Universidad ORT de Montevideo
Facultad de Ingeniería.

OBLIGATORIO 2
Diseño de Aplicaciones1

Facundo Arancet-210696
Marcel Cohen -212426

Proyecto disponible en:
https://github.com/ORT-DA1/Obligatorio1_Arancet_Cohen

Junio, 2018

1 -Descripción general del trabajo y del sistema

El sistema BlueBuilder construido para nuestro cliente, una empresa de construcción de propiedades prefabricadas, es un sistema que permite agilizar el diseño de planos de construcción de las viviendas y la estimación de sus costos. Esta es la segunda versión del sistema.

Resumidamente, el sistema posee los siguientes elementos:

- Perfiles de usuario, a través de los cuales los usuarios interactúan con el sistema.
- Planos, que es el producto que la empresa desea vender. En esta nueva versión, hay planos firmados, que están aprobados para construcción, y no firmados.
- Materiales de construcción, con los cuales se diseñan los planos. En esta nueva versión, las aberturas (puertas y ventanas), pueden tener dimensiones personalizadas.

Esta nueva versión del sistema posee cuatro perfiles con los cuales el usuario puede interactuar con el:

- El perfil de Administrador, que tiene tiene la funcionalidad de dar mantenimiento (Alta, Baja, Modificación), a los demás tipos de usuarios. Está asignado por defecto y no se puede borrar.
- El perfil Diseñador, tiene la funcionalidad de crear y editar planos para clientes.
- El perfil de Arquitecto, este perfil es parte de las nuevas funcionalidades de esta versión. Este perfil tiene la mismas capacidades que un diseñador, pero se le agrega la capacidad de firmar un plano para que quede aprobado para su construcción.
- El Cliente es el cliente de la empresa, es el que compra el plano, y el sistema le permite ver el estado de sus planos con el precio de los materiales para construirlo.

<u>Otras novedades de esta versión:</u>

- Posee persistencia en base de datos, a diferencia de la versión anterior, que almacenaba los datos temporalmente en memoria.
- Existe un nuevo material de construcción, la columna decorativa.
- Existe la opción de ajustar la visibilidad de la grilla en el editor del plano, ahora la grilla puede ser de líneas completas, punteadas o no visible.

Se detallan a continuación, los requisitos del sistema, que fueron validados con el cliente a lo largo del proceso. En nuestro caso pudimos cumplir con la correcta construcción de todas las funcionalidades.

Bugs conocidos:

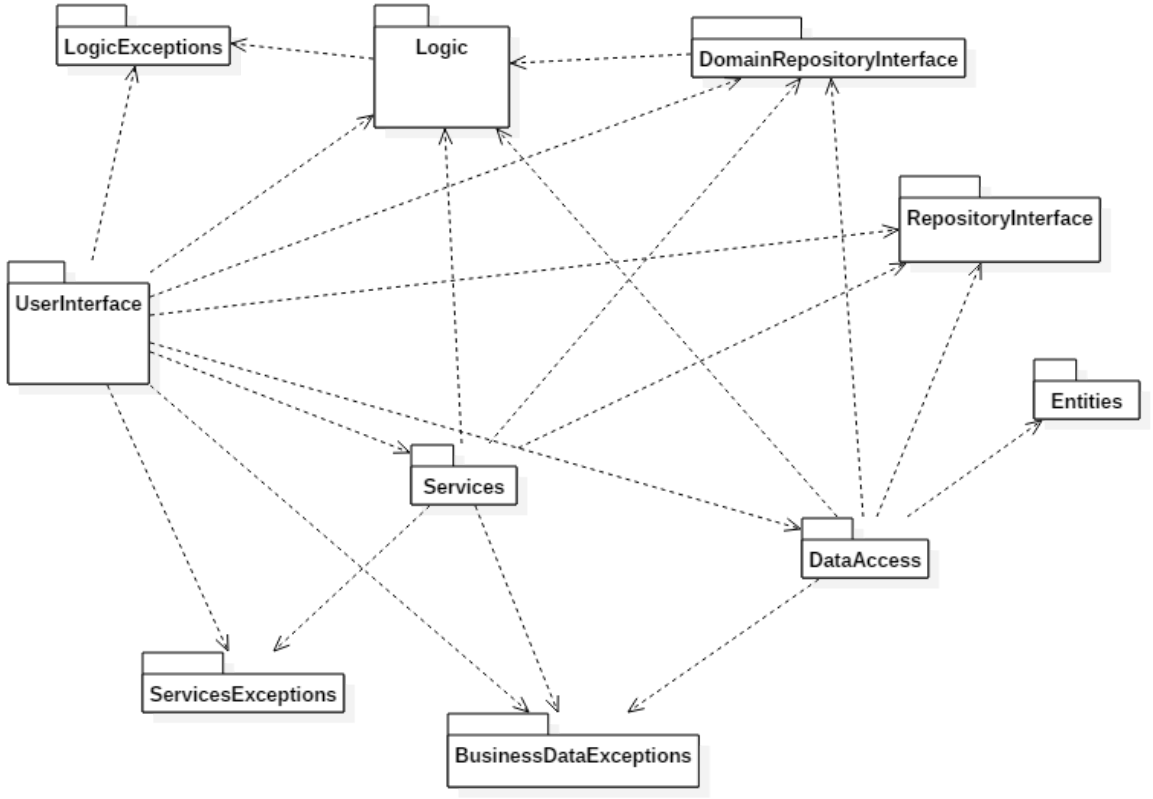
Por otro lado, tenemos en cuenta ciertos errores que ocurrieron durante el proceso de construcción del sistema, los cuales no han sido corregidos.

- Existen commits que: en el comentario no reflejan el cambio realizado en el programa, son poco descriptivos o no son atómicos.
- La interfaz de usuario válida que no se ingresen campos vacíos, es decir, de largo 0. Pero no valida que no se ingresen campos que son espacios en blanco.
- Existen eventos donde se toma el elemento seleccionado de una ListBox. Existe un caso donde no atrapamos la excepción si no se seleccionó ningún elemento.
- En la sección de registro del plano, restringimos las dimensiones del plano a números de 3 dígitos. Si se selecciona un largo o ancho mayor a 300m se puede producir un error crítico en el sistema.
- La clase EditBlueprintView quedó muy extensa debido a todo el código necesario para el dibujado de los elementos de la interfaz. Lo correcto sería extraer todos los métodos de dibujo (PaintWall, PaintBeam, PaintOpening, PaintColumn, etc) a una clase que se encargue de esto. Podría ser interesante implementar esto con el patrón Strategy, de forma que el dibujado de un elemento del plano pueda variar de acuerdo a la estrategia seleccionada desde la interfaz. Esto permitiría la opción de que se pueda configurar el dibujado de los elementos (color, tamaño, forma, etc) a gusto, desde la ui, para cada elemento del plano.

Consideramos que existe otro error, desde el punto de vista de diseño, por que no se implementaron factories para los elementos del dominio, por lo que se encuentran instanciados en muchas clases distintas del sistema.

2- Descripción y justificación del diseño.

Diagrama de paquetes:



Cada paquete lógico fue implementado en un assembly independiente, con el fin de lograr la independencia de librerías. No se incluyeron los paquetes de test en este diagrama, con el fin de reducir la complejidad, y enfocarnos en las funcionalidades de alto nivel, bajo nivel y sus dependencias. De los paquetes de Test se hablará más adelante.

Para que la modificación del código fuente minimice el impacto de cambio nos basamos en 2 patrones:

El primero, es el patrón GRASP de bajo acoplamiento. El minimizar la dependencia entre los paquetes asegurara que la repercusión de un cambio sea mínima.

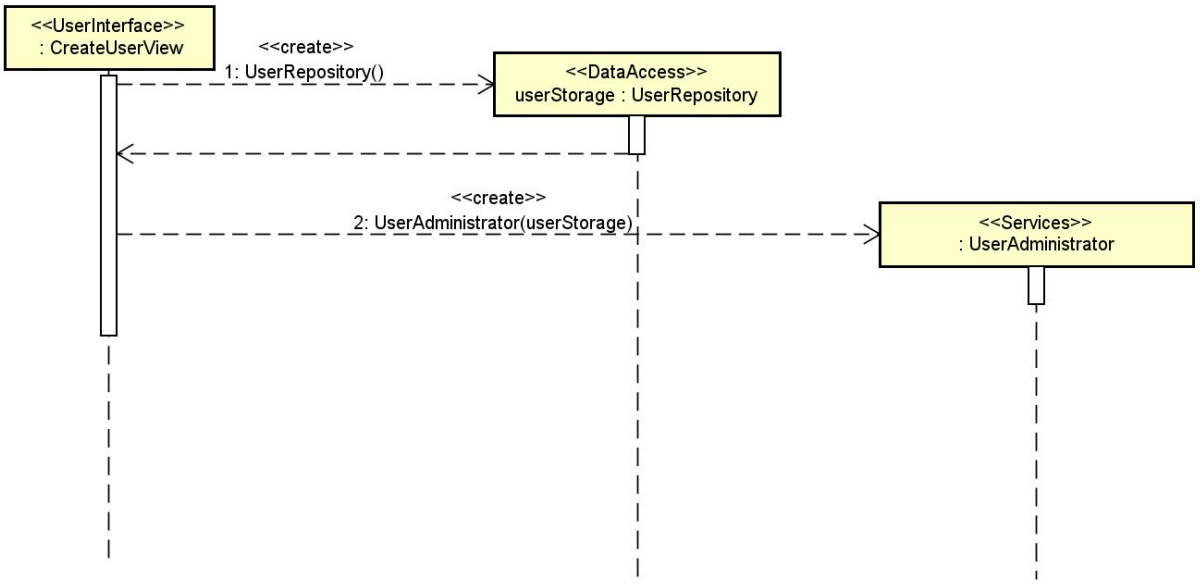
El segundo, es el principio de inversión de las dependencias.

Este patrón, básicamente, establece que los módulos que implementan funcionalidades de alto nivel, es decir, los que implementan la lógica del negocio, no pueden depender de los módulos que implementan funcionalidades de bajo nivel, es decir, la infraestructura.

En el curso se leyó un texto llamado “A Little Architecture” de Robert C. Martin, donde se define que la clave para una buena arquitectura, es que la lógica del negocio no conozca los detalles de la implementación, como la presentación y el acceso a datos. También establece que la lógica del negocio es dueña de las interfaces (las define) que los módulos de bajo nivel deben implementar.
Del segundo principio, surgió la decisión de crear las librerías RepositoryInterface y DomainRepositoryInterface. Al principio, la idea fue tener una librería con todas las interfaces que las clases del módulo de acceso a datos iban a implementar, pero se observó que existen interfaces que conocen las clases del dominio y hay una interfaz genérica IRepository<T>, que no depende del dominio. Entonces se optó separar esta interfaz en una librería separada, con el fin de disminuir el acoplamiento, ya que este nuevo paquete no depende del dominio.

Con estas librerías de interfaces, se logró que el paquete de los servicios, que son los que se encargan de controlar los eventos del sistema, y son de alto nivel , no dependiera de DataAccess, que es la librería donde está la implementación del acceso a datos (que como dicho antes, es de bajo nivel). Esto se logró mediante la inyección de dependencias, Services solo puede referenciarse a objetos de DataAccess mediante su abstracción, pero no puede ser este quien los instancie,si fuera así, los servicios dependerían de DataAccess y se violaría el principio. Entonces, es la interfaz la que se encarga de instanciar los objetos de data access y se los pasa en el constructor a los servicios que esta crea (esa es la denominada inyección).Como la interfaz gráfica implementa una funcionalidad de bajo nivel, no hay problema en que dependa del paquete de acceso a datos.

Con este diagrama de secuencia se muestra en un ejemplo, como las clases de GUI instancian el repositorio y se lo pasan por parámetro al service que instancian. El service utiliza la abstracción de este repositorio.



Listado de clases por paquete:

Se detalla a continuación la lista de clases por paquete. Se agruparon los paquetes en 2 categorías: módulos de alto nivel y de bajo nivel, para aclarar la idea desarrollada anteriormente.

Módulos de alto nivel:

Logic:

- Admin
- Architect
- Beam
- Blueprint
- BlueprintCostReport
- BlueprintPriceReport
- Client
- Column
- ComponentType
- Constants
- DataValidations
- Designer
- Door
- IBlueprint
- IComponent2D
- IComponent3D
- IMaterialType
- IPermissible
- ISinglePointComponent
- MaterialContainer
- NullUser
- Opening
- Permission
- Point
- PunctualComponentPositioner
- Session
- Signature
- Template
- User
- Wall
- WallsPositioner
- Window

LogicExceptions:

- CollinearWallsException
- ColumnInPlaceException
- ComponentInWallException

- ComponentOutOfWallException
- EmptyTemplateNameException
- InvalidDoorTemplateException
- InvalidTemplateDimensionException
- InvalidTemplateException
- InvalidTemplateTypeException
- OccupiedPositionException
- OutOfRangeComponentException
- TemplateNotMatchException
- WallsDoNotIntersectException
- ZeroLengthWallException

RepositoryInterface:

- IRepository

DomainRepositoryInterface:

- IBlueprintRepository
- IPriceCostRepository
- ITemplateRepository
- IUserRepository

BusinessDataExceptions:

- InaccessibleDataException
- InconsistentDataException
- TemplateAlreadyExistsException
- TemplateDoesNotexistException
- UserAlreadyExistisException
- UserNotFoundException

Services:

- BlueprintController
- BlueprintEditor
- BlueprintReportGenerator
- CostsAndPricesManager
- OpeningFactory
- SessionConnector
- UserAdministrator

ServicesExceptions:

- InvalidComponentTypeException
- NoPermissionsException
- WrongPasswordException

Módulos de bajo nivel:

LogicTest:

- AdminTest

- ArchitectTest
- BeamTest
- BluePrintTest
- ClientTest
- ColumnTest
- ComponentsContainerTest
- DataValidationsTest
- DesignerTest
- OpeningTest
- PointTest
- SessionTest
- TemplateTest
- WallTest
- WindowTest

DataAccess:

- BlueBuilderDBContext
- BlueprintAndEntityConverter
- BlueprintRepository
- MaterialAndEntityConverter
- OpeningTemplateRepository
- PriceCostRepository
- UserAndEntityConverter
- UserRepository

DataAccessTest:

- BluePrintsRepositoryTest
- CostPriceRepositoryTest
- OpeningTemplateRepositoryTest
- UsersRepositoryTest

Entities:

- AdminEntity
- ArchitectEntity
- Blueprintentity
- ClientEntity
- ColumnEntity
- CostPriceEntity
- DesignerEntity
- OpeningEntity
- OpeningTemplateEntity
- SignatureEntity
- UserEntity
- WallEntity

ServicesTest:

- BlueprintControllerTest
- BlueprintEditorTest
- BlueprintReportGeneratorTest
- CostsAndPricesManagerTest
- OpeningFactoryTest
- SessionConnectorTest
- UserAdministratorTest

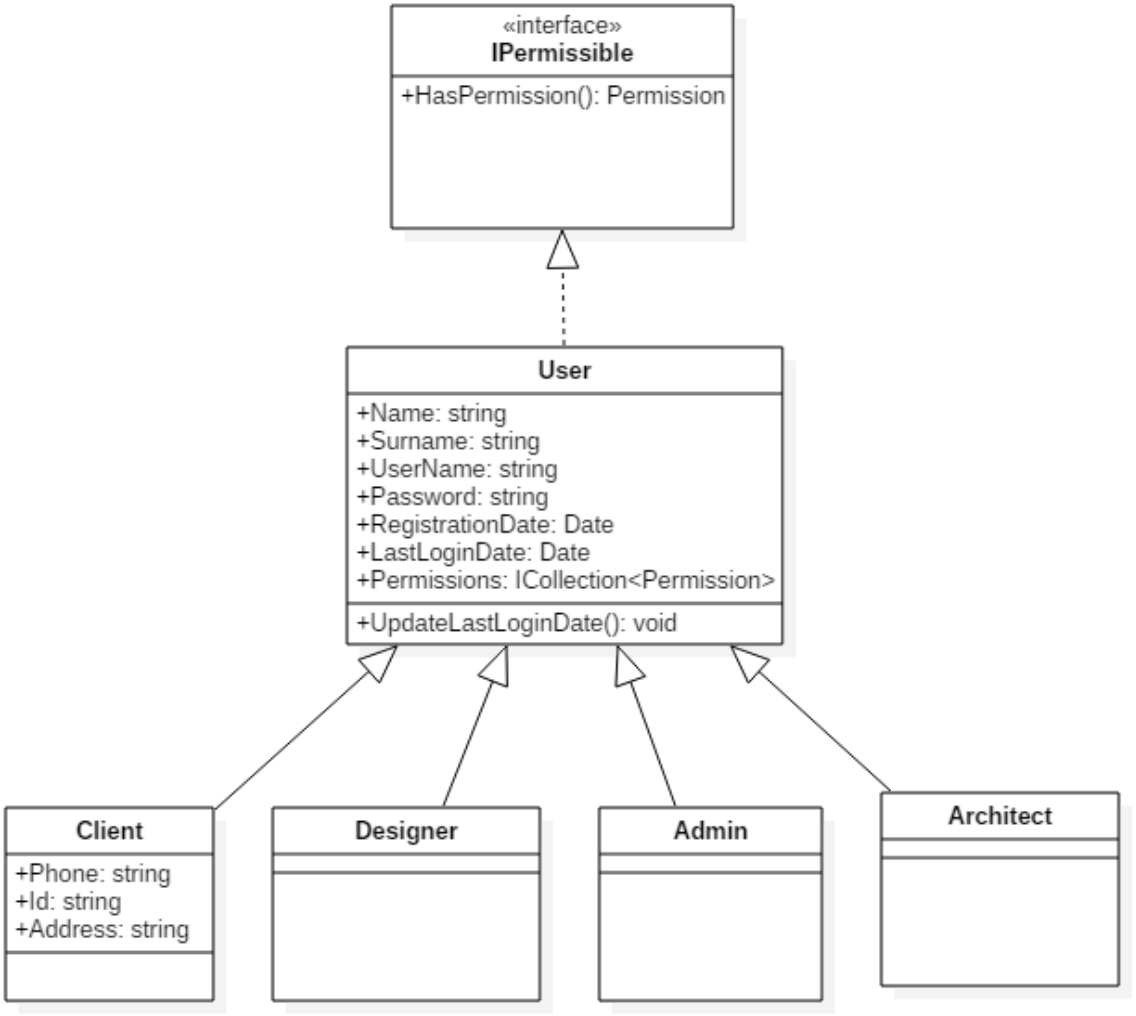
UserInterface:

- AdminUserManagement
- ButtonCreator
- ChooseBlueprintView
- CompleteLineGridPaint
- CreateBlueprint
- CreateTemplate
- CreateUserView
- DottedLineGridPaint
- EditBlueprintView
- GenericHome
- GridPaintStrategi
- InputValidations
- IUserFeatureControl
- LoggedInView
- MainWindow
- ManageCostsView
- NoFlickerPanel
- NoPaintedGridLineStrategy
- PointConverter
- Program
- UserDataVerificationView

Modificaciones en la funcionalidad Alta Baja y Creación de usuarios

Incorporación del usuario Arquitecto al sistema

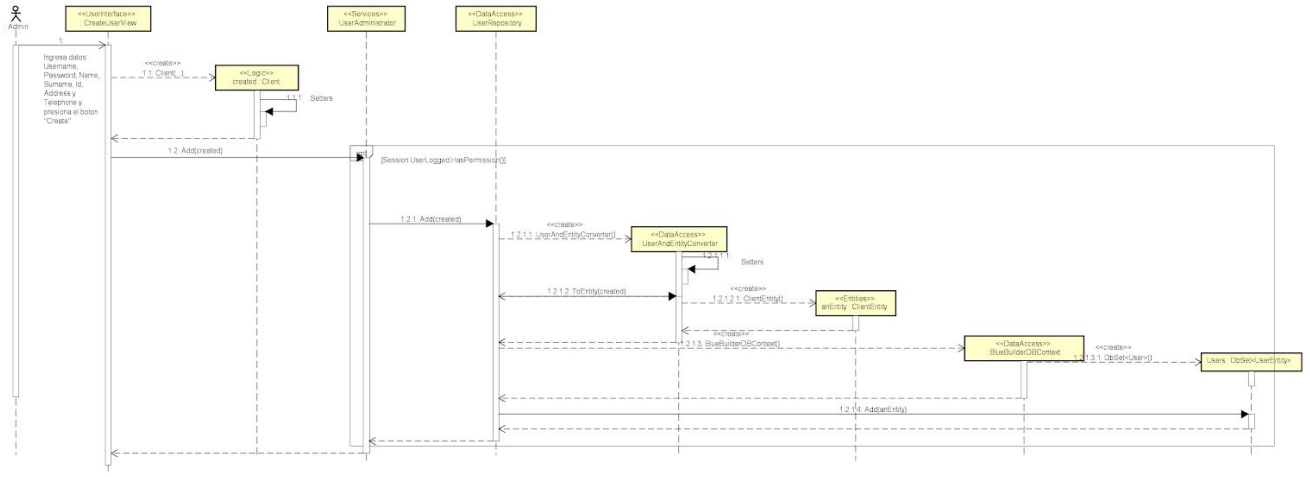
El cliente solicitó que se incorporara en el sistema un nuevo tipo de usuario, el arquitecto, que tiene las mismas capacidades que un diseñador, y además tiene la función de firmar planos, que están aprobados para la construcción.



Consideramos que gracias al diseño de la versión anterior, la repercusión de este cambio fue mínima, en base al tiempo y cambios en el código fuente que nos tomó hacer esta modificación.
Se agrego la clase Architect como subclase de user, que bien pudo haber sido subtipo de Designer, pero como en nuestro sistema, el que sea subtipo de usuario o de diseñador no tiene mucha diferencia. A este objeto se le concedieron todos los permisos de diseñador y uno nuevo, “CAN_SIGN_BLUEPRINT”, que fue incorporado al Enum de Permissions. En base a este permiso, el servicio de firmas valida que sea un arquitecto el que pueda firmar un plano. También, como en la versión anterior, la interfaz gráfica presenta las funcionalidades del sistema dependiendo de los permisos que tenga el usuario ingresado.Entonces en la interfaz se muestran las opciones de firmar plano y demás en base a los permisos que tiene este nuevo tipo de usuario. Cómo architect es un usuario, se guarda con el resto de los usuarios y el login lo trae por el nombre de usuario, como al resto, no hubo necesidad de hacer identificación del tipo en el login, y con la estrategia de permisos, no se pregunta por el tipo,sino por el permiso. De esta forma se evita la identificación de tipos en tiempo de ejecución (Runtime Type Identification).

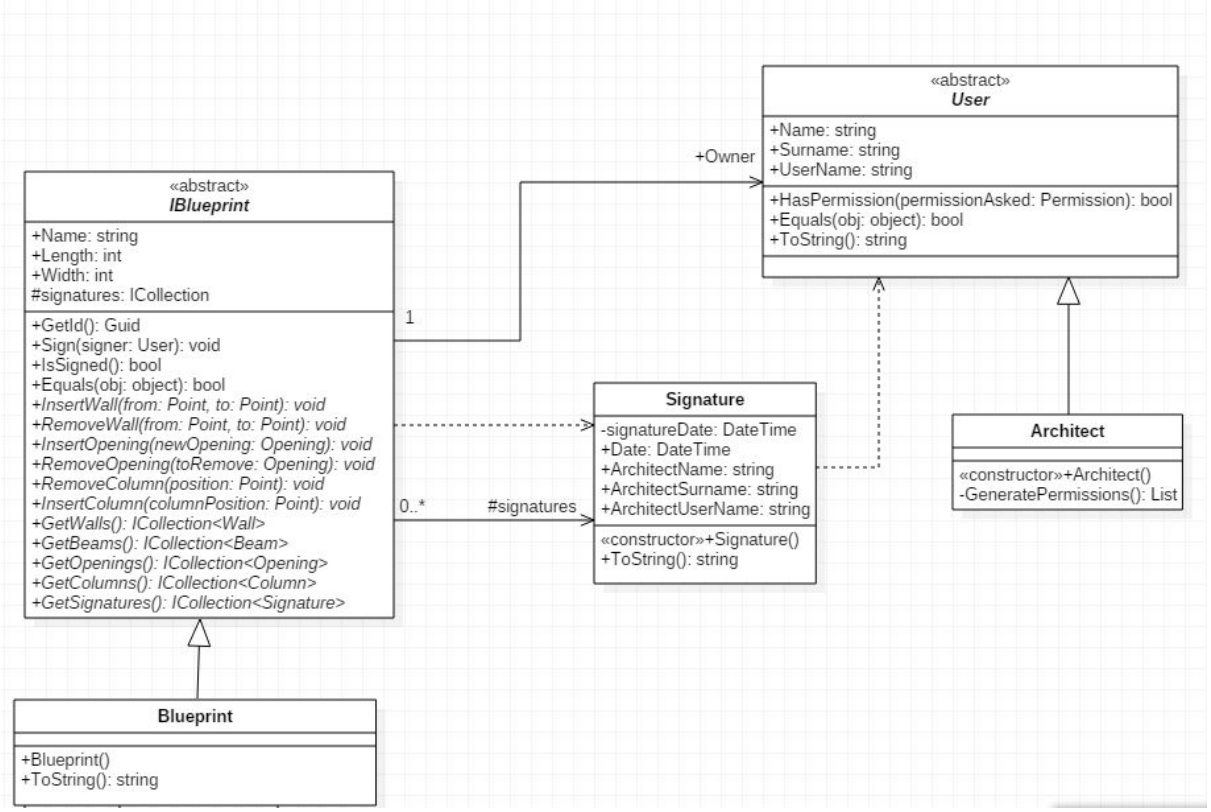
Diagrama de secuencia de creación de un cliente

A continuación se muestra el diagrama de secuencia de la funcionalidad crear cliente. Como se puede observar, la interfaz interactúa con el service UserAdministrator. Esto es una aplicación del patrón Controller, del grupo de patrones GRASP. Este patrón propone el uso de clases controladoras, encargadas de manejar los eventos del sistema, en otras palabras, atiende a la interfaz gráfica, de esta forma se tiene una mejor separación entre la lógica del negocio y la de presentación. Se tomó la decisión de dividir los eventos del sistema en distintos controladores, para aumentar la cohesión y disminuir el acoplamiento. Nótese también, que es la interfaz la que instancia el repositorio y se lo pasa por parámetro al servicio, que se acopla a una abstracción de repositorio, de esta forma se evita la dependencia de los servicios a la implementación del acceso a datos.



En los proximos diagramas de secuencia que se verán en el sistema, se puede apreciar cómo se rigen por el patrón Controller mencionado, y la interfaz accede al dominio por medio de los controllers, que se encuentran en el paquete Services.

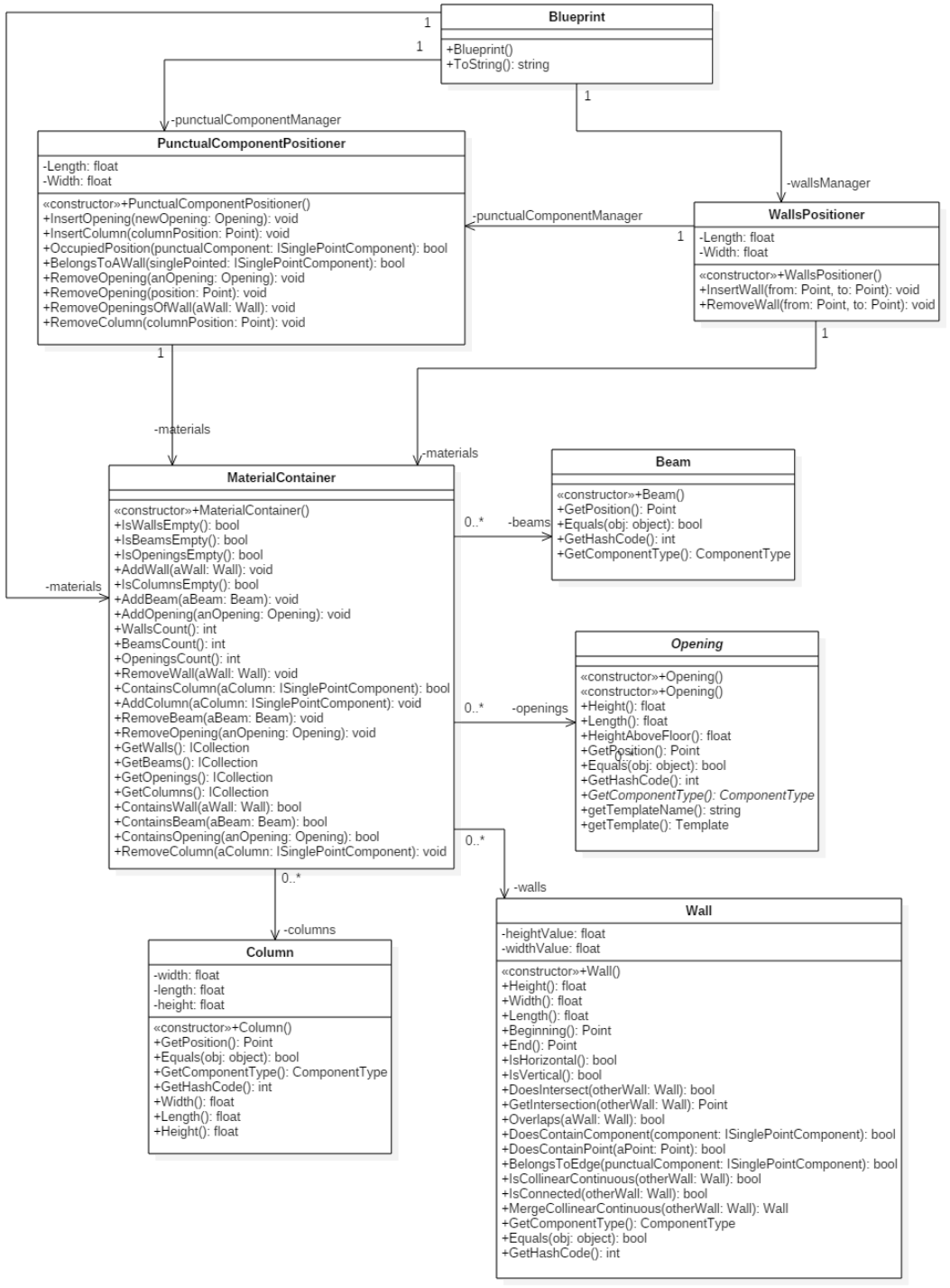
Modificaciones sobre Blueprint



Para implementar la nueva funcionalidad de firmas requerida, se decidió hacer cambios en la interfaz IBlueprint. La misma, pasó a ser una clase abstracta debido a que su objetivo ya no era solamente abstraer el comportamiento de los planos, sino que también describir cómo están formados.
La (ahora) clase abstracta IBlueprint es utilizada por todas las clases que necesiten manipular información de planos, tanto para leerlos como para editarlos.
Para la implementación de las firmas, se agregó una nueva clase que las modele, llamada “Signature”. Luego, en IBlueprint se añadió una lista de esta nueva clase para que sea posible la existencia de tantas firmas como uno quiera.
Se decidió modelar las firmas como una clase (concreta) porque esto facilitaba la persistencia y el cumplimiento del requerimiento de varias firmas sobre un mismo plano. Signature es una estructura de datos que contiene la información del arquitecto y la fecha en la que fue creada. Entendemos como “firma” a este par de datos.
Se almacenan tres atributos para representar la información del arquitecto. Se decidió representar la información de esta forma en la firma para facilitar la persistencia de la misma. Si se elimina un arquitecto, sus firmas persisten para mantener la información de que alguien alguna vez firmó ese plano.

Almacenar el nombre de usuario del arquitecto es suficiente para poder obtener el resto de su información desde la base de datos. Sin embargo, mantener el nombre y apellido del arquitecto en la clase Signature nos permite sobrescribir el método toString(), el cual resulta de utilidad para mostrar las firmas en la ui.

Debido a que la clase Blueprint estaba quedando muy extensa y con demasiadas responsabilidades, se crearon clases nuevas a las cuales se les delegó la lógica de posicionamiento de elementos en un plano.

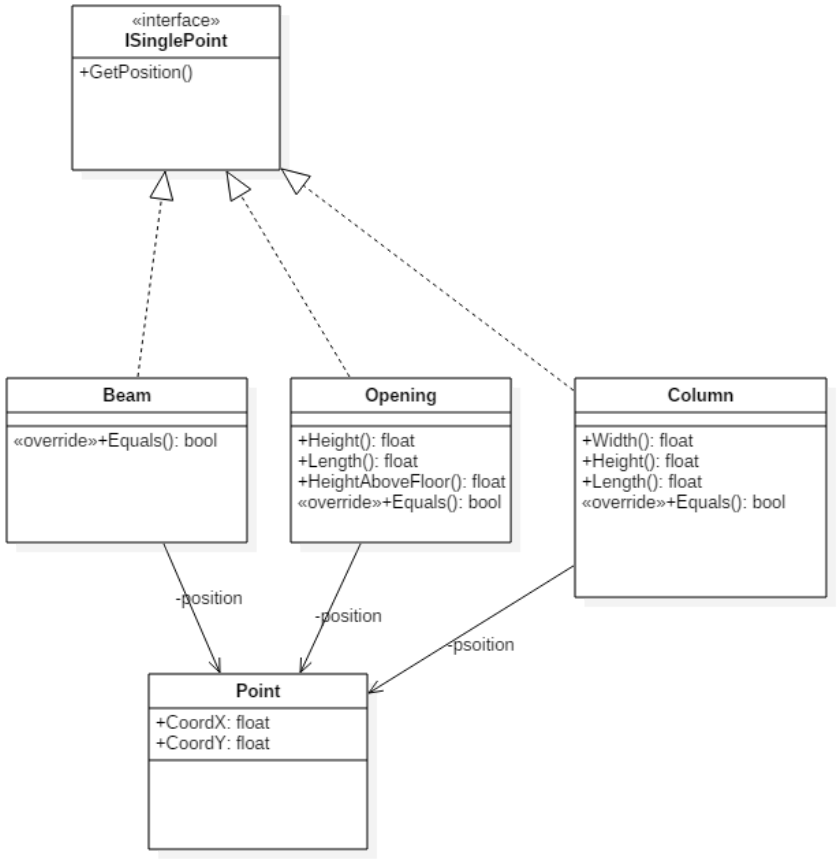


PunctualComponentPositioner se encarga de la colocación y eliminación de elementos que implementan la interfaz ISinglePointComponent (elementos puntuales como una abertura o columna). Por otro lado, WallPositioner se encarga de la colocación y eliminación de paredes específicamente.

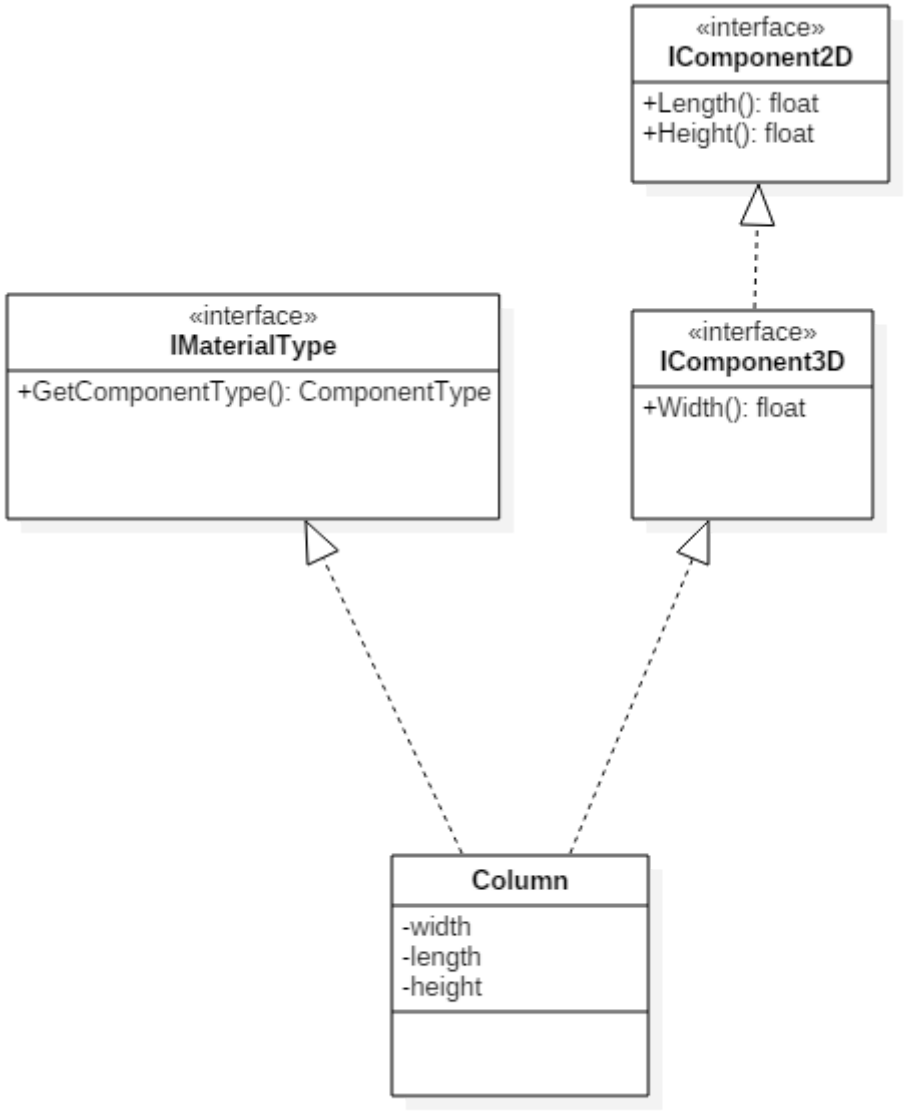
Una posible mejora de este diseño sería agregar una interfaz Positioner<T>, que obligue a implementar métodos para que dado un plano, se agreguen y remuevan los elementos T de ese plano. Deberíamos exigir que el tipo genérico de datos T implemente la interfaz IComponent2D de nuestro paquete dominio para que pueda ser dibujable. Debido al corto tiempo que se dispone para la elaboración de este sistema y a que no existe una cantidad considerable de elementos dibujables que justifiquen un diseño de este estilo, no se implementa esta idea pero se tiene en cuenta para una extensión de los requerimientos a futuro.

Modificaciones en las funcionalidades de creación y edición de planos

Incorporación del material “Columna decorativa”



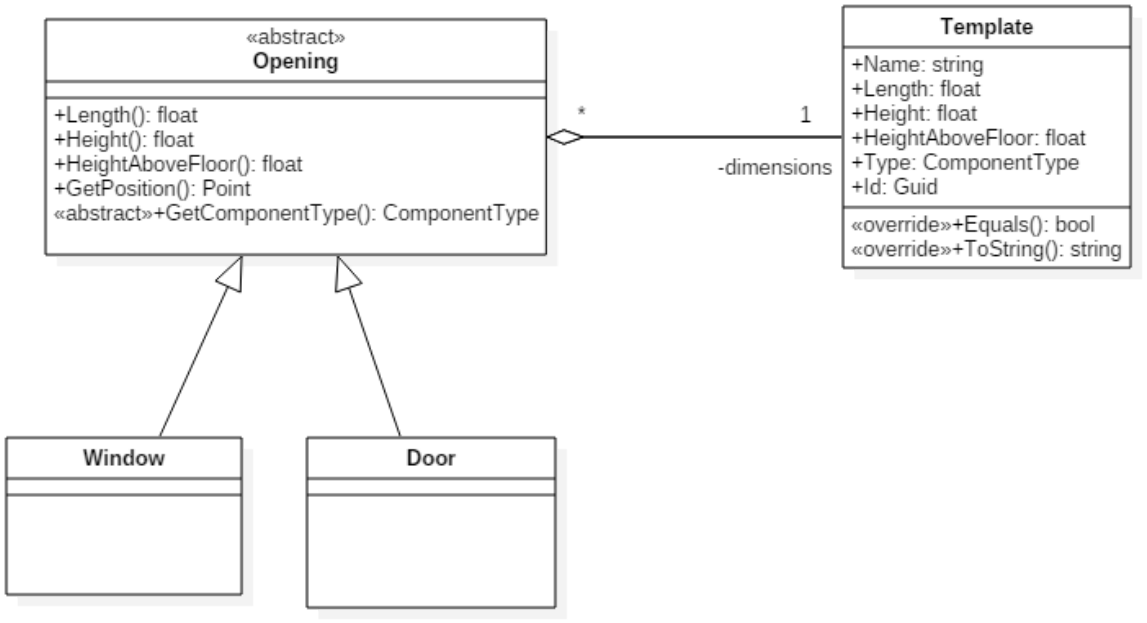
Se creó la nueva clase Column, que implementa la interfaz ISinglePoint, al igual que beam y opening, ya que su representación interna es dada por solo un punto. Esto es ventajoso ya que se pueden aprovechar validaciones en el plano que utilizaban elementos ISinglePoint, como por ejemplo la función:
bool DoesContainComponent(ISinglePointComponent component)
de la clase Wall que devuelve true si contiene un componente puntual dentro de él, (que su ubicación está dentro de la pared), esta se usó para validar si la columna colocada esta dentro de una pared.
Cómo Column implementa dicha interfaz, esto también implica que tienen que devolver un objeto de la clase Point, y por eso depende de esta clase como el resto.



Como los demás materiales en nuestro sistema, este debe proveer su tipo de material y sus dimensiones. Por eso se acopla a las interfaces IMaterialType y IComponent3D que imponen estos comportamientos. Se concluyo que el modelado de la interfaz de dimensiones no fue útil en el sistema, pero se decidió mantener por que consideramos

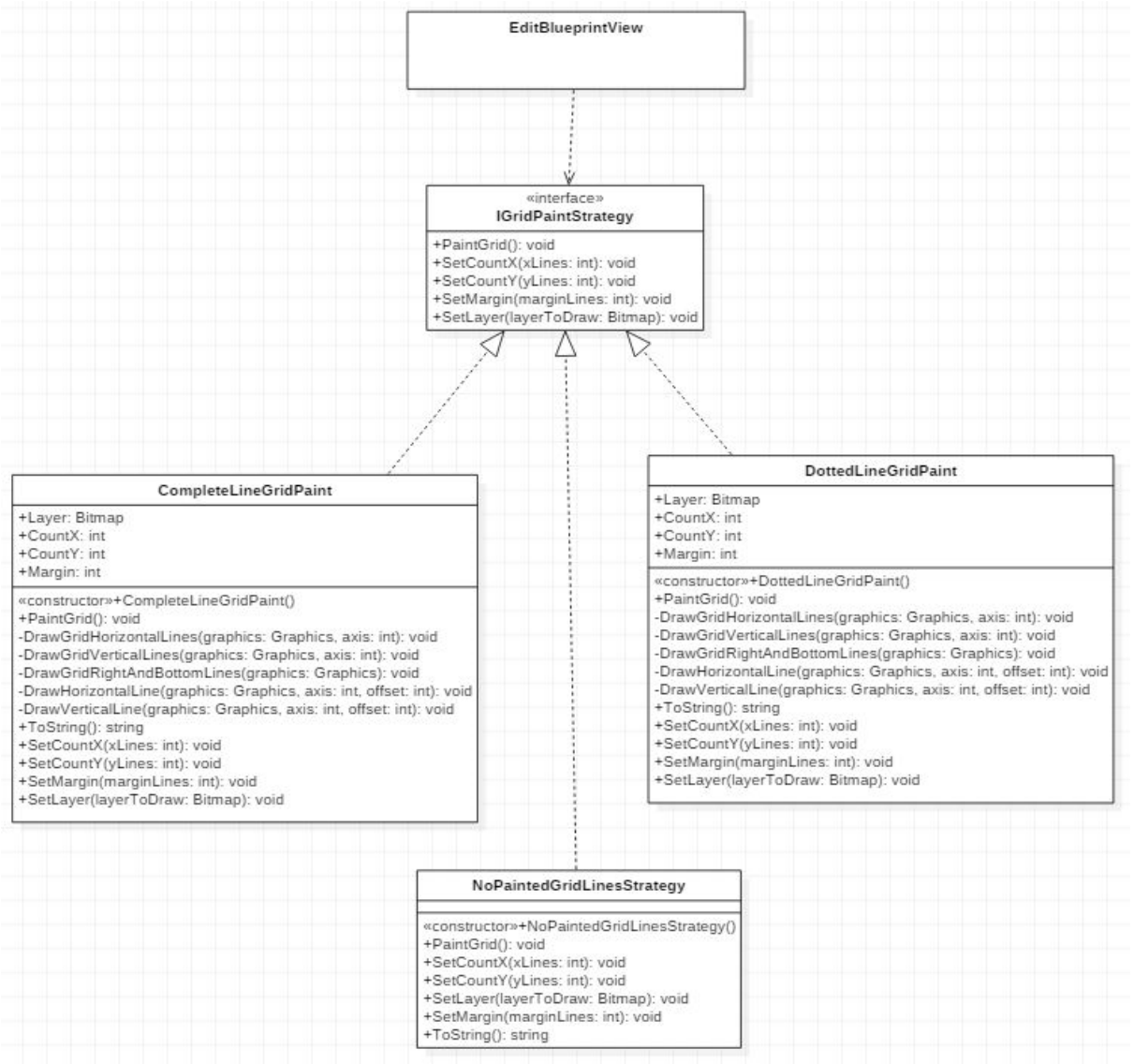
que a futuro puede existir un potencial cliente que le interese realizar alguna operación sobre los materiales basándose en sus dimensiones, porque creímos que la altura de los materiales y el grosor, en el caso de las paredes, iba a tener algún uso en el sistema.

Incorporación de aberturas de largo variable



Para esta versión del sistema, el cliente se solicitó la incorporación de aberturas de dimensiones variables (solo largo y altura, no poseen ancho). Estas aberturas serían de tipo puerta o ventana, no se incorporan nuevos tipos de abertura.
Para modelar esto, se extrajo el “Template” de la abertura de la clase abertura, utilizando composición delegación. Modelamos este objeto con la clase Template, es una clase sin comportamientos que almacena datos como, las dimensiones y un nombre que va a distinguir ese modelo particular de abertura,como “Porton” o “Ventanal deslizante”.Nótese que también conoce el tipo del componente de construcción que es, en el ejemplo anterior “PUERTA” y “VENTANA”, esto fue necesario porque el template tiene que validar sus dimensiones según las reglas del negocio, por ejemplo: Si es una puerta, su altura sobre el piso debe ser 0, si es abertura, su altura sobre el piso no puede superar los 2 metros. Adicionalmente, no puede existir una ventana con un template de puerta y viceversa, entonces eso también tuvo que ser validado.
La ventaja de este modelo, es que las aberturas ya no son el mismo concepto que fueron el diseño pasado, antes eran materiales de construcción, independientes el uno del otro, como el resto. En este diseño, son múltiples instancias de templates ya existentes, que tienen una posición en el plano, y esto posibilita que los templates existan en el sistema independientemente y puedan ser almacenados, para que el administrador o diseñador los pueda seleccionar y colocar una abertura con ese template en el plano.

Pintado de grilla



Para implementrar este requerimiento se tomó la decisión de aplicar el patrón de diseño Strategy. Cada estrategia será una forma de pintar la grilla mientras que el contexto donde se utiliza es el panel EditBlueprintView. Usando la interfaz GridPaintStrategy, obligamos a quienes la implementen a implementar los métodos necesarios para el dibujado de la grilla.
Supondremos que la estrategia ya tiene acceso al Bitmap donde se dibujará la grilla al momento de ejecutar PaintGrid(), por eso no se lo pusimos como parámetro del método. Como se ve en el diagrama, las estrategias tienen el bitmap como atributo.

La interfaz será utilizada por cualquier clase que desee pintar una grilla sobre un Bitmap

Tomamos la decisión de usar este patrón de diseño porque observamos que el contexto EditBlueprintView necesitaba ejecutar un algoritmo de pintado de grilla dependiendo de la selección del usuario. Al modelar la solución así, podemos crear un combo box, cargarlo con elementos de tipo IGridPaintStrategy y obtener de la selección actual del combo cual será el algoritmo de pintado de grilla a utilizar. Hacemos uso del método toString de cada clase que implementa la interfaz para identificar en el combo box cada estrategia. Los beneficios de implementar un strategy aca son que podremos agregar nuevas estrategias de pintado fácilmente (basta con crear una nueva clase que implemente la interfaz y definir los métodos de la misma) y que se facilita la selección de las mismas.

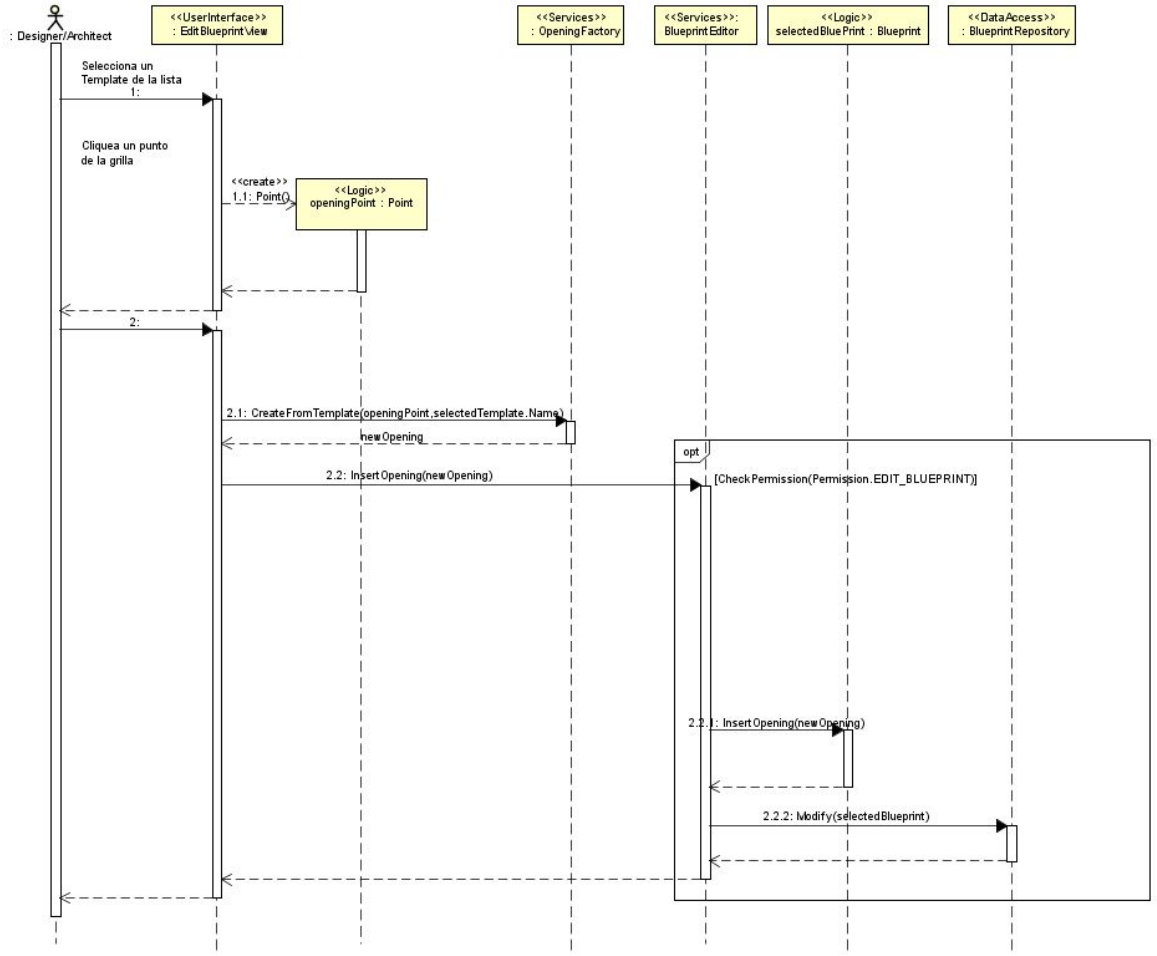
Otra forma de implementar esto podría haber sido definiendo los métodos de pintado de grilla en el mismo EditBlueprintView y llamarlos cuando la opción seleccionada sea la correspondiente. Esta forma de diseñar la solución, en comparación con la anterior es mala porque no permite la extensibilidad de la lógica de dibujado.

EditBlueprintView

Esta clase sufrió cambios debido a que tras implementar los nuevos requerimientos la misma superaba las 800 líneas de código, siendo esta cifra demasiado grande, incluso para una clase de ui. Para realizar el refactoring, se generaron tres nuevas clases a las cuales se les asignó tareas que tenía EditBlueprintView. La perimer clase generada fue Drawer, su responsabilidad es la de delegar acciones referentes al dibujo. Se agregaron dos estructuras de datos Pencilcase y LayersStructure. La primera sirve para guardar objetos de tipo Pen que son utilizados para el dibujo de los diferentes elementos del plano. La segunda sirve para almacenar las diferentes capas de bitmaps en las que se dibuja. Por último se agregó la clase PointConverter que tiene la responsabilidad de saber cómo traducir objetos de tipo System.Drawing.Point (utilizados por los control para representar puntos) a Logic.Point (puntos utilizados por nuestro sistema para representar objetos en el plano). Todas las clases anteriores son utilizadas por Drawer para cumplir con su tarea.
Las diferentes estrategias de dibujado de grilla son utilizadas por Drawer para lograr el dibujo de la misma.

Servicio para la creación de Openings

Para esta nueva entrega del sistema, se agregó una factory de Openings. El motivo de la misma es el de poder crear Openings a partir de Templates. dado el nombre de un template, la factory busca en la base de datos si existe y crea la opening correspondiente a partir de estos datos (cuando existen). Sirve también para encapsular el switch de CommponentType para la creación de las openings.
En el diagrama de secuencia a continuación, se muestra el flujo del sistema cuando se agrega una abertura, se puede observar la función de la factory.

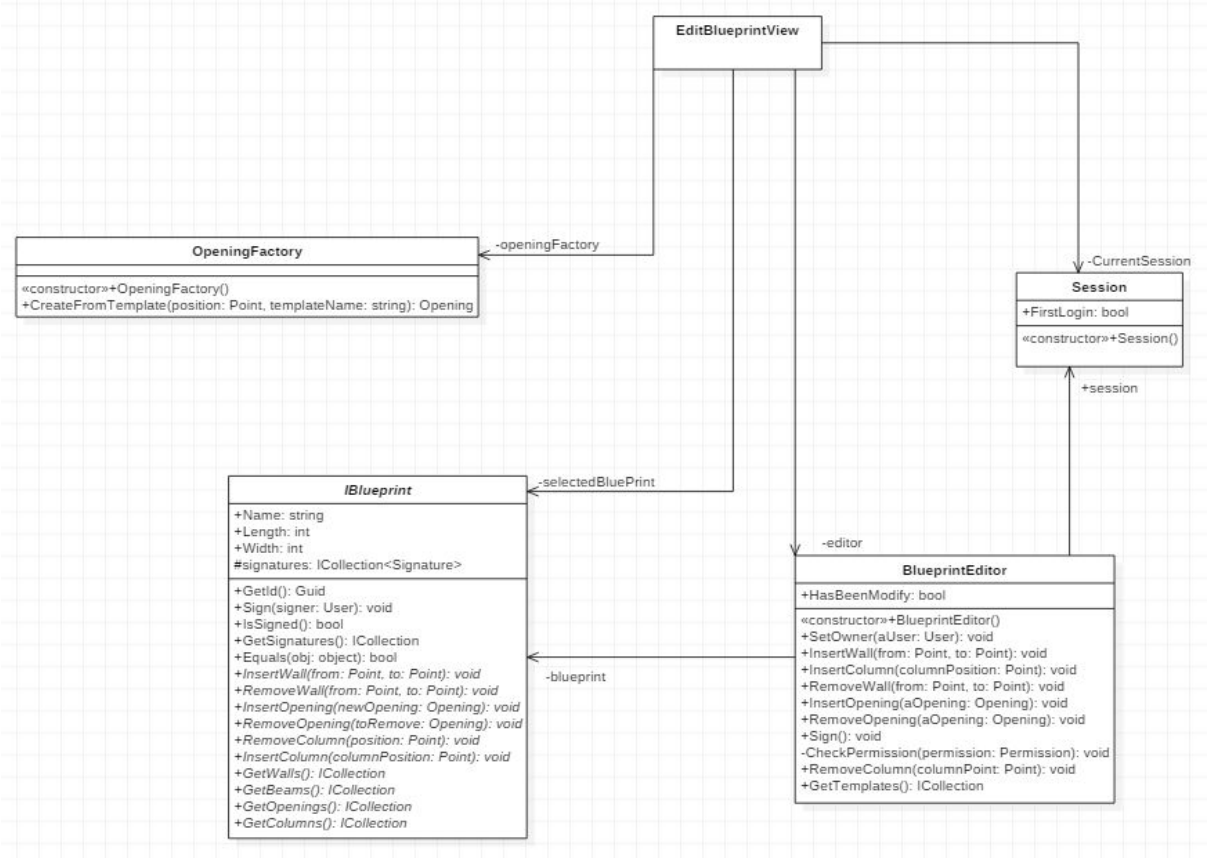


Obligatoriedad de la firma

Si un arquitecto modifica un plano que ya estaba firmado, su firma quedará en el plano. Esto se controla en el evento Leave() del formulario de edición de planos. Existe una bandera en el servicio BlueprintEditor que controla que se inicializa en false y si el plano controlado por este BlueprintEditor es modificado, se pone en true. Entonces, si el arquitecto modificó el plano, y sale del formulario de edición, este se firmará automáticamente.
Esto también se controla si el arquitecto cierra el programa, debido a que EditBlueprintView suscribe una función que checkea esto al delegate FormClosing del formulario padre, del panel padre.

Dado lo anterior, se cubren todas las salidas de edición del plano controladas por el arquitecto (no se controla que quede la firma si el arquitecto apaga la computadora por ejemplo, pero esto escapa a nuestro alcance).

Servicio BlueprintEditor

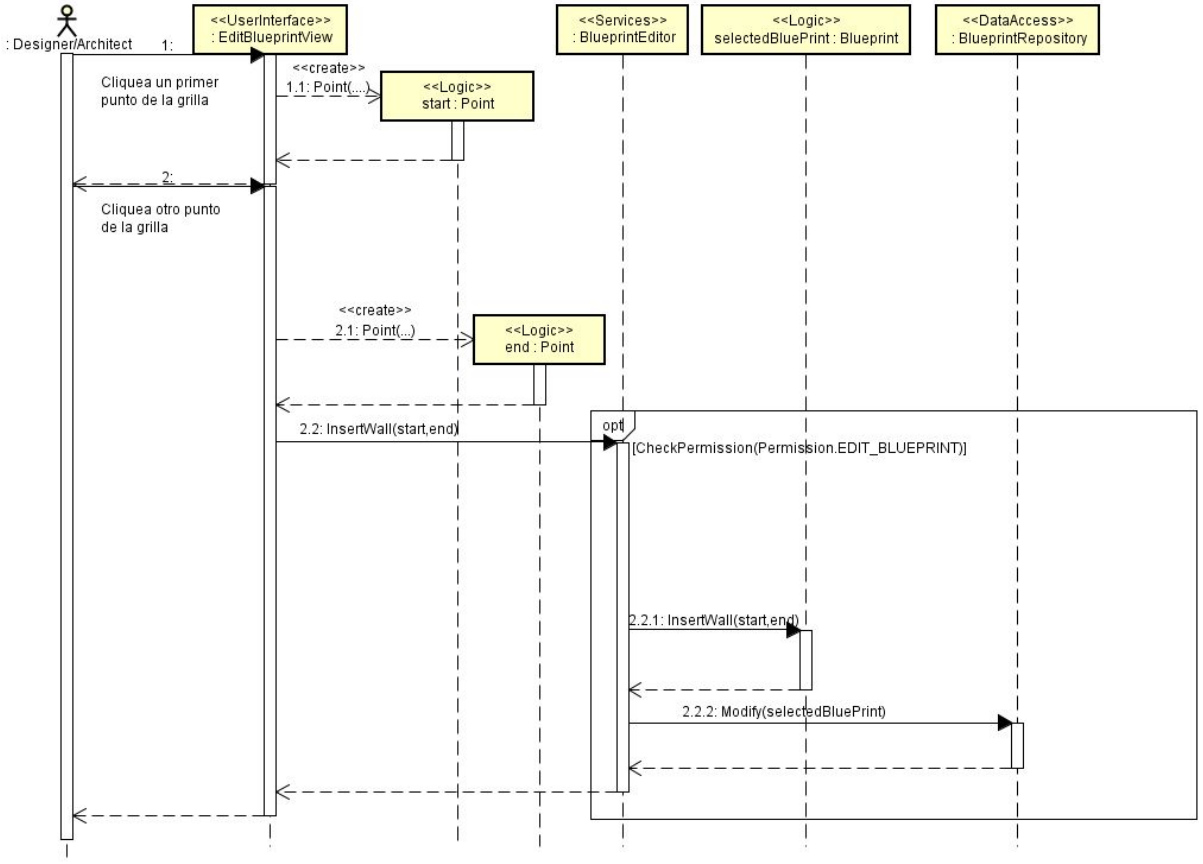


Vimos la necesidad de implementar un servicio que maneje la edición de planos. De esta forma se reduce el acoplamiento entre ui y lógica, mientras que se aumenta el acoplamiento entre ui y servicios. Entendemos la primera como algo no necesariamente malo pero no deseable para nuestro diseño, dado que queremos controlar ciertos permisos y restricciones de negocio utilizando dichos servicios. Utilizar servicios nos sirve para minimizar el impacto de los cambios en la lógica. Observamos que al finalizar todos los cambios sobre la lógica de planos e incluso la implementación de un acceso a datos diferente, el (ahora inexistente) BlueprintView seguía funcionando correctamente. Esto demostró que el impacto de los cambios es mínimo en nuestro diseño. Se decidió remover BlueprintView y solo dejar EditBlueprintView para evitar código repetido. EditBlueprintView muestra los paneles y botones que corresponda dependiendo de los permisos del usuario que esté logueado a él.

El servicio BlueprintEditor, además nos permite controlar permisos estrictos sobre la edición de planos (hace falta poner los métodos de modificación del blueprint internos al paquete Logic).

BlueprintEditor es útil pero además necesario. Veamos que nosotros queremos poder controlar que un usuario tenga permisos para la manipulación de los datos almacenados. Esto quiere decir que no queremos que en la ui se acceda a la base de datos desde el repositorio, queremos que sea a través de una clase controladora como lo es BlueprintEditor.

Finalmente, una utilidad que nos brinda este servicio es la bandera HasBeenModify que permite saber si el editor fue utilizado. Esto es lo que usamos para obligar a un arquitecto a firmar el plano cuando abandona el área de trabajo. Si la bandera está en true, esto quiere decir que el plano fue editado y por lo tanto el arquitecto debe dejar su firma. Si la bandera está en false, quiere decir que el arquitecto entró al panel de edición pero no modificó nada, por lo que no necesitará dejar su firma.
El diagrama a continuación muestra, con un ejemplo, la función del servicio en el flujo de la edición del plano.



Excepciones

En esta sección se detalla sobre las 3 librerias de excepciones que tiene el sistema:

- LogicExceptions: Son las excepciónes de las entidades del dominio en si, básicamente, son las excepciones lanzadas por los setters de los objetos y por el plano cuando se ingresan materiales de forma inválida.
- ServiceExceptions: Son las excepciones relativas a los servicios, como la no tenencia de permisos para cierta operación.
- BusinessDataExceptions: Son excepciones genericas relativas al acceso a datos, como nombre de usuario repetido, usuario inexistente, Datos inaccesibles o corrompidos.

Listado de excepciones por paquete:

LogicExceptions

Nombre de Excepción	Descripción	Clases que la arrojan
CollinearWallsException	La excepción es arrojada cuando se intenta ingresar una pared que se superpone con otra.	Wall WallsPositioner
ZeroLengthWallException	Esta excepción se produce cuando se intenta crear una pared cuyo principio y fin coinciden.	Wall
WallsDoNotIntersectException	La excepción es arrojada cuando se quiere obtener el punto de intersección entre 2 paredes que no se intersectan	Wall
OutOfRangeComponentException	La excepción es arrojada cuando la pared no pertenece al dominio del plano, sea por que esta fuera del rango o por no ser vertical u horizontal	PunctualComponent Positioner PunctualComponent Positioner
ComponentOutOfWallException	La excepcion se arroja cuando se intenta ingresar una abertura o una viga fuera de una pared que la contenga	PunctualComponent Positioner

ColumnInPlaceException	La excepción se arroja cuando una pared colocada esta ocupando el lugar de una columna	WallPositioner
OccupiedPositionException	La excepción se arroja cuando una posición en la que se quiere insertar una columna, esta ocupada.	PunctualComponent Positioner
InvalidTemplateException	Excepción genérica de tempates, se implemento para que todas las excepciones de template heredaran de esta y poder capturar la genérica desde la UI, en una sola sentencia catch	Template
EmptyTemplateNameException	La excepción se arroja cuando el template tiene nombre vacío	Template
InvalidTemplateDimensionException	La excepción se arroja cuando alguna de las dimensiones del Template creado viola alguna regla del negocio	Template
InvalidDoorTemplateException	La excepción se arroja cuando se intenta instanciar un template de puerta con una altura sobre el suelo mayor a 0	Template
InvalidTemplateTypeException	Se arroja cuando se quiere instanciar un template de un tipo	Template

	que no sea ni puerta ni ventana.	
TemplateNotMatchException	Se arroja cuando se quiere crear una opening con un template que no es del mismo tipo que esta	Opening

ServiceExceptions

Nombre de Excepción	Descripción	Clases que la arrojan
InvalidComponentType	La excepción se arroja cuando se trata de crear un opening con un tipo que ni siquiera es de opening	OpeningFactory
WrongPasswordException	La excepción es arrojada cuando se intenta ingresar con un usuario existente pero una contraseña incorrecta	SessionConnector
NoPermissionException	La excepción es arrojada cuando se intenta acceder a una funcionalidad sin los permisos suficientes	Todas las clases controladoras

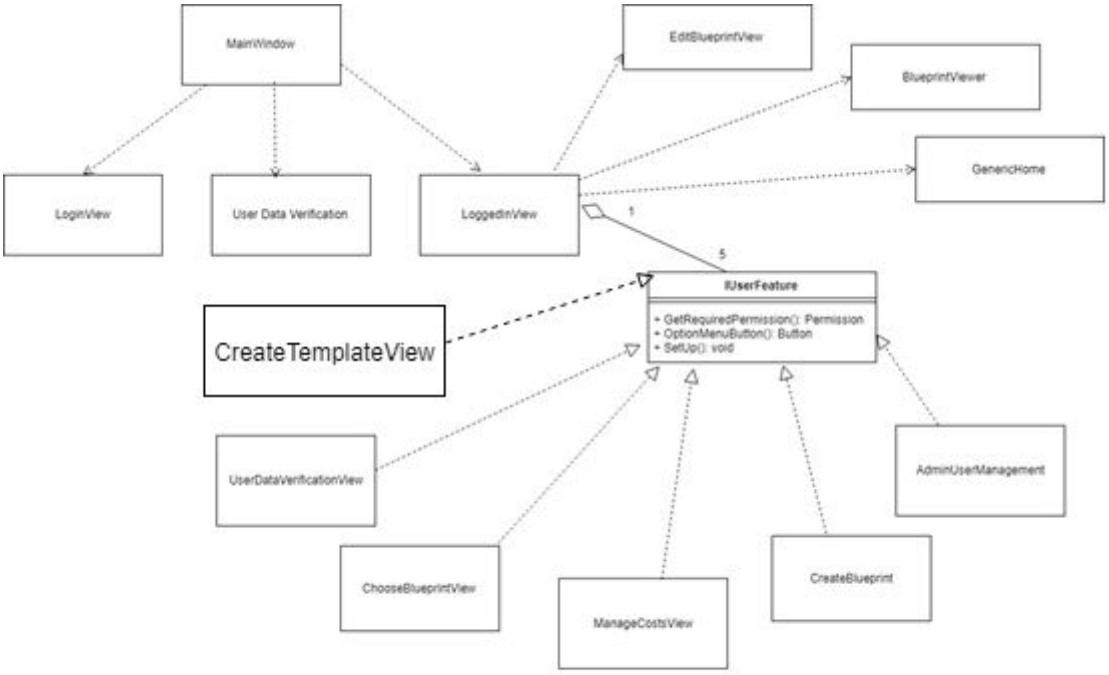
DataAccesExceptions

Nombre de Excepción	Descripción	Clases que la arrojan
UserNotFoundException	La excepción es arrojada cuando se busca un usuario que no existe	UserPortfolio

UserAlreadyExistisException	La excepción es arrojada cuando se quiere agregar un usuario que ya existe	UserPortfolio
TemplateAlreadyExistsException	La excepcion se arroja cuando se quiere ingresar un template con un nombre repetido.	OpeningTemplateRepository
TemplateDoesNotExistException	La excepcion se arroja cuando se quiere accede a un template que no existe	OpeningTemplateRepository
InaccesibleDataException	La excepcion se arroja cuando se pierde el acceso a los datos	Todos las clases repositorio
InconsistentDataException	La excepcion se arroja cuando los datos almacenados son inconsistentes con la logica de negocio, posiblemente fueron manipulados externamente	Todas las clases repositorio

Modificaciones en la interfaz gráfica

Dejando de lado las clases Strategy que se describieron anteriormente, cabe destacar que el diseño de ventanas de la version pasada respondio muy bien a la modificacion. El diagrama a continuación representa la estructura, y las relaciones entre los objetos en nuestro paquete UserInterface.



Como se puede ver, lo único incorporado al diagrama de ventanas, es el panel CreateTemplateView (que se puede ver resaltada del resto) , que permite al arquitecto crear templates para aberturas. El cambio simplemente consistió en crear el panel con su lógica, implementar la interfaz IUserFeature, y agregarla a la colección de paneles de LoggedInView, con eso ya se logró que el panel exista dentro de las opciones.Al implementar la interfaz IUserFeature, se le asigno el permiso requerido para acceder a ella, que es el permiso que tienen los diseñadores y arquitectos. Otros cambios a nivel de interfaz, fue agregar más componentes al panel de edición de planos, ya que adquirió nuevas funcionalidades. También se realizó la modificación de agregarle lógica al panel que muestra los planos para mostrarle los planos firmados y no firmados a cada usuario acorde a sus permisos, y también indicar si los planos fueron firmados y mostrar la lista de firmas si lo fue.

Base de datos y Entity Framework

Un requerimiento que surge para esta entrega del sistema es la persistencia. Para la misma se utilizó el paquete Entity Framework (EF) que provee Microsoft para el mapeo de la objetos a entidades de bases de datos.

A continuación describiremos las decisiones que tomamos para que la implementación de EF no violara el Dependency Inversion Principle.
Para implementar EF debemos definir clases que se comporten como entidades para el sistema. Estas entidades deben tener atributos que serán mapeados a la base de datos por medio de migraciones, las cuales dejamos en automáticas porque observamos que este modo mapeaba las entidades de la forma que queríamos. EF exige que las entidades tengan sus atributos públicos.

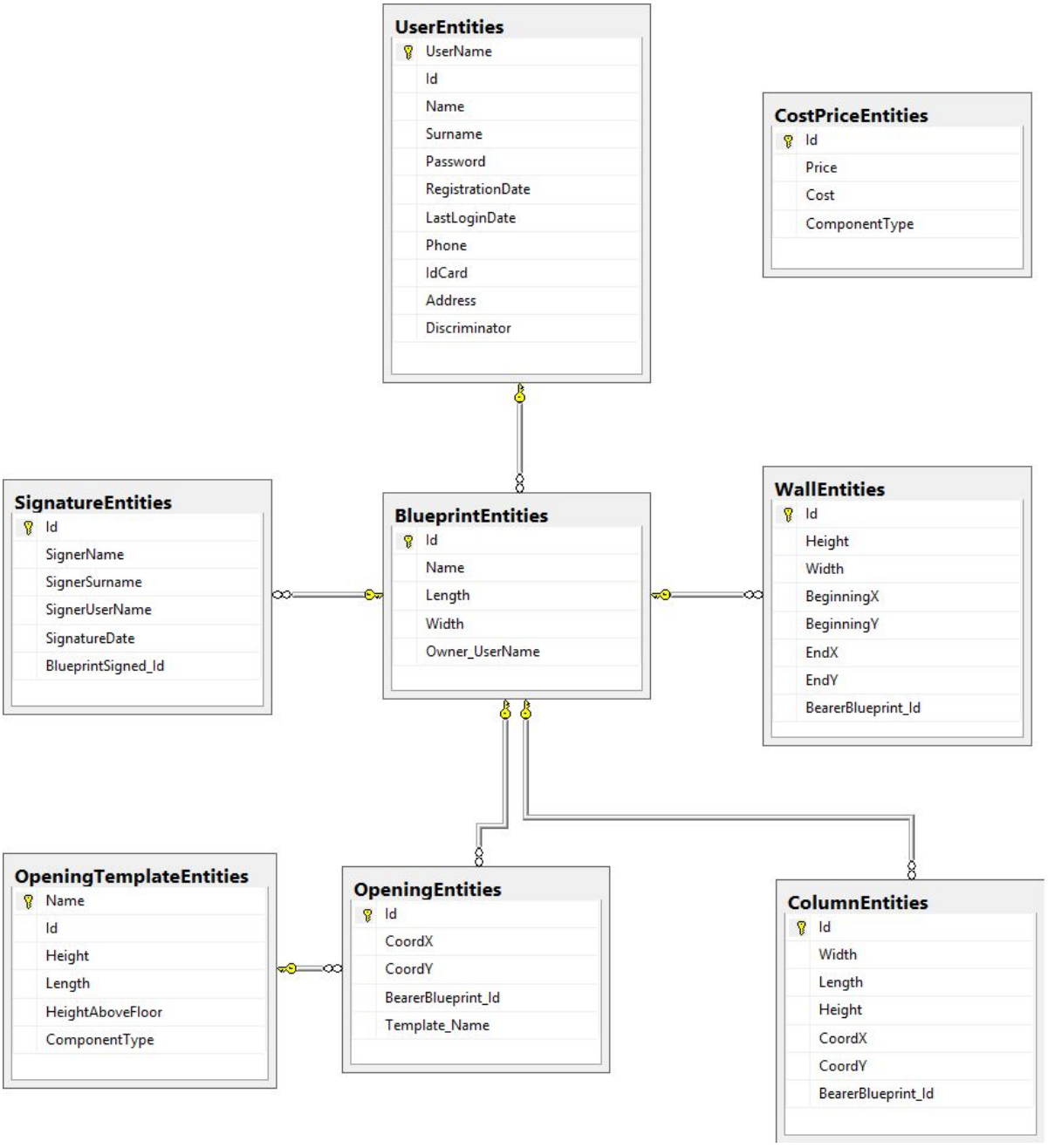
Existen dos formas de implementar las entidades:

La primera es forzar que los objetos del dominio se comporten como entidades de EF. De esta forma nos ahorramos toda la lógica (no trivial) del mapeo de objetos del dominio a entidades y viceversa. Asignando un campo GUID a cada clase del dominio que se desea persistir y poniendo sus atributos públicos basta para utilizarlos como entidades. Esto último es el problema de realizar esta solución, no solo nos obliga a hacer públicos los atributos de nuestras clases sino que además viola DIP, ya que las clases del dominio son dependientes del paquete Data Access. Por lo tanto decidimos que esta opción era inviable para cumplir con nuestro objetivo.

La segunda opción es tener clases separadas para representar las entidades y otra clase que mapee los objetos para cada una de estas. Con esta solución no tenemos necesidad de hacer públicos los atributos de las clases del dominio y logramos implementarla sin violar DIP (ver diagrama de paquetes). Sin embargo, tenemos el inconveniente de tener que pensar una lógica de traducción entre entidades y clases del dominio. Esto último nos generó problemas varias veces, debido a que la mínima diferencia que haya entre lo que se quiso persistir y lo que se traduce provoca inconsistencias, que terminan en la caída del sistema. Las pruebas ayudaron a reducir este problema rápidamente.
Otra desventaja de la segunda opción es que es necesario hacer RTTI para la traducción de clases abstractas (como User). Estas desventajas no son tan pesadas como la desventaja del método anterior. Es altamente necesario que el dominio no dependa de módulos de bajo nivel debido a que esto imposibilita la migración de sistema (o queda contaminado) y provoca que sea necesario recompilar el módulo del dominio cuando hay cambios en dichos módulos de bajo nivel.

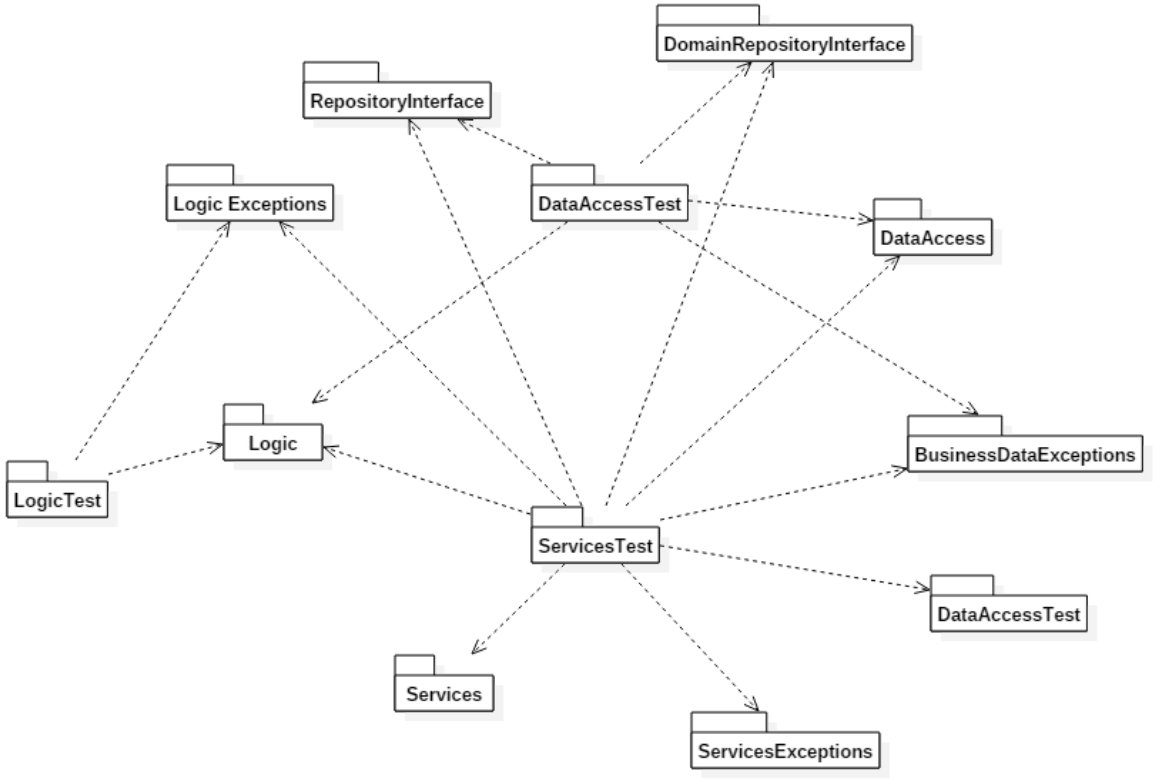
Por estos motivos se optó por la segunda implementación.

A continuación, se muestra el diagrama de tablas de la base de datos generada por las migraciones de Entity Framework.

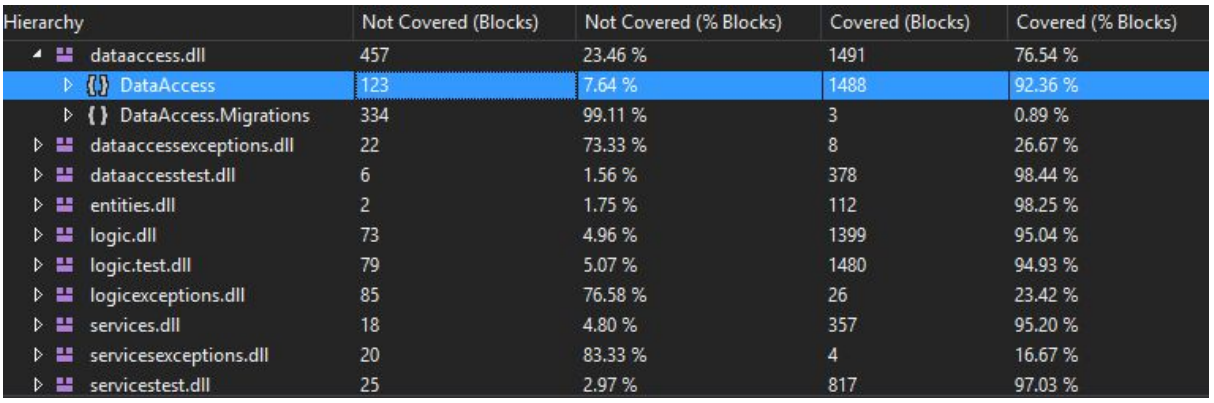


3 - Cobertura de las pruebas unitarias

El siguiente diagrama muestra la dependencia de los paquetes de prueba, este diagrama fue separado del diagrama de paquetes del principio para reducir la complejidad.



Existen 3 paquetes de prueba: LogicTest, ServicesTest y DataAccessTest.
A continuación se muestra el análisis de la cobertura de las pruebas unitarias desglosadas por paquete.



Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
dataaccess.dll	457	23.46 %	1491	76.54 %
DataAccess	123	7.64 %	1488	92.36 %
DataAccess.Migrations	334	99.11 %	3	0.89 %
dataaccessexceptions.dll	22	73.33 %	8	26.67 %
dataaccesstest.dll	6	1.56 %	378	98.44 %
entities.dll	2	1.75 %	112	98.25 %
logic.dll	73	4.96 %	1399	95.04 %
logic.test.dll	79	5.07 %	1480	94.93 %
logicexceptions.dll	85	76.58 %	26	23.42 %
services.dll	18	4.80 %	357	95.20 %
servicesexceptions.dll	20	83.33 %	4	16.67 %
servicestest.dll	25	2.97 %	817	97.03 %

Se observa que los paquetes Logic y Services tienen una cobertura de código alta, mantienen el mismo nivel que la versión pasada. Esto se debe a que se continuó realizando desarrollo guiado por pruebas en las nuevas funcionalidades agregadas a la lógica del negocio.
También, se puede ver que en los paquetes de excepciones, los porcentajes de cobertura en los paquetes de excepciones son bajos. Esto se debe a que no se realizaron pruebas unitarias sobre las excepciones, de las cuales muchas tienen 3 constructores, y ninguno de estos se probó. Se consideró que como solo tienen constructores vacíos, sin lógica, no era necesario probarlos.
Al analizar la cobertura de DataAccess, se ignora el paquete de migraciones generado por EntityFramework, ya que es codigo autogenerado y no testeable. Se puede observar que la cobertura es del 92%, relativamente bajo respecto al nivel de la versión anterior.Esto se debe en particular por que algunas funcionalidades de mapeo de objetos de Logic a estructuras de Entities no se implementaron usando TDD.
El resto de las líneas de código no cubiertas se debe a pruebas poco exhaustivas, casos imposibles de testear y olvido por parte de los desarrolladores.

En resumen se puede observar a partir de la cobertura, y también de los commits en el repositorio, que se continuó utilizando TDD en el desarrollo de la nueva versión de la aplicación.