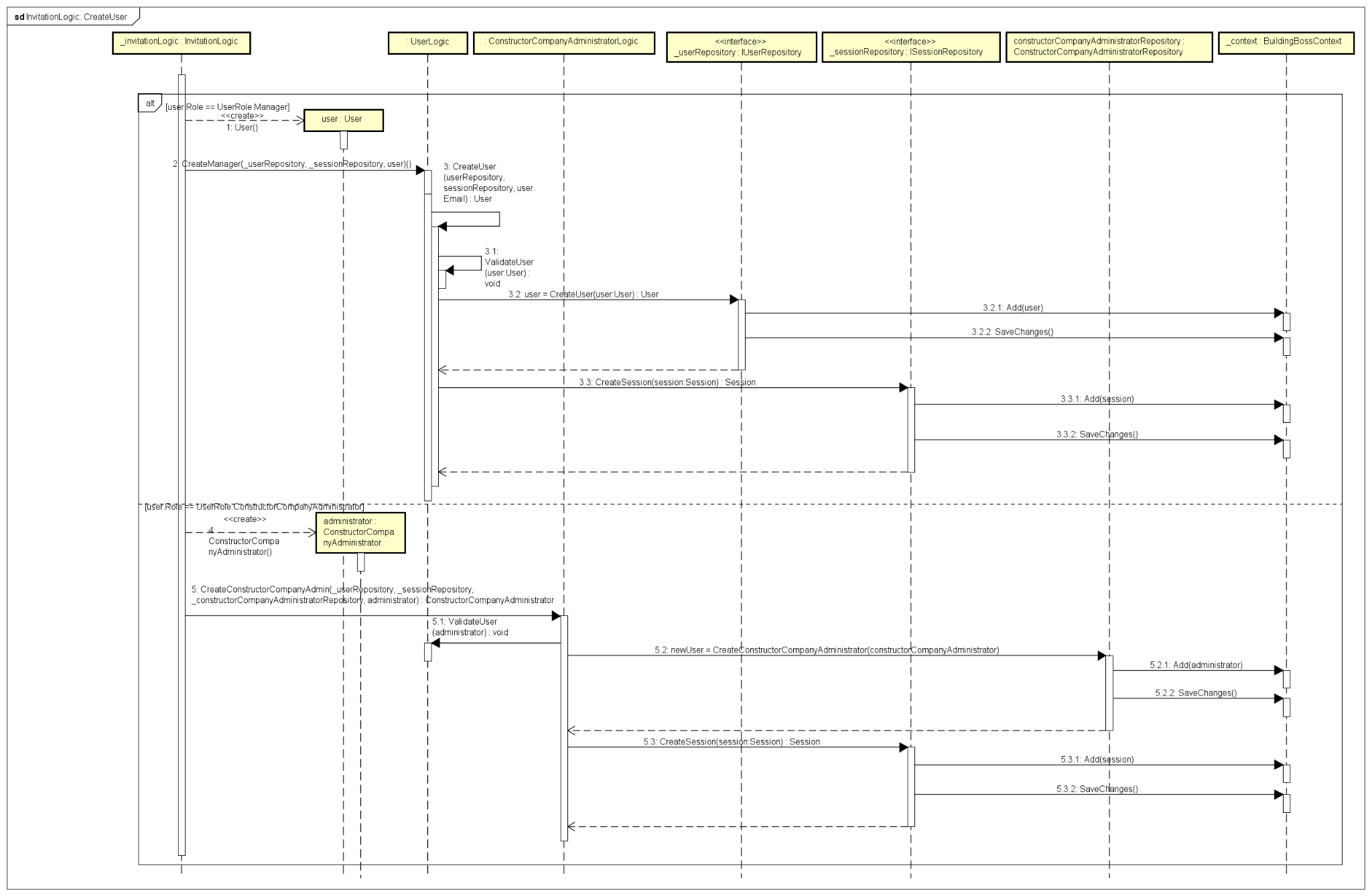
## 

## Diagrama de DataAccess

## 

## Diagrama de secuencia de GetReport

## Diagrama de secuencia de UpdateInvitationStatus



## Diagrama de secuencia de CreateConstructorCompanyBuilding

