Universidad ORT de Montevideo
Facultad de Ingeniería.

OBLIGATORIO1
Diseño de Aplicaciones1

Facundo Arancet-210696
Marcel Cohen -212426

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

Mayo, 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.

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
- Materiales de construcción, con los cuales se diseñan los planos

Esta versión del sistema posee tres 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 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.

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 numeros de 3 digitos. Si se selecciona un largo o ancho mayor a 300m se puede producir un error crítico en el sistema.

Gestión de usuarios. Esta funcionalidad solo está disponible para el usuario administrador. Consiste en crear, borrar y editar información personal de los demás usuarios (Clientes y Diseñadores) registrados.

Aclaraciones:
Nótese que no permitimos modificar el nombre de usuario de los usuarios registrados, que es el campo identificador entre todos los usuarios. Esta decisión se tomó en base a dos razones: Si un administrador cambia el nombre de un usuario (Ej. a un cliente), sin avisarle, este puede intentar intentar ingresar, y el sistema le indicará que no fue encontrado el usuario, entonces el cliente no sabe si le borraron la cuenta o le cambiaron el nombre. La segunda razón, es la observación del estado del arte, no hemos encontrado ningún sistema que permita cambiar el campo identificador (sea nombre de usuario, email o ci) del usuario. Entonces decidimos dejarlo como campo inmutable.

El sistema no le da la opción al administrador de crear otro administrador, ya que interpretamos de la letra que es uno solo y está dado por defecto, sin embargo, el diseño del sistema permite que sea fácilmente escalable a muchos administradores.

Al crear un usuario, el administrador genera la contraseña, no es autogenerada, por que si lo fuera así, el administrador no sabría la contraseña (que fue generada internamente) y no se la podria comunicar al cliente, y la funcionalidad que la aplicación le envie un mensaje por algún medio (mail,sms,etc) de su nueva contraseña está fuera de nuestro alcance, por costos y tiempo.

Modificar información de usuario. El sistema le permite (es una opción del menú), a todo usuario modificar su información en el sistema, salvo el nombre de usuario, que como ya mencionamos ,no se puede cambiar. Adicionalmente, como fue solicitado por el cliente, los perfiles de cliente que ingresan por primera vez en el sistema, se les debe mostrar sus datos de registro y permitirles su modificación.

Gestionar costos y precios de materiales. El administrador debe poder configurar los costos (para la empresa) y los precios (para el cliente) por unidad de cada material.

Aclaraciones:
Como se indica en la letra, pueden existir nuevos en el futuro, entonces se intento de dinamizar la forma de editar y guardar los costos y precios.

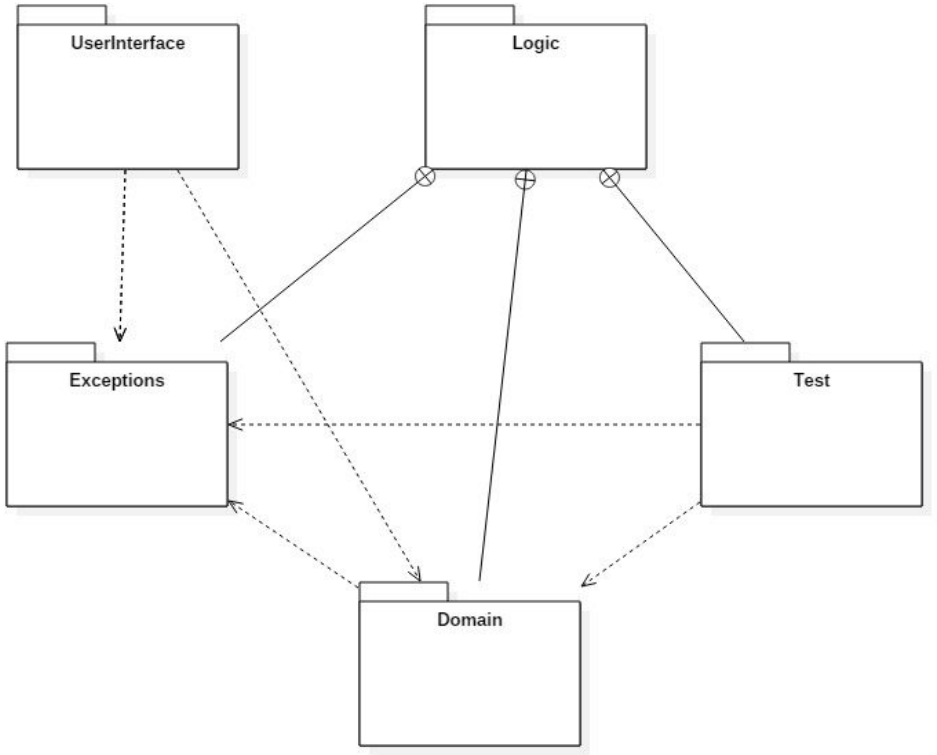
Gestión de planos. El sistema debe permitirle al diseñador crear, editar y borrar planos para cualquier cliente. Un diseñador crea un plano para un cliente, pero luego cualquier otro diseñador tiene la libertad de editarlo o borrarlo.
Del lado del cliente, este debe poder visualizar los planos que se le han creado que están en el sistema (no fueron borrados), y también debe poder visualizar el precio de los materiales desglosado por material.

Aclaraciones:
Fue aclarado en el foro, que una vez borrado el usuario, sus plano son eliminados del sistema.

2 Descripción y justificación de diseño

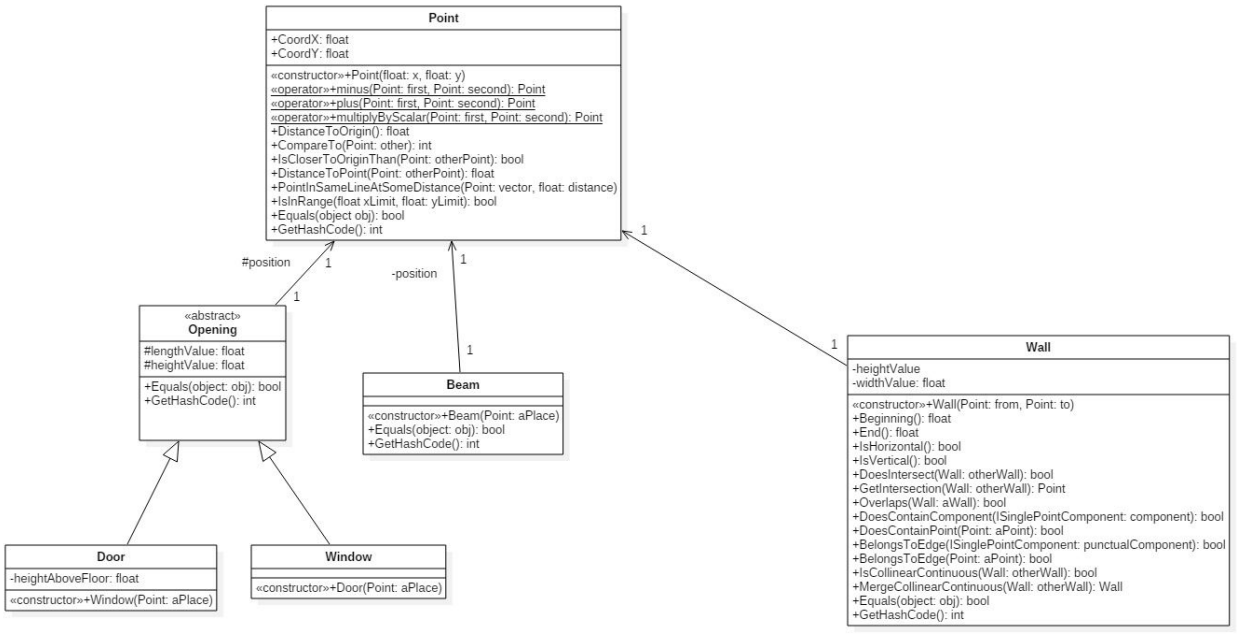
Diagrama de paquetes:

Agrupamos las clases del sistema en 4 paquetes: Logic.Domain, Logic.Exceptions, Logic.Test y UserInterface. Los 3 paquetes dentro de Logic están relacionados con la lógica del sistema. Logic.Domain contiene las clases del dominio, Logic.Exceptions contiene todas las las excepciones que son arrojadas por los métodos en las clases del dominio y Logic.Test es la clase que contiene las clases de prueba para las clases del dominio, osea, las clases del paquete Logic.Domain. Este paquete también depende del paquete de excepciones por que al ser exhaustivas las pruebas, prueban también los caminos donde se esperan excepciones.



Justificación del diseño

Modelado de los componentes de construcción



Comenzamos por definir cuales son los objetos que representaran los materiales de construcción. Estos son: Wall (Pared), Beam (Viga), Window (Ventana) y Door (Puerta). La letra del obligatorio dice que puertas y ventanas son tipos de abertura, y que es posible que el sistema pueda incorporar más tipos de aberturas, entonces decidimos generalizar las aberturas. Esto resultó en la creación de la clase abstracta Opening, de esta forma, los demás componentes se acoplan a esta clase abstracta, y así, al agregar nuevas aberturas (que es un evento probable) el impacto del cambio será mínimo.
Para modelar estos materiales de construcción,y su distribución en el espacio 2D del plano, evaluamos 2 opciones:

1- Utilizar una matriz que simule el plano 2D acotado, que para este obligatorio es discreto (las coordenadas X e Y son múltiplos de 1m), y los materiales serían ubicados en los casilleros de dicha matriz, modelando su posición en el plano. Los materiales no conocerían su posición, sino la matriz.

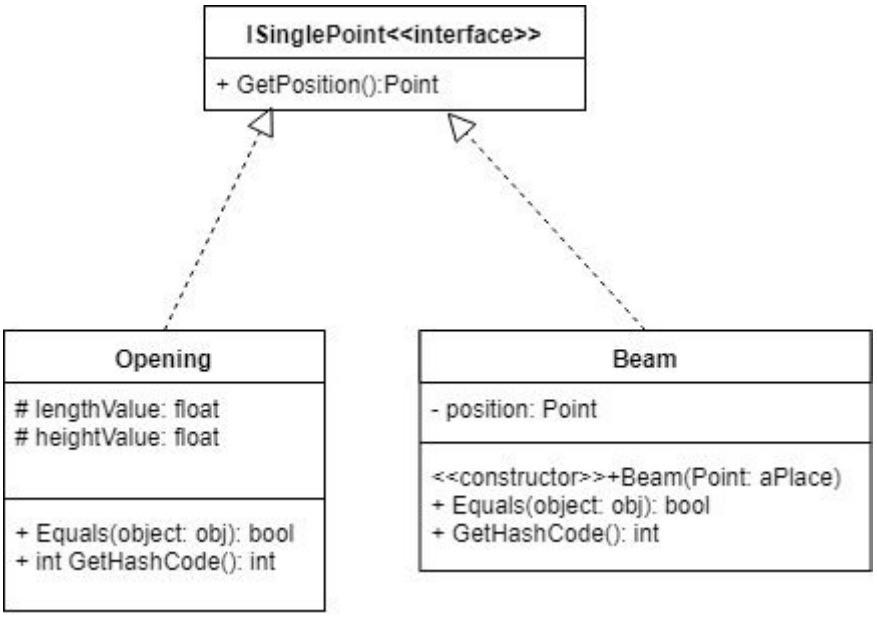
2- Que todos los componentes conozcan sus posiciones, y poder tenerlos almacenados en colecciones, para esto se necesitaría alguna forma de contener las posiciones.

De estas dos opciones, nos quedamos con la segunda. Primero, porque realizamos un análisis que consistió en ver, sin mucho detalle, para tener una idea, cómo se realizarían las operaciones en el sistema con ese modelo. En este análisis encontramos que la solución de la matriz, es más performante que la solución de la colección, pero tiene excesiva complejidad en sus algoritmos y su mantenimiento es muy costoso, ya que involucraría realizar RTTI (Run Time Type Identification) en muchas ocasiones, y todo eso es justamente lo que se pide evitar en este curso.
En segundo lugar, observamos que en el segundo modelo, por más que el acceso y validaciones en posiciones no son de O(1) como en la matriz, sino que O(n), se encontraron ciertas ventajas. Las operaciones son más sencillas, además, la librería Linq de c# optimiza las funciones sobre colecciones, simplificando el código, esto hace que el sistema sea más fácil de mantener.
La otra ventaja de este modelado, es que no se restringe el plano a valores NxN, entonces, se realizó todo el modelado de los puntos en los reales y toda la algoritmia de la construcción del plano (distancias, cálculo de intersecciones, etc), se modelo en el espacio RxR, que vendría a ser el plano 2D completo, y no acotado como pide la letra. Obviamente se realizaron las validaciones necesarias para que todos los materiales tengan posiciones en este plano acotado, pero se tiene la ventaja de que este modelo puede funcionar para cualquier subconjunto de RxR, en este caso NxN, y esto hace que el modelo sea versátil y reutilizable para cualquier tipo de plano 2D.

Como se puede ver en el UML, se modelo el punto (La clase Point), que posee una coordenada X y una Coordenada Y como propiedades. Su responsabilidad no es solo almacenar una posición en el espacio, también poder determinar ciertas propiedades que este tiene respecto al plano, cómo calcular su distancia al origen de coordenadas o a otro punto, determinar si está en cierto rango, etc.

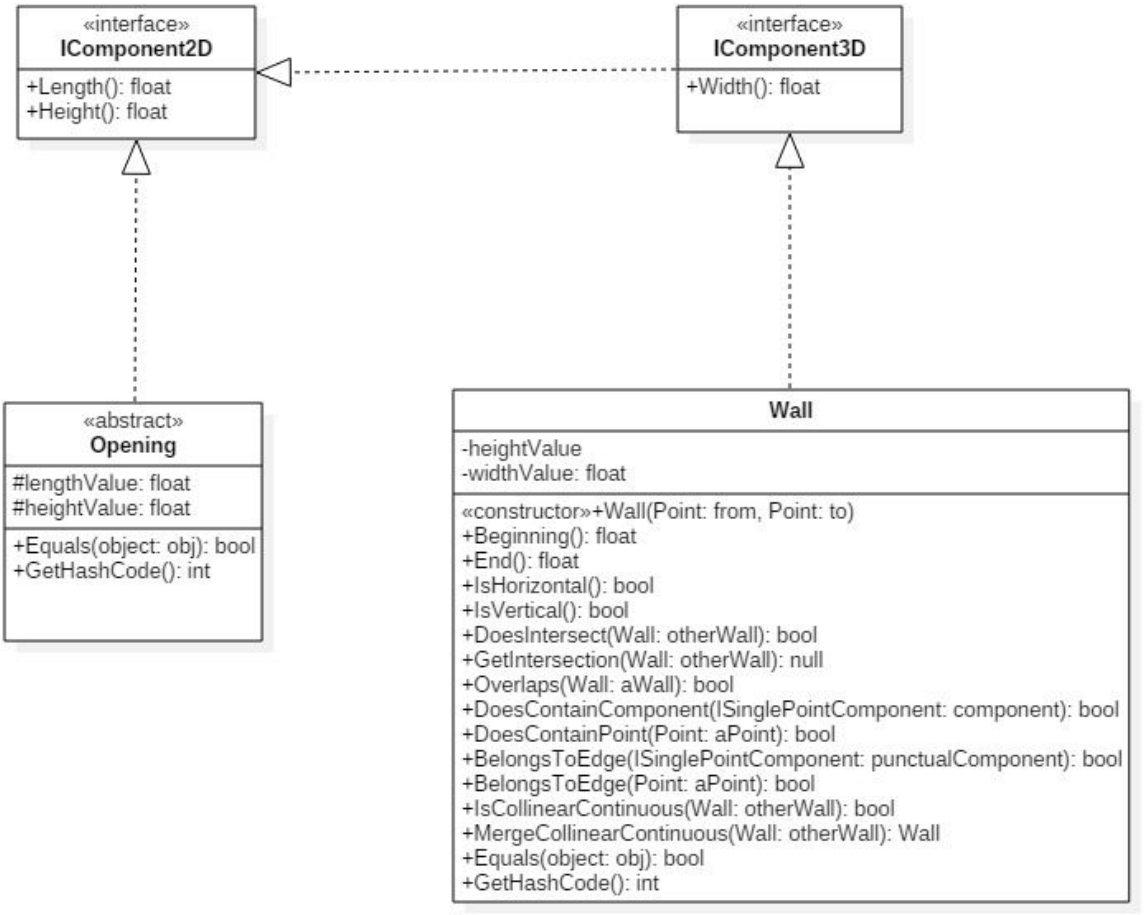
Resumiendo esta sección, los componentes de construcción contienen punto/s que almacenan su posición en el plano.

Utilización de interface para componentes de construcción puntuales (en el plano).



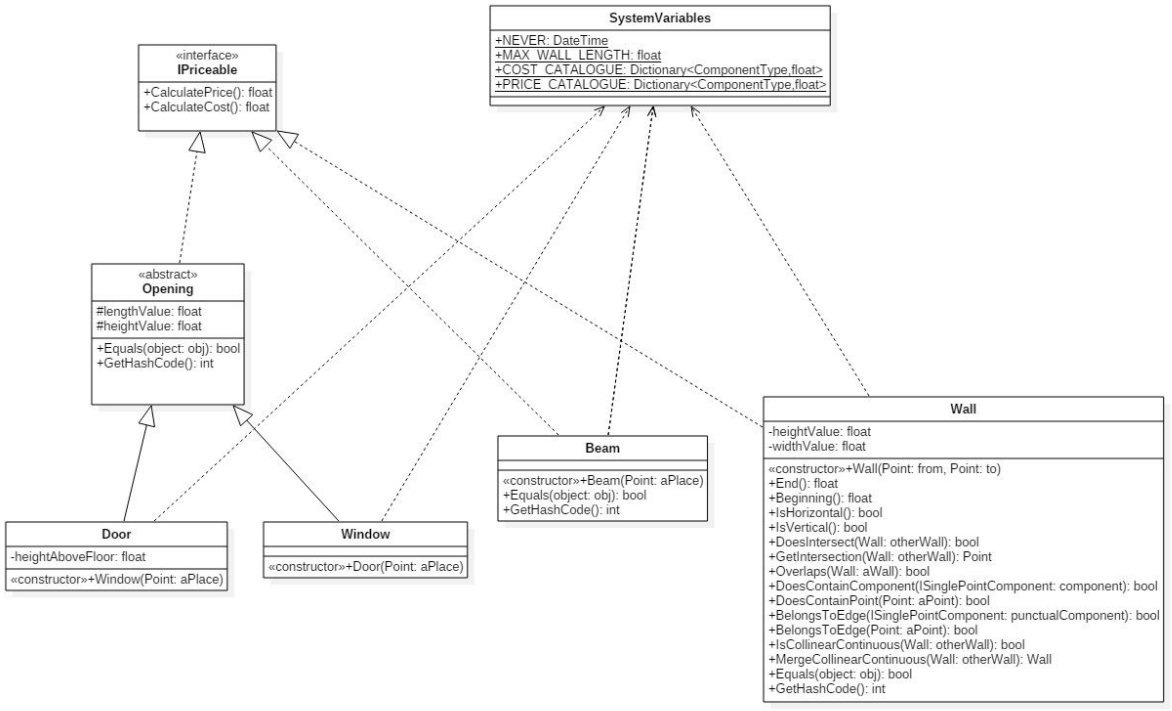
Durante el desarrollo, se consideró conveniente que los componentes de construcción se puedan agrupar en dos conjuntos, según la forma en que se ubican en el espacio, estos son: Los materiales puntuales, cuya representación en el plano es dada por un punto, y los no puntuales. Por eso se optó por incorporar una interfaz que define el comportamiento de todos los materiales puntuales, esta es ISinglePoint. La justificación de su uso es la siguiente: ante las validaciones en el plano, una viga no puede ocupar el mismo lugar que una abertura y viceversa. No es el caso de la pared, que no es puntual y puede contener vigas y aberturas en ella. Para este tipo de validaciones, es más reutilizable comparar al nuevo objeto puntual ingresado (abertura o viga) contra el resto de los componentes puntuales, y para ver si hay alguno cuya posición coincida. También, como vigas y aberturas deben pertenecer a una pared, se puede generalizar la validación a todos los materiales puntuales del sistema.

Definición de interfaces para describir las dimensiones de los materiales:



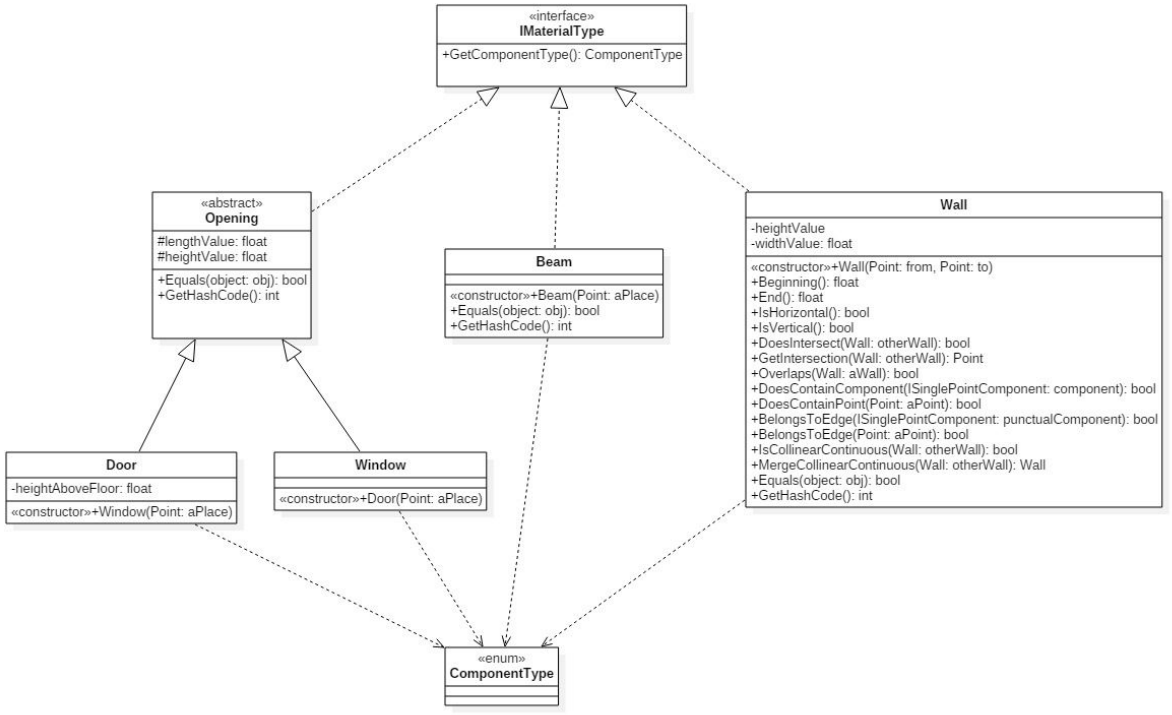
Se observó que un comportamiento común en todos los componentes, es que la mayoría tienen dimensiones, existen también componentes que en el dominio del problema son adimensionales, que es el caso de la viga. En este dominio, identificamos objetos bidimensionales, como las aberturas, que son definidas por su largo y altura. También identificamos objetos con tres dimensiones, en este caso la pared, donde se especifica su largo, alto y ancho. De este análisis, surgió la creación de la interfaz Component2D, que especifica los comportamientos de calcular su largo y altura.Luego se definió la interfaz Component3D, que extiende a Component2D y le agrega la función de calcular el ancho. De esta forma, se especifican propiedades comunes que deben cumplir ciertos tipos de materiales. También permite el polimorfismo, por ejemplo, los componentes que implementan la interfaz Component2D pueden calcular su largo de distinta forma, a veces puede ser una simple devolución, como las aberturas, o un cálculo, como la pared. Esto hace que se puedan incorporar nuevos tipos sin realizar muchos cambios.

Modelado de los costos y precios de los materiales:



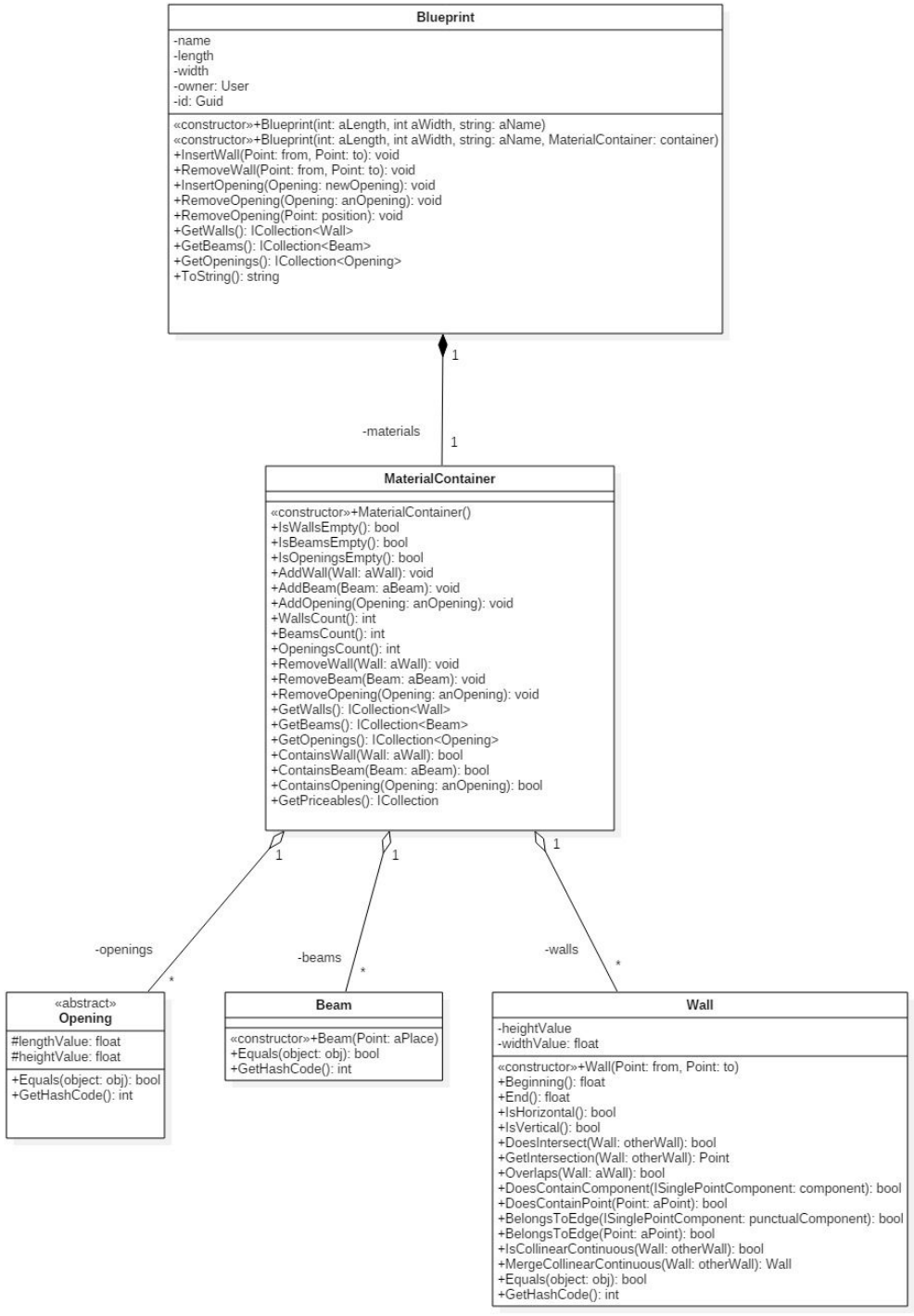
Siguiendo el mismo pensamiento de la decisión de diseño anterior, se hizo lo mismo con los precios y costos de los materiales. Todos los materiales tienen precios (para el cliente) y costos (para la empresa). Definiendo la interfaz, nos aseguramos que todos implementen los comportamientos de calcular sus costos y precios.De esta forma evadimos el RTTI usando polimorfismo, porque no tenemos que preguntar si es pared para multiplicar la distancia en metros por el precio del metro de pared, o preguntar si es puerta para devolver el precio de la unidad.
El modelado de los precios y costos de los materiales se detallará a continuación.

Definición de interface IMaterialType para los componentes de construcción



El cliente también solicitó que el administrador pudiera modificar los precios de los materiales, y junto con estas modificaciones, cambien los precios de los planos. Para lograr esto, se decidió que tendría que existir un diccionario que vincule a cada material con su precio y con su costo.
También,más adelante, se vio la necesidad que que en la UI, el panel de dibujo del plano pudiera vincular cada material con un determinado color para dibujar y una determinada figura.
Esta necesidad de asociar un tipo de material con una propiedad, y evitando el RTTI, resultó en que se especificara un comportamiento común a todos los materiales, un comportamiento que nos permita saber su tipo de material, asi podríamos usarla como entrada en estos diccionarios a implementar. De esta idea surgio la interface IMaterialType, que especifica que todo material debe devolver un elemento de un Enum, ComponentType, y con esto se realiza la correspondencia material-precio, material-color, etc. De todas formas, podemos ver que no deja de ser RTTI, aunque es más flexible, ya que se pueden agrupar materiales en mismos tipos y reutilizar tipos.

Modelado del plano de construcción:



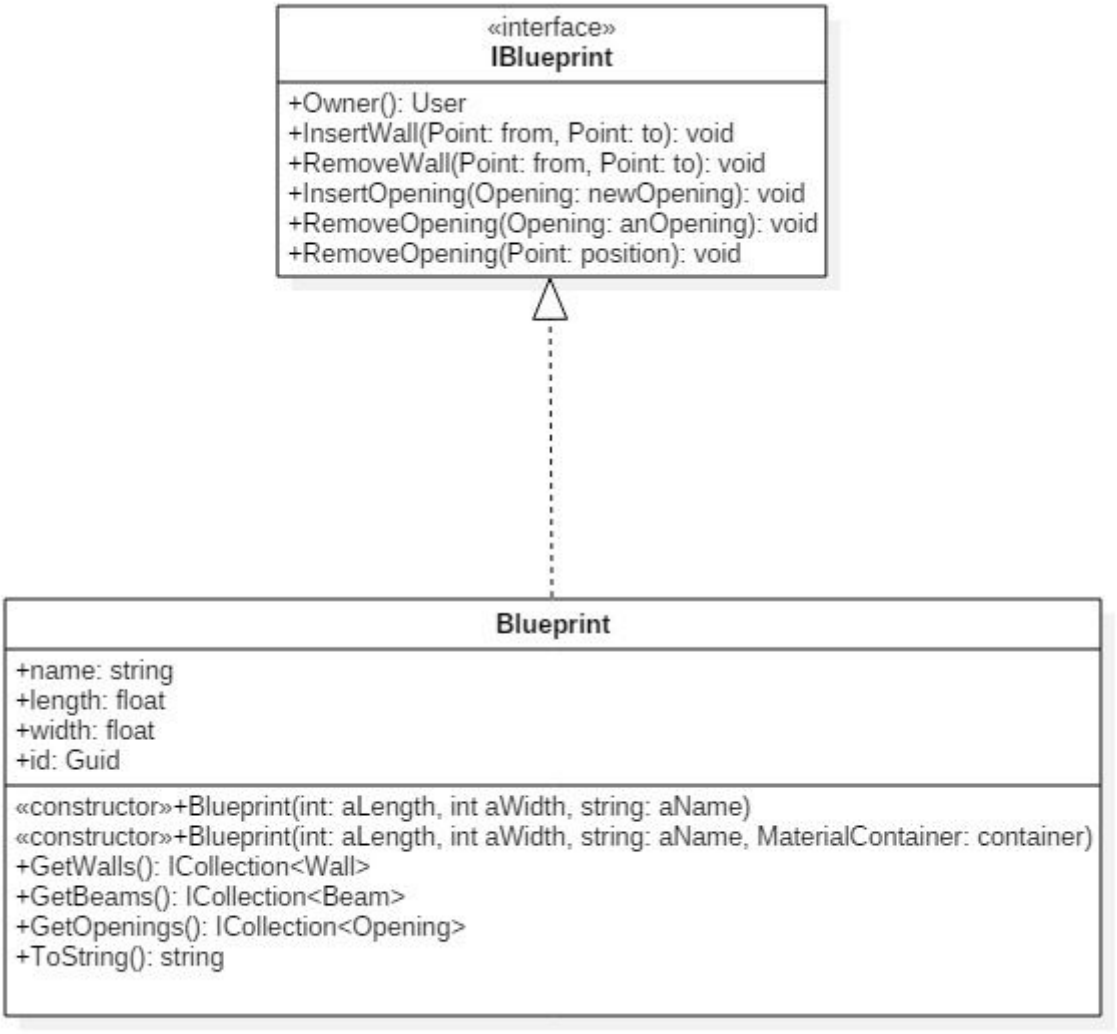
Para representar los materiales en el plano se modelo la clase Blueprint, que representa el plano. Inicialmente, este iba a contener los componentes de construcción y conocer toda la lógica de como colocarlos en el plano. Nos pareció que de esta forma, la clase tendría muchas responsabilidades (violando el SRP),

entonces se utilizó composicion-delegación para delegarle la responsabilidad de almacenar los materiales de construcción a una nueva clase, llamada MaterialContainer. Esta clase tiene la responsabilidad de guardar, remover y saber si ya existen los componentes, ocultando su implementación. De esta forma, Blueprint solo se encarga de colocar y quitar los elementos del contenedor de acuerdo a la lógica de la construcción.

Como se puede ver, Blueprint posee dos constructores, uno de ellos tiene un parámetro adicional que es una instancia del contenedor de los materiales. La incorporación de este constructor adicional, surgió durante el proceso de TDD, ya que cuando se probaban los algoritmos de Blueprint, se necesitaba saber el estado del contenedor, es decir, cuantos materiales de cada tipo quedaban luego de las operaciones testeadas. El problema que surgió, fue que había que encontrar la forma de acceder al MaterialContainer que el objeto Blueprint escondía, y no era correcto crear un método de acceso público que revelara la implementación de Blueprint, y si se optara por hacer este sacrificio, se estaría violando la ley de Demeter en las pruebas, ej. se utilizarian sentencias como esta: testBluepring.GetMaterialComponent().WallsCount(), para consultar la cantidad de paredes en el plano.
Entonces, gracias a este constructor, se tienen dos objetos en la clase prueba: El Blueprint y su MaterialComponent, que se le pasa por parametro al construirlo, pero tenemos la referencia a este objeto en memoria, ya que es una variable de la clase de prueba. Entonces para los Asserts se consulta directamente al contenedor, sin violar la ley de Demeter.

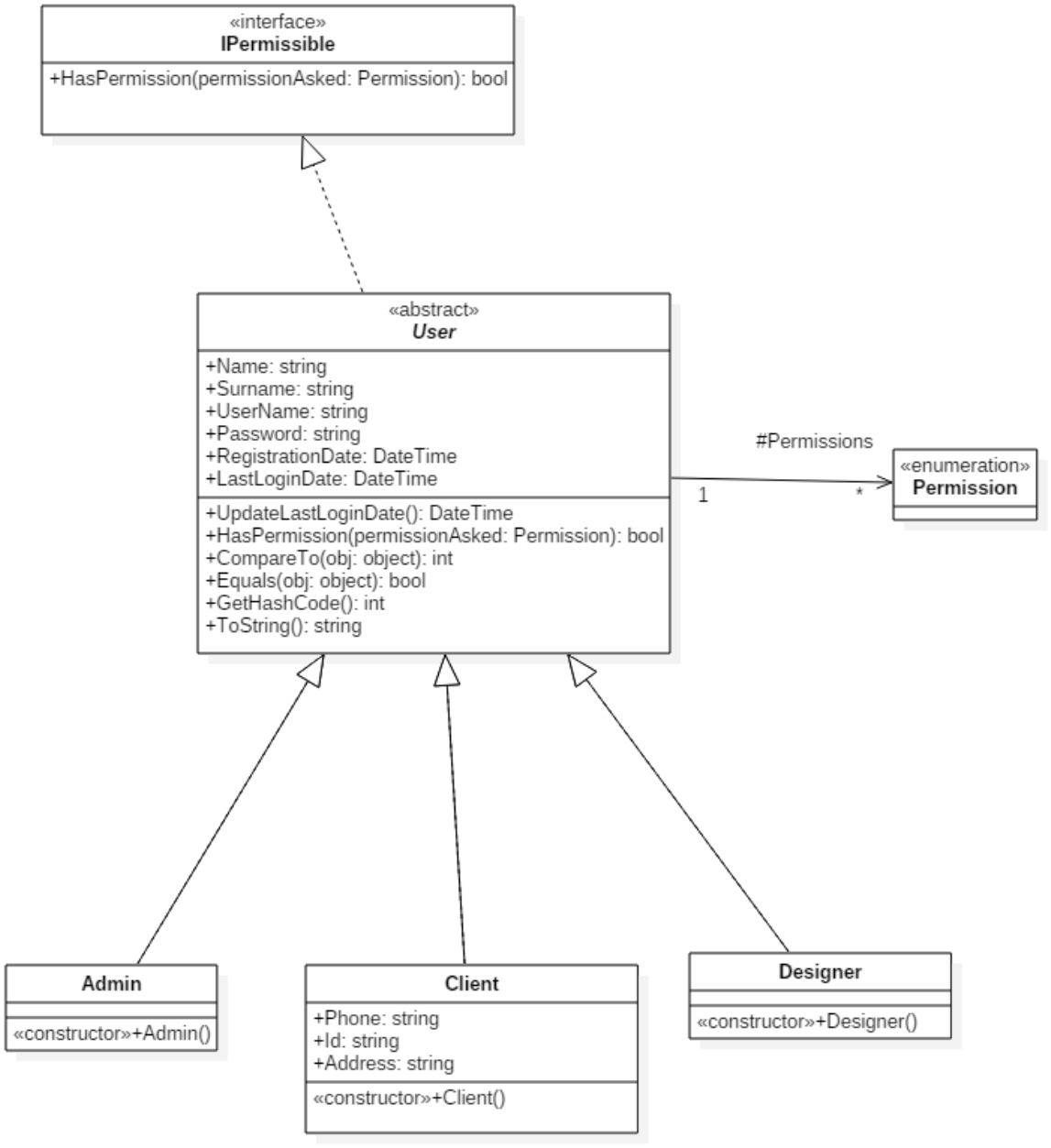
De todas formas, nos dimos cuenta más adelante, que este diseño no es muy mantenible, ya que la clase Blueprint tiene la responsabilidad de saber como se colocan todos los materiales, lo que implica que al ingresar nuevos materiales habrían muchos métodos que modificar en esta clase Blueprint. Dicho de otra forma, la lógica del plano está hecha “a la medida” de los materiales disponibles en esta versión, con una dificultad para evolucionar.

Se entendió que en un diseño más óptimo, todos los materiales serian implementaciones de una misma abstracción, entonces la representación del plano se reduciría a una colección de elementos de esta abstracción. A su vez, cada elemento de esta abstracción tendría que saber él mismo como colocarse en el plano y cómo se relaciona con los demás materiales. Al realizar el análisis, consideramos que era un diseño muy dificil de lograr, entonces se tomó la siguiente decisión: Terminar la implementación que se estaba realizando y si se contaba con tiempo suficiente, probar esta nueva implementación. Para esto se creó la interface IBlueprint, que define los comportamientos del plano.



De esta forma, como el acoplamiento del plano con el resto del sistema es a través de la clase BlueprintPortfolio, que sirve como fachada (se explicará más adelante), si esta fachada se acopla a la interface IBluerint, si se lograba la implementación del plano deseada, se podría sustituir fácilmente con un mínimo impacto de cambio. Lamentablemente, no se logró esa implementación, y esta interfaz no resultó muy útil en el sistema.

Usuarios:



- Diseño de usuarios:
  Para modelar los usuarios se creó una clase abstracta User que contiene los atributos comunes a los tres tipos de usuarios que la letra del obligatorio menciona: cliente, diseñador y administrador. Para cada uno de estos tipos de usuario, se creó una clase que los represente y que hereden de User. Se opta por usar una herencia y no una interfaz porque está claro que este es el tipo más general de los usuarios, no será necesario heredar de otra clase (la herencia es única). Además, los usuarios no solo comparten comportamiento, también atributos. Las responsabilidades de estas clases quedan acotadas a conocer los datos de los usuarios y a las operaciones que corresponda hacer

con estos datos. Para el caso de este obligatorio, los usuarios deben actualizar su fecha de último login al sistema automáticamente, por lo que existe un método en la clase abstracta que todos sus hijos pueden usar.

- Justificación del diseño:
  La utilización de una clase abstracta permite extender de ella para futuras implementaciones. Esto facilita los cambios de forma tal que no haya un impacto en las clases que dependen de ella.
- Ejemplo de implementación de cambio:
  Se quiere agregar un nuevo tipo de usuario “Constructor”. Basta con crear una nueva clase que herede de User.

Permisos:

Entendemos un “permiso” como un token que da acceso a las funcionalidades del sistema.

- Diseño de permisos:
  Los requerimientos (letra del obligatorio) especifican que los usuarios deben tener diferentes permisos para la utilización de diferentes features.
  Modelamos esto agregando en la clase User una lista de permisos, los cuales se representan por medio de una clase pública enum.
- Justificación de diseño:
  El motivo de diseñar los permisos como un enum es que facilita la alta y baja de los mismos, así como también su manejo en la ui (excepción NoPermission para control de los mismos). Además, así evitamos los literales para cumplir con clean code.
  Los usuarios (User) deben implementar la interfaz IPermissible, que tiene como método a HasPermission(Permission permission). Este método devuelve true si y solo si el usuario tiene el permiso preguntado.
- Ejemplo de implementación de cambio:
  Esta forma de diseñar los permisos nos permite soportar cambios de acceso a funcionalidades fácilmente. Supongamos el ejemplo de que ahora los diseñadores también pueden agregar clientes: otorgando el permiso correspondiente a la creación de clientes al diseñador es suficiente (una línea de código, agregar el permiso en el constructor de Designer).

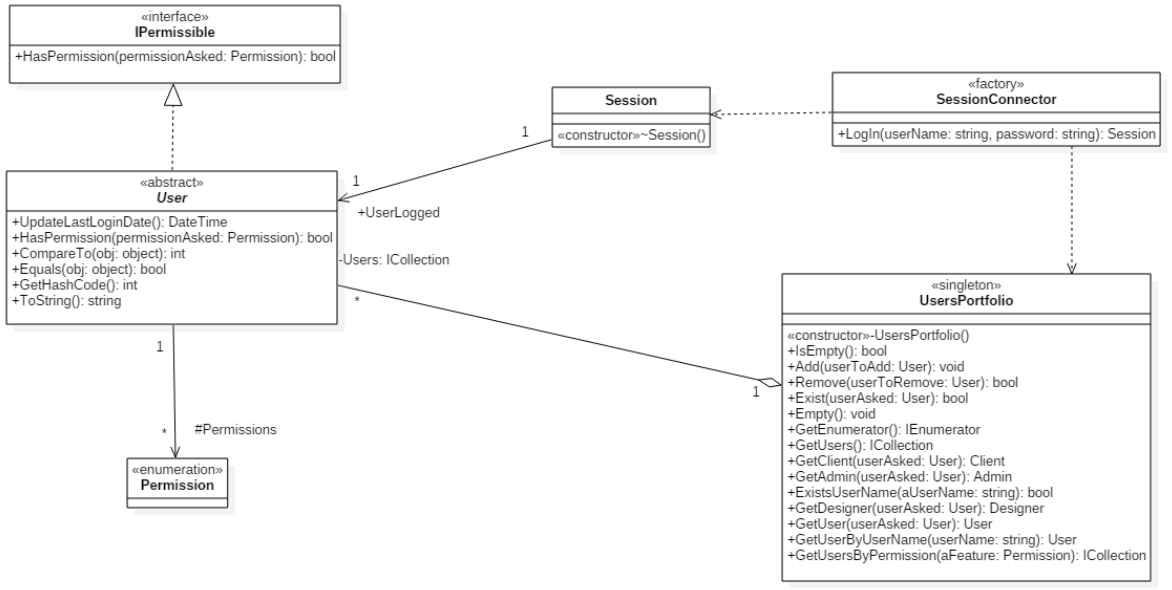
Almacenamiento de usuarios:

- Diseño de almacenamiento de usuarios:
  Los usuarios del sistema se guardan en una clase llamada UserPortfolio, la cual es un Singleton. Esta clase es internal para mantener seguros los datos de los usuarios y poder controlar permisos de acceso a los mismos.
- Justificación de diseño:
  Decidimos usar el patrón de diseño “singleton” para mantener una integridad de los datos de los usuarios. Tener los usuarios todos juntos facilita el acceso

y modificación de los mismos, para esto necesitamos el singleton. Así, todas las clases de la lógica pueden acceder a la información de los usuarios pero no las clases de la ui.

- Ejemplo de implementación de cambio:
  Supongamos que se decide implementar una base de datos a la lógica. Este cambio impacta solamente sobre el singleton, basta con cambiar la implementación de este para que todo siga funcionando correctamente.

Session:

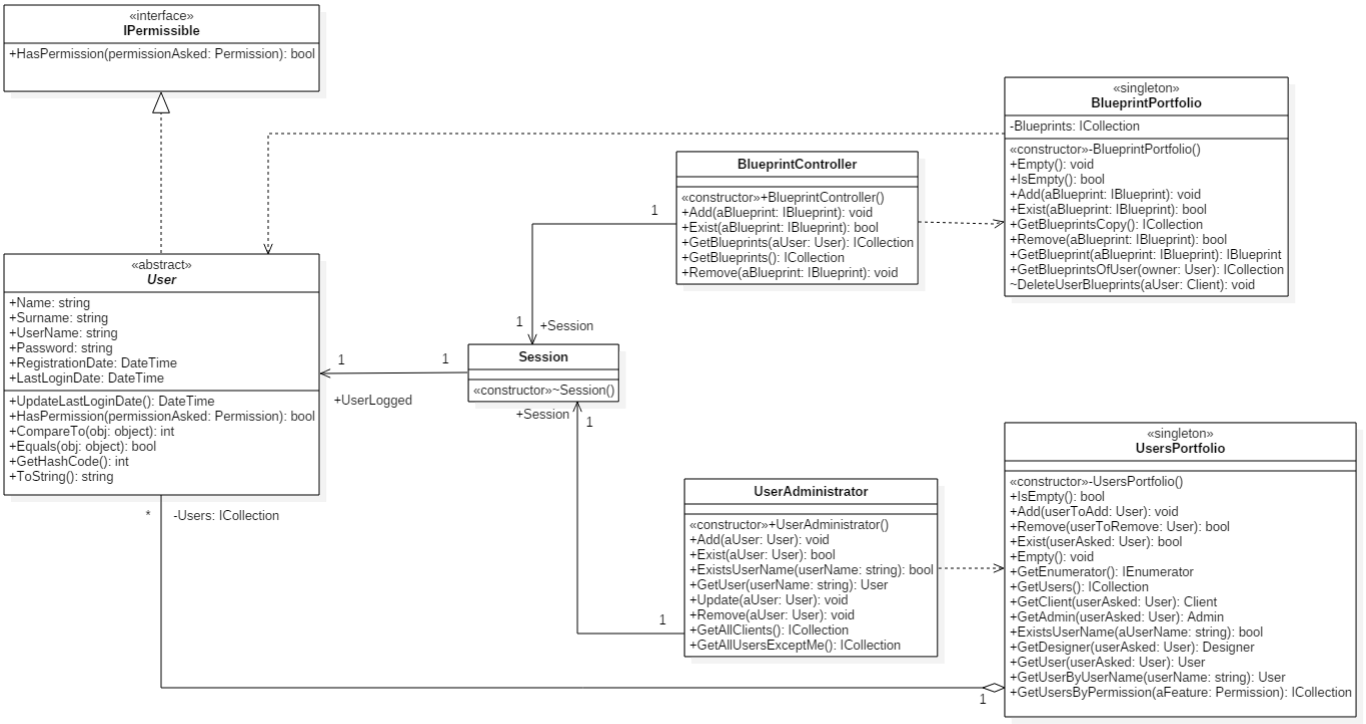


- Diseño de sesiones de usuarios:
  Utilizando una clase Session se modela el inicio de sesión de los diferentes usuarios del sistema. En esta clase se almacena la información del usuario (no seteable luego de la construcción) y cualquier otra variable que se necesite para utilizar luego en la ui (por ejemplo, la información de si es el primer login de un usuario viaja en la sesión). Se utilizó una fábrica para la creación de estos objetos. La clase que representa dicha fábrica se denomina SessionConnector. Esta clase tiene un método “LogIn” que dado un nombre de usuario y contraseña, si el usuario pertenece al UserPortfolio (los usuarios tienen redefinido el equals, son iguales si sus nombres de usuario son iguales) y la contraseña se corresponde con la pasada por parámetro, devuelve un objeto Session que es utilizado por la ui para consultar permisos e información del usuario.
- Justificación de diseño:
  Modelar la sesión como un objeto que es construido por medio de una fábrica permite administrar los permisos a las diferentes features. La fábrica es necesaria para controlar que el usuario que se loguea pertenezca al UserPortfolio y no un usuario inventado en la ui. Funciona como una

credencial, posee información importante del usuario que lo permite identificarse.

- Ejemplo de implementación de cambio:
  Se quiere implementar que el sistema permita varios usuarios logueados al mismo tiempo. Basta con solicitarle varios objetos Session al SessionConnector, uno por cada usuario logueado.

UserAdministrator y BlueprintController:



Un requerimiento del sistema es que el usuario administrador pueda gestionar la información de los otros usuarios. Esto genera el siguiente debate: ¿los métodos que accedan a UserPortfolio deben estar en la clase Admin (clase que modela al usuario administrador) o en otra clase?. En principio uno puede pensar que lo lógico es que la clase Admin tenga estos métodos, ya que es parte de su comportamiento. Sin embargo, necesitaríamos repetir código y vulnerar permisos si quisiéramos que varios usuarios tengan la misma funcionalidad. Es por esto que tomamos la decisión de separar estas responsabilidades de la clase Admin y delegarlas a una clase UserAdministrator. El mismo razonamiento fue aplicado para la administración de planos, creando un BlueprintController.
Aclaración: programamos estas clases pensando que el objeto devuelto por las estructura de datos (UserPortfolio y BlueprintPortfolio) devolvía copias de los objetos cuando en la realidad son punteros a los objetos almacenados, por lo que no se

controla que un cliente no tenga permisos de editar sus planos. Si se devolviera una copia de los objetos, implementando un método Update en BlueprintController que (para el caso de los planos) recibe un plano y actualiza sus datos en BlueprintPortfolio, se podría controlar permisos de edición de planos para los diferentes usuarios del sistema.
Estas clases fueron diseñadas intentando utilizar el patrón de diseño “Actor - Role pattern”, el cual consta de modelar separadamente roles de actores, para luego modelar sus relaciones.
Definimos como actor al objeto que realiza tareas en el sistema y rol a la tarea que es realizada por un actor.
UserAdministrator es la clase que modela el rol que puede tomar un actor (User). Esta clase es la encargada de ejecutar las modificaciones de datos de usuarios si quien las ejecuta tiene permisos de hacerlo.
Estas clases que modelan los diferentes roles que un usuario puede tomar en el sistema, reciben en su constructor un objeto Session, el cual utilizan para ver la información del usuario y sus permisos. Con esta información, la clase que modela el rol puede exigir que quien ejecuta sus métodos sea un usuario con permisos que pertenece al sistema (el objeto Session de un usuario solo puede ser creado si SessionConnector lo encuentra en el UserPortfolio).
Si se intenta ejecutar un método de las clases rol con una sesión (usuario) sin permisos suficientes, el método arroja NoPermissionsException. Esta excepción es utilizada por la ui para saber que un usuario no puede acceder a esa feature del sistema.

- Diseño de UserAdministrator: Este rol tiene los métodos de modificación de UserPortfolio. En cada método se verifica que el usuario de la sesión tenga los permisos necesarios para ejecutarla.
- Justificación de diseño: Este modelado del problema separa los actores de los roles, permitiendo que el mismo usuario pueda asumir diferentes roles. Con esto conseguimos que si diferentes usuarios tienen funciones en común, no tenemos que repetir código, solo asignar los permisos correspondientes.
- Ejemplo de implementación de cambio:
  Si agregamos un nuevo usuario Recepcionista que puede ver todos los usuarios, basta con asignarle el permiso correspondiente y que se loguee con el rol “UserAdministrator” para usar la funcionalidad correspondiente. Notar que no se repite código.

- Diseño de BlueprintController: Este rol tiene los métodos de modificación de BlueprintPortfolio. En cada método se verifica que el usuario de la sesión tenga los permisos necesarios para ejecutarla.
- Justificación de diseño: Existen dos usuarios que manipulan los planos, los clientes y los diseñadores. Ambos pueden ver planos pero con diferentes

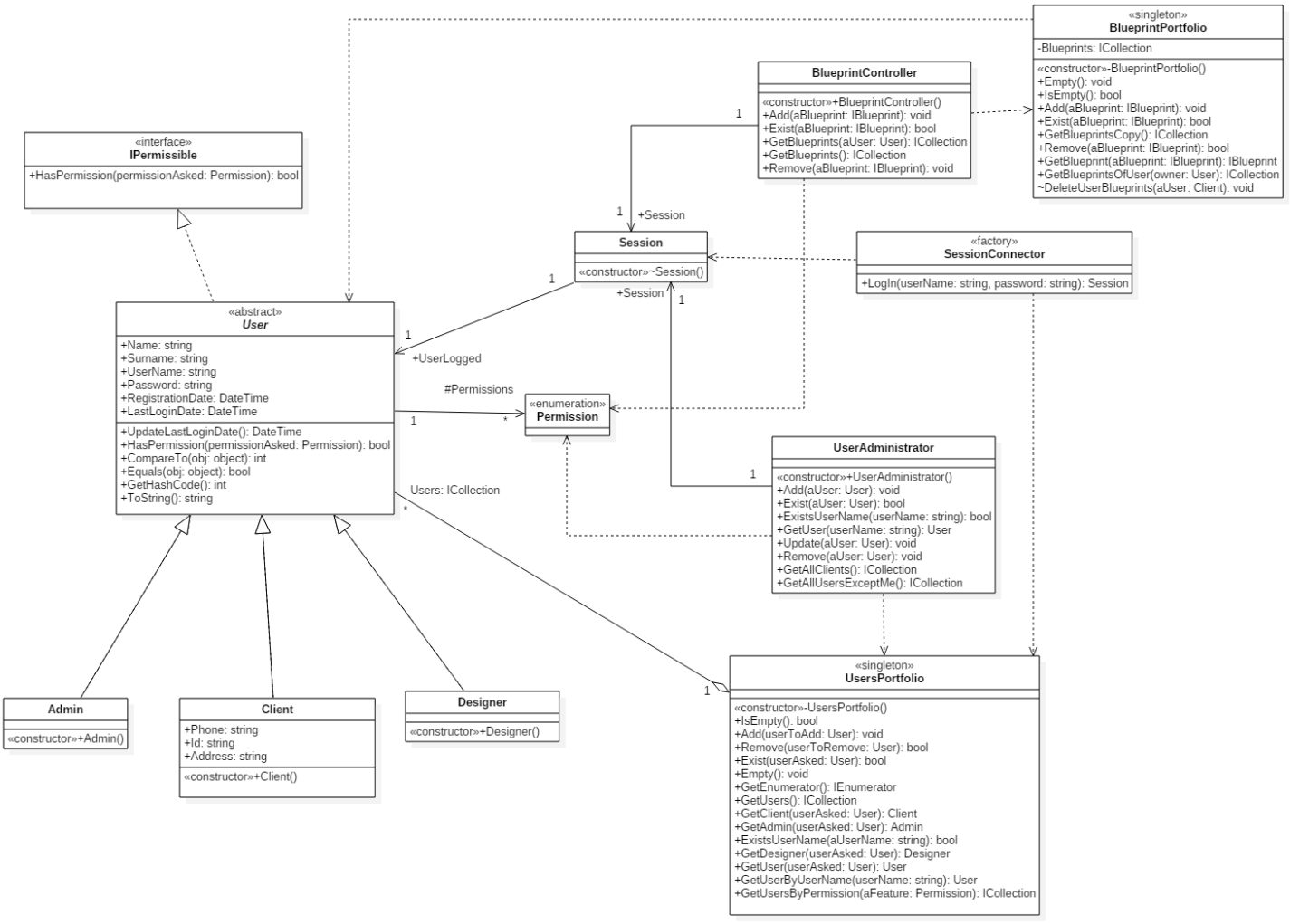
niveles de restricción, un cliente puede ver sus planos y un diseñador puede ver todos los planos. Los clientes tienen el permiso de ver planos propios y los diseñadores tiene el permiso de ver todos los planos. Este diseño es muy versátil a cambios.

- Ejemplo de implementación de cambio:
  Supongamos que los administradores ahora también puede ven los planos en el sistema. Si se le agrega el permiso de ver planos (una línea de código), el usuario administrador tendrá los permisos necesarios, por lo que podrá usar la función de BlueprintController que obtiene de todos los planos.

Las ventajas que presenta este diseño son:

- No es necesario repetir código para modelar comportamientos en común entre usuarios
- Se le puede exigir a la ui que haya un usuario logueado con permisos suficientes para poder realizar las operaciones con la lógica.
- La alta y baja de funcionalidades en los usuarios es muy sencilla de implementar (basta con cambiar la lista de los permisos del usuario). SEs reutilizable en cualquier sistema que maneje usuarios con permisos ya que estos están desacoplados del resto de la lógica (no hay dependencias de los usuarios a otras clases del dominio).

Por otro lado,.esta solución presenta la desventaja de que es necesario estar controlando constantemente los permisos del lado de la Ui, lo cual es molesto a la hora de programar (considerando que para este obligatorio, el control de accesos no es un requerimiento de tanta importancia).
Otra desventaja de esta solución es que usa el método HasPermission(Permission permission), la cual es poco explícita por ser Permission un enum. El compilador no puede ayudar a detectar errores si lo que se utiliza es un enum para modelar permisos.
La idea de utilizar este patrón surgió de la lectura del artículo de Martin Fowler “Dealing with Roles”.
El patrón que intentamos implementar fue el Role Object descrito por Fowler en su artículo, siendo UserAdministrator y BlueprintController estos objetos que modelan roles.



Podemos ver todas las partes juntas en el diagrama anterior. El flujo normal de programación y funcionamiento sería el siguiente: un usuario se loguea al sistema ingresando nombre de usuario y contraseña en un SessionConnector, obteniendo así su objeto Session con la información del usuario (en caso de que el usuario exista y la contraseña ingresada sea correcta). Una vez que se tiene el objeto Session correspondiente, un usuario puede asumir el rol de UserAdministrator o de BlueprintController, utilizándolo en los constructores de estas clases. Luego de obtener una instancia de UserAdministrator o BlueprintController, el usuario puede hacer uso de sus métodos si tiene los permisos suficientes. Hasta aquí, no solo nos aseguramos de que un usuario puede acceder a una feature si tiene permisos, sino que también controlamos que un programador no utilice mal la lógica de permisos (que no los pase por arriba). No hay forma de que un usuario use features si no se loguea y además tiene los permisos suficientes para usarla.

Una opción que puede surgir para mejorar esta idea es dividir las clases de roles en roles más específicos y así no sobrecargar una clase con muchas responsabilidades (no violar el principio de responsabilidad única). Por ejemplo, BlueprintController podría separarse en BlueprintViewer y BlueprintEditor, de modo que separamos la

responsabilidad de controlar la visión y controlar la edición de planos en dos clases, haciéndolo más fácil de mantener y entender.

Manejo de excepciones

El capítulo 7 del libro Clean Code trata sobre el manejo de errores, y explica la importancia del manejo de excepciones para que el código sea más claro.

Nuestro código debe ser legible, pero también debe ser robusto. Estos no son objetivos contradictorios. Podemos escribir código claro y robusto si vemos el manejo de errores como una tarea separada, algo que se puede ver independientemente de nuestra lógica principal. En la medida en que podamos hacer eso, podemos razonar al respecto de forma independiente, y podemos hacer grandes progresos en la mantenibilidad de nuestro código.

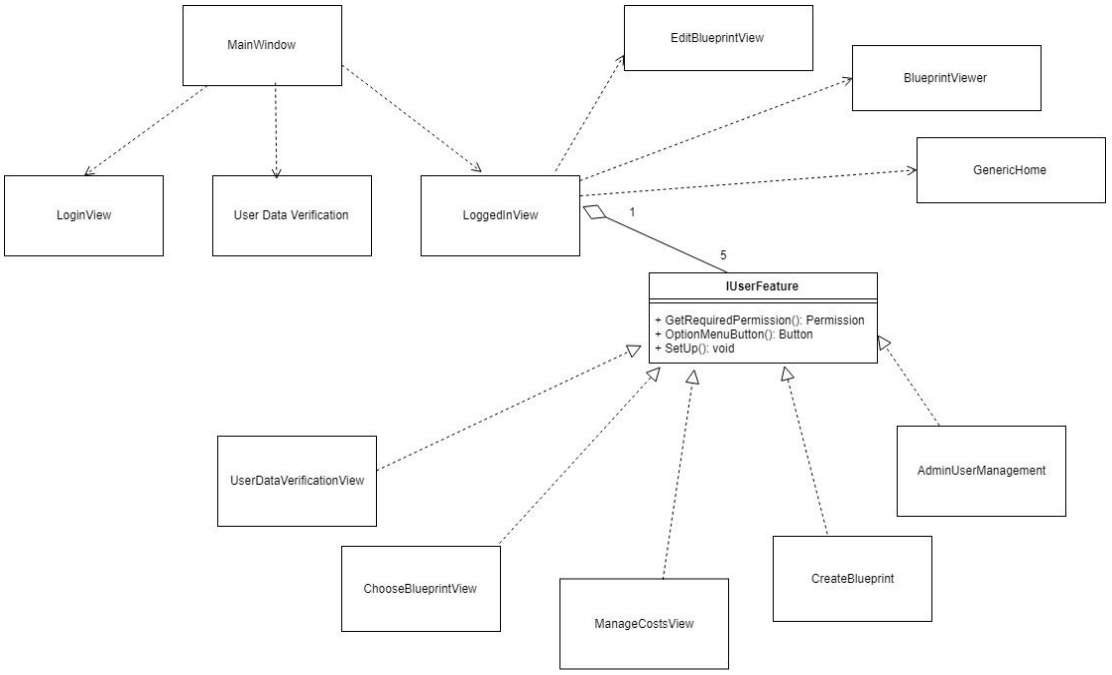
Con las excepciones y su manejo mediante sentencias try-catch-finally, podemos definir el flujo normal del método y en otro lado el manejo de errores. Esto hace que sea más sencillo de entender. También se menciona en el libro que las excepciones tienen que ser descriptivas del error acontecido. Por eso, decidimos en nuestro obligatorio, crear algunas excepciones descriptivas.

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 Blueprint
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	Blueprint

	por que esta fuera del rango o por no ser vertical u horizontal	
ComponentOutOfWallException	La excepcion se arroja cuando se intenta ingresar una abertura o una viga fuera de una pared que la contenga	Blueprint
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
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	UserAdministrator BlueprintController

Diagrama de la interfaz de usuario

Este diagrama a continuación representa la estructura, y las relaciones entre los objetos en nuestro paquete UserInterface.



Clean Code

Durante todo el proyecto de desarrollo nos basamos en los estándares de codificación que provee el libro Clean Code de Robert C. Martin.

A continuación se detallarán los puntos cubiertos del libro, con algunos ejemplos:

Nombres significativos:
Se intentó en todo momento utilizar nombres descriptivos en las variables y funciones.

Bloques e indentado:
Se configuró el code-formatter de visual studio para que utilice los estándares de indentación de c#.

Uso de excepciones en lugar de error codes

Como se vio en el curso, el uso de excepciones hace que el código se mas claro de entender

```
private void SetPhone(string aPhone)
{
    if (String.IsNullOrEmpty(aPhone))
    {
        throw new ArgumentNullException();
    }
    phone = aPhone;
}
```

Formato vertical:
Ninguna clase del sistema supera las 500 líneas de código.

Formato horizontal:
Se procuró que las líneas de sentencia no superen un largo de 200 caracteres

La Ley de Demeter:
En la justificación del diseño, se puede ver un ejemplo en el que se utilizó una referencia a memoria para no violar la ley de demeter. En la clase de prueba de Blueprint.

No retornar null, no pasar null
Un ejemplo de esta práctica es la clase NullUser, que es utilizada cuando no hay usuario logueado en el sistema.

Organización de clases:
Las clases se organizaron en el siguiente orden, de arriba hacia abajo.
constantes estáticas publicas
variables estáticas privadas
variables de instancia privadas
funciones públicas con sus funciones privadas de utilidad a continuación

3 Pruebas Unitarias

Se analizó la cobertura de código de las pruebas unitarias. Nos centramos solamente en el paquete Logic.Domain, que es el que posee el dominio del sistema. El test arrojó una cobertura total del 92.5% de cobertura de líneas de código, lo cual indica que la mayoría del código fue testeado por las pruebas unitarias.
Esto es una consecuencia de la técnica de desarrollo que utilizamos, TDD, desarrollo guiado por pruebas. Esta técnica la aprendimos en el curso, consiste en crear las pruebas de los métodos, antes que los métodos. De esta forma, todo lo que se desarrolla esta previamente testeado por pruebas de caja blanca, entonces es evidente que al final se tendrá un porcentaje alto de cobertura de pruebas. El hecho de tener un nivel de cobertura tan alto, por sí solo, no es evidencia de que realizamos TDD, ya que se pudieron haber realizado al final del proceso, pero si se analizan los commits en el repositorio, se puede observar que en la mayoría de los commits, se crean pruebas antes que los métodos.