

Universidad ORT

Escuela de Ingeniería

Obligatorio 2

Diseño de Aplicaciones 1

Autores:

Fernando Agustín Hernández Gobertti (173631) – Ingeniería en Telecomunicaciones

Carolina Rodríguez Fernández (173620) – Licenciatura en Sistemas



Docentes:

- Carlos Nicolás Fornaro
Rosado (Teórico)
- Leonardo Raúl Cecilia
Delgado (Práctico)



Fecha Límite de Entrega: 21 de Junio del 2018

Índice del Trabajo Realizado:

Introducción al Documento.....	2
Descripción General del Trabajo	3
Descripción General del Sistema.....	4
Mecanismos Generales del Sistema.....	23
Descripción y Justificación de Diseño.....	46
Cobertura de Pruebas Unitarias	62

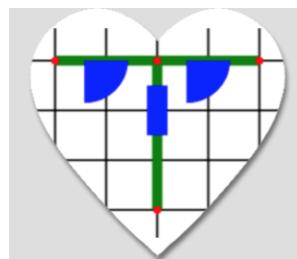
Introducción al Documento:

El motivo del presente documento es el de brindar información requerida sobre el trabajo Obligatorio 2 realizado para la Asignatura de Diseño de Aplicaciones 1. El trabajo fue realizado en *Visual Studios© Enterprise* (2015 y 2017) y *Microsoft SQL Management Studio© 2014*, todas ejecutándose sobre el sistema operativo Windows 10. Las herramientas accesorias utilizadas para realizar los subsiguientes diagramas fueron *StarUML v.2.8.1*, *Draw.io v.8.6.3* (versión online) y *Astah Professional 7.2*, siendo presentadas finalmente en el formato solicitado haciendo uso del último programa propietario.

A continuación se explicitan aspectos clave que fueron aclarados oportunamente como necesarios a documentar y justificar, en consecuencia del trabajo realizado, en conjunto a diagramas e imágenes coherentes y relacionados con cada tópico referido. Estos aspectos abarcan temáticas tales como:

- **Descripción del Trabajo:** Comunicación de la comprensión global otorgada a la consigna recibida, focalizándose en puntos relevantes a expandir en unidades subsiguientes.
- **Descripción del Sistema:** Comunicación general del programa realizado en cuanto a su arquitectura interna, relación entre componentes y proceso de desarrollo y mecanismos generales implementados. Utilización de *Entity Framework* a efectos de establecer persistencia por Base de Datos.
- **Diagramas del Sistema:** Realización de Diagramas de Paquetes, Diagramas de Clase (UML 2.0) y (a modo de ejemplo) Diagramas de Interacción ilustrativos del formato y comportamiento del Sistema.
- **Justificación de Diseño:** Explicación de causas y motivos de la arquitectura y comportamiento final del sistema, enfocados en la escalabilidad y mantenibilidad del Software, al igual que en los parámetros presentados en *Clean Code*. Se justificará también mediante Principios y Patrones de Diseño utilizados y mencionados en clase.

Los requerimientos relevados fueron actualizados continuamente en el Foro del sitio interactivo online *Moodle* de Aulas de la Facultad, en base a preguntas y respuestas de los distintos participantes de la asignatura. En el trabajo actual se procuró continuar con lo delegado en las mencionadas actualizaciones, siempre y cuando no contradigan con lo especificado en la letra del Obligatorio disponible en el mismo sitio.



Descripción General del Trabajo:

El trabajo presentado consiste en un software de diseño y mantenimiento de planos para Arquitectos, Diseñadores y sus Clientes. Este diseño se realiza en forma interactiva con la aplicación gráfica, y el resultado del mismo es publicado a la comunidad de forma automática.

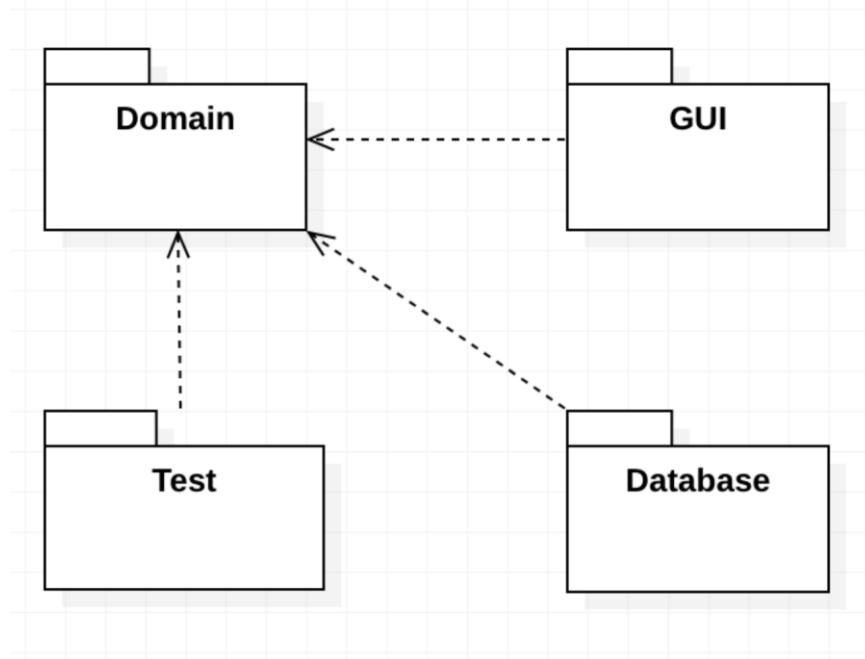
A efectos de cumplir con su objetivo, los Diseñadores y Arquitectos deben estar en contacto y relacionados con los posibles Clientes (para poderles realizar específicos diseños adecuados a sus gustos) y deben poder visualizar y descargar los planos ya realizados y publicados en la aplicación. Tanto los Diseñadores como los Arquitectos pueden crear o realizar cambios en planos (nuevos o ya existentes respectivamente), pero estos últimos poseen la funcionalidad adicional de poder firmarlos. Al momento de firmar un plano por primera vez, el Diseñador ya no será capaz de modificarlo (aunque sí lo podrá el Arquitecto) hasta que un Arquitecto lo firme por segunda vez. A su vez, los Diseñadores y Arquitectos pueden definir nuevas Aberturas. Los Clientes, por otro lado, son posibles de únicamente visualizar y descargar sus planos encargados, teniendo cuenta de todo el historial de los mismos asociados con su cuenta.

El programa consta también de un usuario Administrador, el cual tiene las capacidades de registrar cuentas de Diseñadores y Clientes, modificar sus datos y eliminarlos. Para poder hacer esto, el Administrador puede visualizar la Lista de Arquitectos, Diseñadores y Clientes de la aplicación, pudiendo en todo momento editarlos según considere apropiado. Finalmente, el Administrador se encuentra encargado de controlar la Oferta y Demanda del Mercado promovido por la aplicación, fijando los costos (valor requerido al Diseñador) y precios (solicitados al Cliente) de los materiales utilizados para la construcción resultante del plano. Estos materiales son diferentes para cada elemento a agregar al plano y se consideran homogéneos en cada caso (es decir, ningún elemento contiene más de un material).

Los elementos que actualmente se encuentran disponibles para colocar en el plano son: paredes, vigas y aberturas (puertas, ventanas y columnas decorativas). De estos, el más importante en cierto modo son las paredes, a efectos que es de quien dependen los otros elementos, puesto que no se pueden colocar vigas ni puertas ni ventanas en lugares donde no existan paredes, al igual que en lugares donde ya existen columnas. Cada elemento posee una nomenclatura particular y universal representable en el plano. Considérese también que a la vez de colocar elementos, todo Diseñador y Arquitecto tiene la posibilidad de eliminar elementos, knowing que no pueden existir vigas ni puertas ni ventanas si se remueven sus respectivas paredes de las cuales dependen. Todo Arquitecto puede firmar múltiples planos todas las veces que desee.

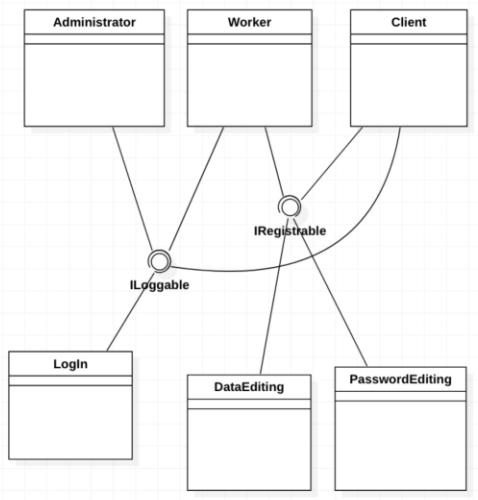
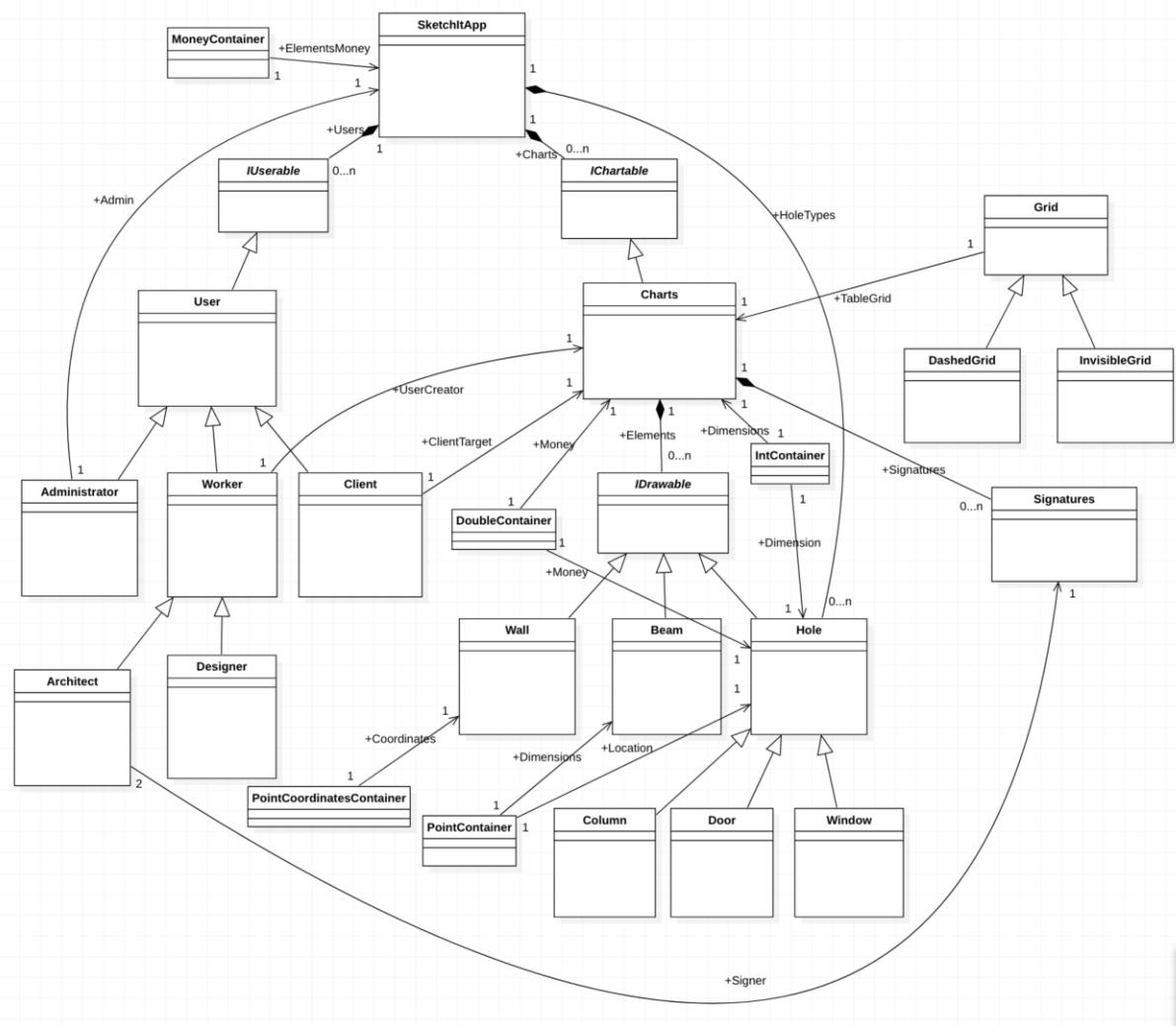
Descripción General del Sistema:

El sistema Solución elaborado se encuentra compuesto por 4 Proyectos: *Domain*, *GUI (Graphic User Interface)*, *Tests* y *Database*, cada uno con su motivo y funcionalidad marcada. Se consideró apropiado esta cantidad de Proyectos puesto que se partió de la premisa que, al aumentar la cantidad de los mismos (más allá del mínimo indispensable), aumentaría innecesariamente la complejidad del sistema. Cada proyecto (en este caso refiriéndose a *Domain*, *GUI* y *Database*) se encuentra a su vez formado por únicamente un *namespace*, del mismo nombre. Nótese también que, tal cual fue recomendado por los docentes y si bien no era obligatorio, se prosiguió a utilizar código en inglés durante todo el trabajo.



Se prosiguió de esta forma a utilizar la metodología de “*Code First*” al momento de implementar la persistencia por Base de Datos del Sistema, buscando de esta forma que el Repositorio de Datos (de Bajo Nivel) dependa de la Lógica de Negocios (*Domain*) y no al revés. Esto se consiguió realizando la Base de Datos una vez ya tenido el código de la lógica de la aplicación, considerando a la Base de Datos como un accesorio y una herramienta para alcanzar un fin, y no la sustentación de nuestro sistema.

Se buscó en todo momento distribuir lo más equitativamente posible las responsabilidades del Sistema, de tal manera que una Clase o unas pocas Clases no posean la mayoría de la Lógica del Sistema.

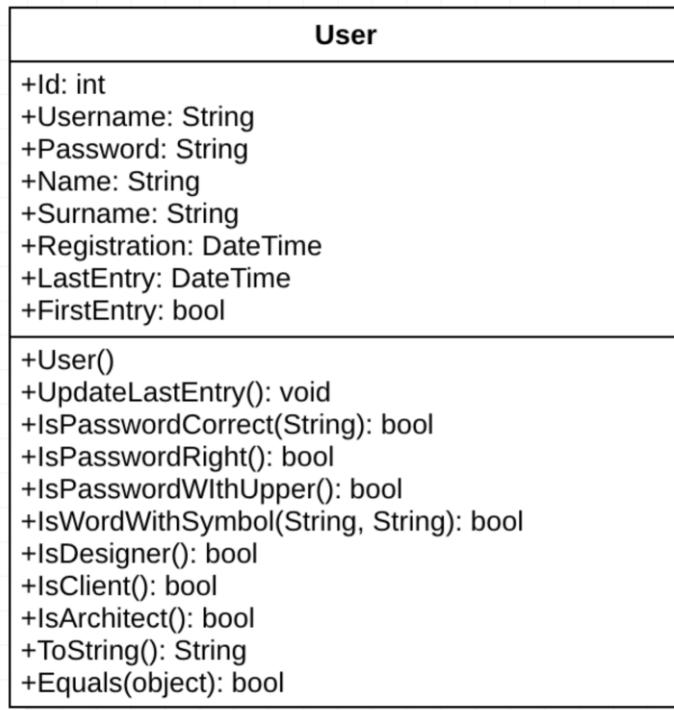


En el Proyecto *Domain* se encuentran todas las clases, atributos, objetos y métodos de formato *Class Library* correspondientes a la lógica del programa y la inicialización y relacionamiento entre los componentes del mismo. Nótese que, siempre tratando de realizar la menor cantidad de cambios respecto al Obligatorio 1 a efectos de argumentar (o criticar) un buen diseño inicial, existen algunos notorios que se proseguirán a explicar en la sección de “Justificación de Diseño”. Su correspondiente Diagrama de Clases UML (*Unified Modelling Language*) se encuentra representado a continuación. De esta forma, en este *namespace* se encuentran las siguientes clases e interfaces:

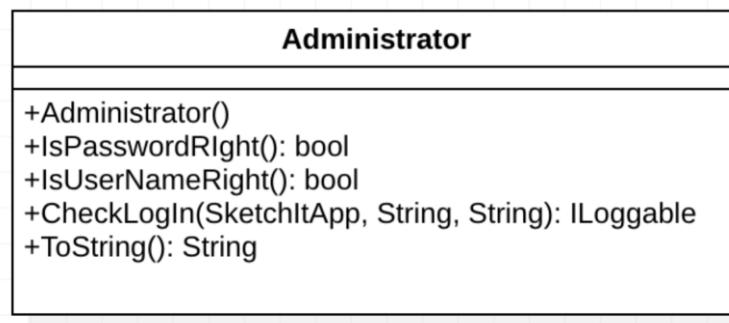
- **SketchItApp.cs:** Clase central del programa, a partir de la cual surge la instancia del programa utilizado al iniciar la aplicación y el cual contiene los componentes básicos de la misma (acceso a Usuarios y acceso al Servicio: los Planos y los tipos de Aberturas). Siguiendo se presentan los métodos y atributos de la Clase *SketchItApp*.

SketchItApp
+Users: List<IUsable> +HoleTypes: List<Hole> +Admin: Administrator +Charts: List<IChartable> +ElementsMoney: MoneyContainer +instanceOfProgram: SketchItApp
+GetInstance(): SketchItApp +SketchItApp() +GetHoleFromHoleTypes(String): Hole +UpdateChartsData(): void +GetChartsFromClient(Client): List<IChartable> +IsChartNotYetRegistered(Chart): bool +GetClientsFromDesigner(Designer): List<Client> +GetChartsFromDesigner(Designer): List<IChartable> +IsClientAvailableToAdd(Client): bool +IsDesignerAvailableToAdd(Designer): bool +IsArchitectAvailableToAdd(Architect): bool +GetChartFromList(Chart): Chart +GetClientFromList(Client): Client +GetDesignerFromList(Designer): Designer +GetArchitectFromList(Architect): Architect +OnClosingProgram(): void +IsClientRemovable(Client): bool +IsDesignerRemovalbe(Designer): bool +IsDesignerNotYetRegistered(Designer): bool +IsClientNotYetRegistered(Client): bool +IsArchitectNotYetRegistered(Architect): bool +CalculateCostAndPrice(IChartable): double[2] +DivideChartIntoSingleSignatures(List): List<Chart> +AddChartsSigned(Chart, Architect): void +GetDesignersFromUsers(): List<IUsable> +GetClientsFromUsers(): List<IUsable> +GetArchitectsFromUsers(): List<IUsable>

- **User.cs:** Clase base de *Administrator.cs*, *Worker.cs* y *Client.cs* que define al Usuario del Sistema. Dos Usuarios son iguales si su Nombre de Usuario es el mismo o si su Nombre y Apellido son los mismos.



- **Administrator.cs:** Clase que define al Administrador del Sistema, con sus potestades y responsabilidades únicas ya mencionadas.



- **Worker.cs:** Clase base de *Designer.cs* y *Architect.cs* que define las características comunes a ambos, tales como los criterios de registro correcto y de igualdad. Dos Trabajadores son iguales si su Nombre de Usuario es el mismo o si su Nombre y Apellido son los mismos. El nombre de la clase fue

elegido de modo de diferenciar a los que brindan el servicio (los trabajadores o servidores) respecto de los que lo reciben (los clientes). Se consideró que el nombre Server.cs sería poco apropiado ya que podría dar lugar a confusiones en un futuro.

Worker
<pre>+IsWellRegistered(): bool +InitEditing(Form): String[5] +GetDataEditing(String[3]): void +PasswordEquals(String): bool +StoreData(SketchItApp, IRegistrable): bool +ChangePassword(SketchItApp, String): void +AddChartsHelped(ref SketchItApp): void +AddClientsHelped(ref SketchItApp): void +Equals(object): bool</pre>

- **Client.cs:** Clase que define al Cliente del Sistema, con sus también características específicas mencionadas. Dos Clientes son iguales si su Nombre de Usuario es el mismo, o si su Nombre y Apellido son los mismos, o si su Cédula de Identidad es la misma o si su Teléfono es el mismo o si su Dirección es la misma.

Client
<pre>+NumberID: String +Telephone: String +Address: String +IsWellRegistered(): bool +InitEditing(Form): String[5] +GetDataEditing(String[5]): void +StoreData(SketchItApp, IRegistrable): bool +ChangePassword(SketchItApp, String): void +PasswordEquals(String): bool +CheckLogIn(SketchItApp, String, String): ILoggable +IsNumberRight(): bool +CheckNewNumberID(): bool +CheckNumberWithCeros(String, int): bool +CheckOldNumberID(String): bool +CheckRangePositiveNumber(int, int): bool +IsTelephoneRight(): bool +IsTelephoneBeginningRight(): bool +ToString(): String +Equals(object): bool +IsDesigner(): bool +IsClient(): bool +IsArchitect(): bool</pre>

- ***Architect.cs:*** Clase que define al Arquitecto del Sistema, con sus también características particulares mencionadas.

Architect
+ChartsSigned: int
+StoreData(SketchItApp, IRegistrable): bool
+ChangePassword(SketchItApp, String): void
+CheckLogIn(SketchItApp, String, String): ILoggable
+IsDesigner(): bool
+IsClient(): bool
+IsArchitect(): bool

- ***Designer.cs:*** Clase que define al Diseñador del Sistema, con sus también características particulares mencionadas.

Designer
+ChartsHelped: int
+ClientsHelped: int
+AddChartsHelped(ref SketchItApp): void
+AddClientsHelped(ref SketchItApp): void
+StoreData(SketchItApp, IRegistrable): bool
+ChangePassword(SketchItApp, String): void
+CheckLogIn(SketchItApp, String, String): ILoggable
+IsDesigner(): bool
+IsClient(): bool
+IsArchitect(): bool

- ***Chart.cs:*** Clase que define al Plano a dibujar a modo de Grilla, con una lista de elementos dibujables, un Diseñador que lo crea, un Cliente que lo solicita (objetivo), un capital asociado (costo y precio), un conjunto de firmas y una grilla personalizable. A efectos prácticos, se prefirió utilizar el término *Chart*, que en inglés significa realmente “Gráfico”. Dos planos son iguales si sus elementos son los mismos y tienen el mismo Diseñador que los creó, y dirigidos al mismo Cliente objetivo.

Chart
<pre>+Id: int +Name: String +Elements: List<IDrawable> +Signatures: List<Signature> +ClientTarget: Client +UserCreator: Worker +Dimensions: IntContainer +Money: DoubleContainer +TableGrid: Grid +Chart(Worker, Client) +DrawGrid(ref Graphics, int): void +DrawElements(ref Graphics): void +GetClientTarget(): Client + GetUserCreator(): Worker +GetSignatures(): List<Signature> +GetElements(): List<IDrawable> +IsSigned(): bool +IsCompletelySigned(): bool +IsWallAddable(Wall): bool +RemoveWall(Wall): void +GetPixels(): int[2] +IsChartWellRegistered(): bool +IsEmpty(): bool +IsWallNotYetRegistered(Wall): bool +IsBeamNotYetRegistered(Beam): bool +IsWallWellRegistered(Wall): bool +IsPointOnChart(Point): bool +IsIntersection(): bool +GetIntersection(Wall, Wall): Point +AddBeamsDueToLength(Wall): void +SortElements(): List<IDrawable> +IsBeamNotYetRegisteredNearDoor(Door): bool +IsWindowNotYetRegisteredNearDoor(Door, int): bool +IsNoBigDoorInSurroundingsOfDoor(Door, int): bool +GetBigDoorOnPoint(Point): Door +IsDoorAvailable(Door, int): bool +EliminateHoles(Wall): void +EliminateBeams(Wall): void +IsHoleInBetween(Wall): bool +IsBeamNotYetRegisteredWindow(Window, int): bool +IsDoorNotYetRegisteredNearWindow(Window, int): bool +IsWindowAvailable(Window, int): bool +IsWindowsWithSameOrientationWall(Window): bool +IsDoorCrossingWall(Door): bool +DivideWallIntoSmaller(Wall, Point): Wall +IsPossibleIntersection(Wall, Wall): bool +CaptureIntersectionPoint(Wall, Wall): Point +GetWallFromList(IDrawable): Wall +IsCrossIntersection(Wall, Wall): bool +IsTeHorizontalIntersection(Wall, Wall): bool +IsWindowNotYetRegistered(Window): bool +AddWindow(Window): void +AddDoor(Door): void +IsDoorNotYetRegistered(Door): bool +IsTeVerticalIntersection(Wall, Wall): bool +IsWallBetweenWall(Wall, Wall): bool +IsBeamToChartAddable(Beam): bool +GetWallOfPoint(Point): Wall +CalculateMoney(double[,]): double[2] +ToString(): String +Equals(object): bool +CountElements(): int[5] +IsColumnAvailable(Column): bool +AddColumn(Column): void +IsColumnNotYetRegistered(Column): bool +GetSignatureWithOneSigning(): Signature +IsPointOnHole(Point): bool</pre>

- **Wall.cs:** Clase que define al elemento del Plano conocido como Pared, dibujado como una línea gruesa verde con una coordenada inicial de origen y final de destino. Nótese que inherentemente la Pared se encuentra orientada, factor fundamental a considerar al momento de implementar las funciones de intersección. Dos Paredes son iguales si sus dirección son iguales (es decir, si pasan por la misma línea).

Wall
+Coordinates: PointCoordiantesContainer
+Wall(Point[2])
+Draw(ref Graphics): void
+IsWall(): bool
+CountElements(int[5]): void
+CalculateLength(): int
+CheckLength(): int
+ApproximateWall(Point[2]): Point[2]
+ApproximateCoordinates(int): int
+IsInSameLine(): bool
+Equals(object): bool
+AddLengthToOrigin(int): Point
+IsHorizontal(): bool
+IsVertical(): bool
+IsNullLength(): bool
+IsPointInsideWall(Point): bool
+CalculateMoney(double[,]): double[2]
+NextPointOfWall(Point): Point
+IsPointOnHole(Point): bool
+IsBigDoorOfPoint(Point): bool
+UpdateMoneyData(SketchItApp): void
+GetElementMoney(): double[2]

- **Beam.cs:** Clase que define al elemento del Plano conocido como Viga, dibujado como un punto grueso rojo de coordenadas determinadas. Dos Vigas son iguales si sus coordenadas son las mismas.

Beam
+Dimensions: PointContainer
+Beam(Point)
+CalculateMoney(double[,]): double[2]
+Draw(ref Graphics): void
+IsPointInsideWall(Point): bool
+IsWall(): bool
+CountElements(int[]): void
+Equals(object): bool
+IsPointOnHole(Point): bool
+IsBigDoorOnPoint(Point): bool
+UpdateMoneyData(SketchItApp): void
+GetElementMoney(): double[2]

- **Hole.cs:** Clase base de *Column.cs*, *Door.cs* y *Window.cs* que define al elemento del Plano conocido como Abertura. Dos Aberturas son iguales si sus coordenadas son las mismas.

Hole
<pre>+Money: DoubleContainer +Name: String +Location: PointContainer +Dimension: IntContainer +HeightFromFloor: double +SetActualSize(double, double): void +ApproximateCoordinate(int): int +GetAppropriateLocation(Point): Point +IsSmallHole(): bool +IsMediumHole(): bool +IsBigHole(): bool +Equals(object): bool +Draw(ref Graphics): void +IsPointInsideWall(Point): bool +IsWall(): bool +CalculateMoney(double[,]): double[2] +CountElements(int[5]): void +IsPointOnHole(Point): bool +IsBigDoorOfPoint(Point): bool +UpdateMoneyData(SketchItApp): void +GetElementMoney(): double[2]</pre>

- **Column.cs:** Clase que define al elemento del Plano conocido como Columna, dibujado como un punto marrón (más grueso que las Vigas) de coordenadas determinadas.

Column
<pre>+Column(Point) +Draw(ref Graphics): void +IsPointInside(Point): bool +CalculateMoney(double[,]): double[2] +IsWall(): bool +CountElements(int[5]): void +IsPointOnHole(Point): bool +IsBigDoorOfPoint(Point): bool +UpdateMoneyData(SketchItApp): void +GetElementMoney(): double[2]</pre>

- **Door.cs:** Clase que define al elemento del Plano conocido como Puerta, definido por una coordenada determinada y una orientación en sentido antihorario (0°, 90°, 180°, 270°).

Door
+OrientationDoor: int
+Door(Point)
+IsPointInsideWall(Point): bool
+IsWall(): bool
+CalculateMoney(double[,]): double[2]
+Draw(ref Graphics): void
+RotateDoor(): void
+GetDoorFromHole(Hole): void
+GetBorderPoints(): Point[2]
+IsPointOnHole(Point): bool
+IsBigDoorOfPoint(Point): bool
+UpdateMoneyData(SketchItApp): void
+GetElementMoney(): double[2]

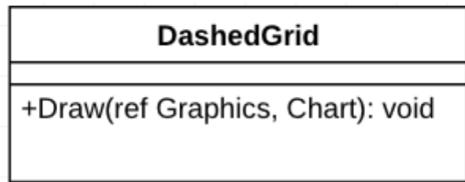
- **Window.cs:** Clase que define al elemento del Plano conocido como Ventana, definido por una coordenada determinada y una orientación (vertical u horizontal).

Window
+OrientationWindow: int
+Window(Point)
+IsPointInsideWall(Point): bool
+IsWall(): bool
+CountElements(int[5]): void
+CalculateMoney(double[,]): double[5]
+Draw(ref Graphics): void
+RotateWindow(): void
+GetWindowFromHole(Hole): void
+GetBorderPoints(): Point[2]
+IsPointOnHole(Point): bool
+IsBigDoorOfPoint(Point): bool
+UpdateMoneyData(SketchItApp): void
+GetElementMoney(): double[2]

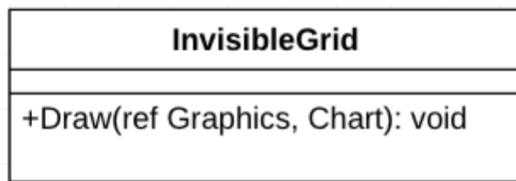
- **Grid.cs:** Clase que define la representación gráfica de la Grilla personalizada del *Chart* cuando ésta es seleccionada para que posea líneas continuas y sólidas.

Grid
+Draw(ref Graphics, Chart): void

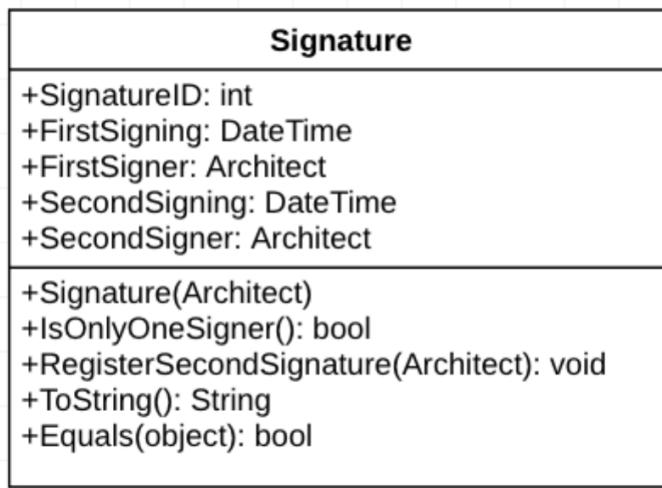
- **DashedGrid.cs:** Clase que define la representación gráfica de la Grilla personalizada del *Chart* cuando ésta es seleccionada para que posea líneas discontinuas y punteadas.



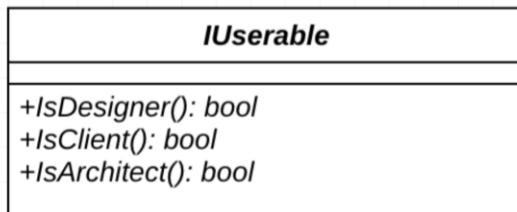
- **InvisibleGrid.cs:** Clase que define la representación gráfica de la Grilla personalizada del *Chart* cuando ésta es seleccionada para que no posea líneas, para lo cual solo se dibuja el recuadro delimitador del Plano.



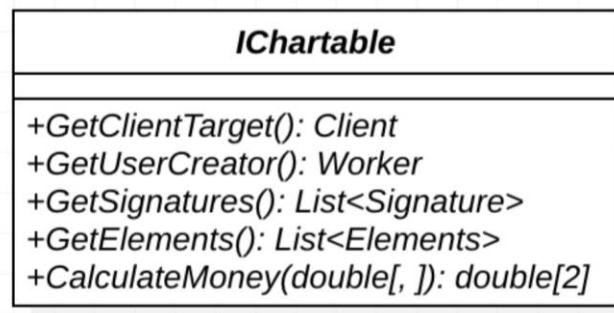
- **Signature.cs:** Clase que define a la composición de una Firma del Plano. Todo Plano puede tener varias firmas, y todas ellas están compuestas por un Primer Firmante y un Segundo Firmante (y sus correspondientes tiempos de Firmado), ambos Arquitectos. Para un mismo Plano solo puede haber una Firma con ningún Segundo Firmante.



- **IUsable.cs:** Interfaz que contiene la firma de métodos relacionados con los Usuarios. Es el nexo entre la Clase *SketchItApp.cs* (Sistema) y *User.cs*, de tal forma de generar otro punto de articulación para un posible agregado futuro de otros tipos de Entidades completamente diferentes a los Usuarios registrados, pero que se busquen tratar de la misma forma. Nótese que se debió *Refactorear* a Clase Abstracta para que sea posible implementarla en *Entity Framework* (más de esto en siguientes unidades).



- **IChartable.cs:** Interfaz que contiene la firma de métodos relacionados con los Planos. Es el nexo entre la Clase *SketchItApp.cs* (Sistema) y *Chart.cs*, de tal forma de generar otro punto de articulación para un posible agregado futuro de otros tipos de Entidades completamente diferentes a los Planos generados, pero que se busquen tratar de la misma forma. Nótese que se debió *Refactorear* a Clase Abstracta para que sea posible implementarla en *Entity Framework* (más de esto en siguientes unidades).



- **IRegistrable.cs:** Interfaz que contiene la firma de métodos relacionados con el Registro de Trabajadores (Diseñadores y Arquitectos) y Clientes, a utilizar de forma polimórfica.

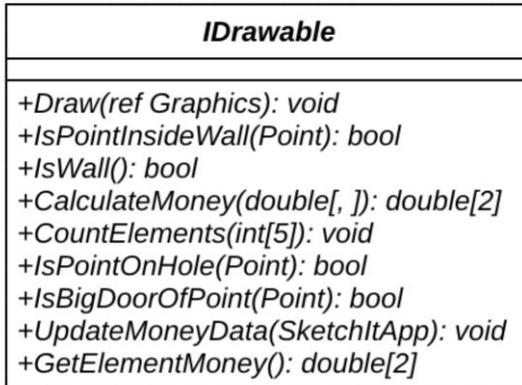
**IRegistrable**

```
+IsWellRegistered(): bool
+StoreData(SketchItApp, IRegistrable): bool
+GetDataEditing(String[3]): void
+ChangePassword(SketchItApp, IRegistrable): void
+PasswordEquals(String): bool
+InitEditing(Form): String[2]
```

**ILoggable**

```
+CheckLogIn(SketchItApp, String, String): ILoggable
```

- ***ILoggable.cs*:** Interfaz que contiene la firma de métodos relacionados con la Identificación (*Log In*) del Administrador, y de Arquitectos, Diseñadores y Clientes, a utilizar de forma polimórfica.



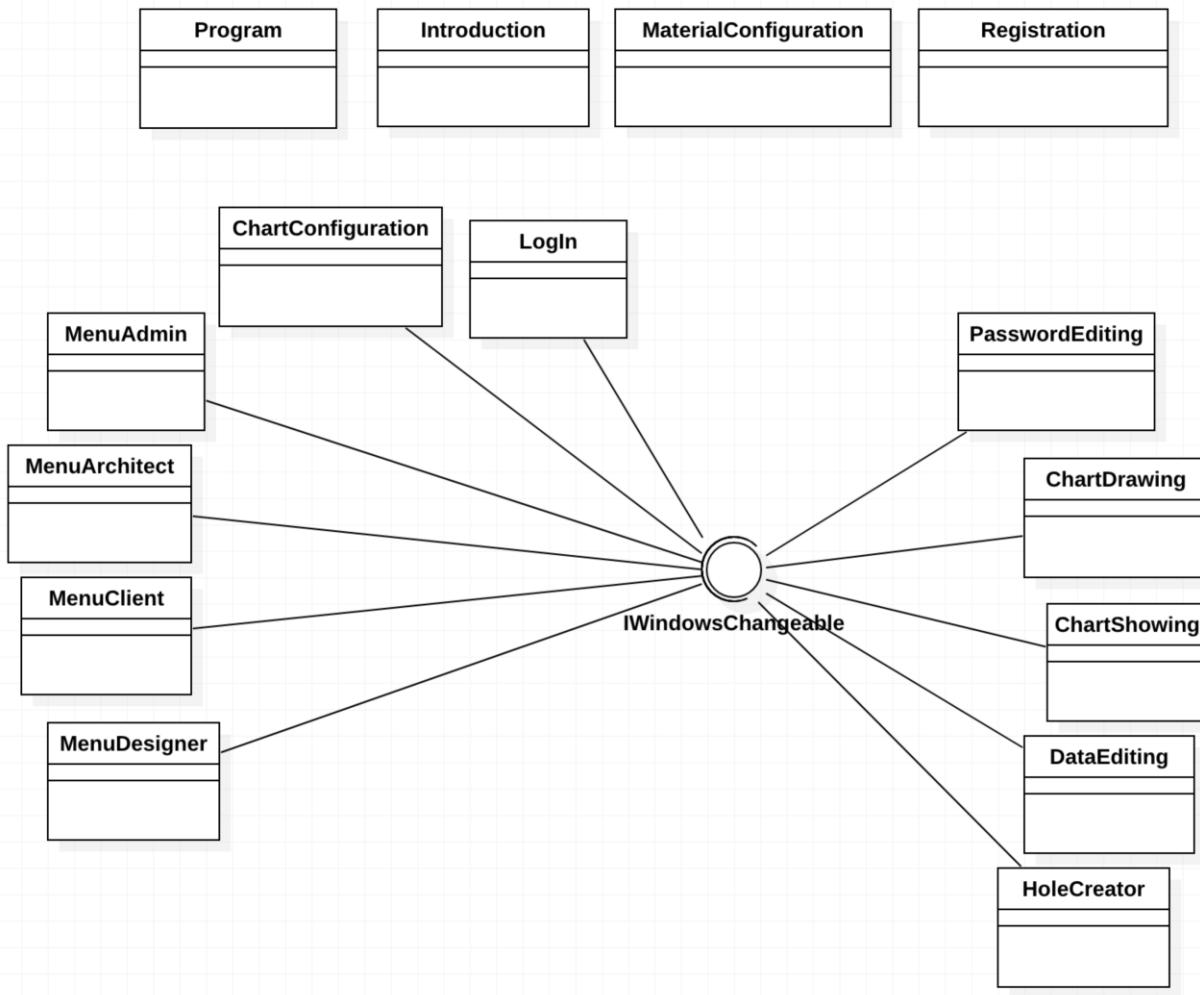
-

- Clases Contenedoras:** Existen un total de 5 Clases *Container* consistentes en aglomeraciones de entidades que, en otro caso, podrían existir en una entidad específica proporcionada por Visual Studio. Sin embargo, debieron crearse estas clases por motivos de compatibilidad con *Entity Framework*, debido a la no persistencia de elementos del tipo *Point*, o *Arrays* (más de esto adelante).

DoubleContainer	IntContainer	MoneyContainer
+Id: int +Cost: double +Price: double +DoubleContainer()	+Id: int +Width: int +Length: int +IntContainer()	+Id: int +CostWall: double +PriceWall: double +CostBeam: double +PriceBeam: double +CostDoor: double +PriceDoor: double +CostWindow: double +PriceWindow: double +CostColumn: double +PriceColumn: double +MoneyContainer()
PointCoordinatesContainer	PointContainer	
+Id: int +OriginX: int +OriginY: int +DestinationX: int +DestinationY: int +PointCoordinatesContainer()	+Id: int +X: int +Y: int +PointContainer()	

- Exceptions:** Este proyecto posee dos excepciones (consideradas como las realmente necesarias), consistentes en las clases de *AlreadyExistingUserException.cs* e *InvalidUserException.cs*.

En el Proyecto GUI se colocaron todas las clases, atributos, objetos y métodos de formato *Windows Form* relacionados con la representación e interacción gráfica con el usuario, haciendo uso (dependiendo) de la lógica y clases existentes en *Domain*. Su correspondiente Diagrama de Clases UML (*Unified Modelling Language*) se encuentra representado en la Figura inferior. De esta manera, en este *namespace* existen las siguientes clases e interfaces:



- ***Program.cs***: Clase que contiene el *Main* del programa, y la que determina que se indique a GUI como el *StartUp Project*.
- ***Introduction.cs***: Clase que inicializa la primer *Form* de la Aplicación, consistente en un mensaje de bienvenida y presentación.
- ***LogIn.cs***: Clase que inicializa y determina la Identificación (*Log In*) de Usuarios y deriva a cada Usuario registrado a sus correspondientes *Forms*.

- **MenuAdmin.cs:** Clase que inicializa y despliega información para el Administrador una vez autenticado, y satisface las necesidades del mismo en cuanto a modificaciones a realizar al contenido ya registrado o por registrar al Sistema.
- **Registration.cs:** Clase que facilita al Administrador a registrar Diseñadores y Clientes.
- **DataEditing.cs:** Clase que permite la edición de los datos de Diseñadores y Clientes (nótese que los datos del Administrador no se pueden modificar). Estos datos pueden ser modificables por el Administrador en cualquier momento y de cualquier Usuario (salvo de sí mismo), mientras que cada Diseñador y Cliente pueden modificar sus datos únicamente la primera vez que se identifican al Sistema.¹
- **MaterialConfiguration.cs:** Clase que hace posible la fijación de Coste y Precio de cada material ofrecido por el Sistema, que como se mencionó antes se corresponde con uno diferente por cada elemento del Plano.
- **MenuDesigner.cs:** Clase que despliega la información relevante y permitida para el Diseñador una vez autenticado, y la cual le ofrece la realización de sus responsabilidades ya explicitadas.
- **MenuArchitect.cs:** Clase que despliega la información relevante y permitida para el Arquitecto una vez autenticado, y la cual le ofrece la realización de sus responsabilidades ya explicitadas.
- **HoleCreator.cs:** Clase que permite la creación de nuevos tipos de Aberturas (limitándose a otras Puertas o Ventanas) por parte de Arquitectos o Diseñadores.²
- **ChartConfiguration.cs:** Clase que permite la configuración inicial de un Plano, una vez que el Diseñador decidió crear y registrar uno nuevo.
- **ChartDrawing.cs:** Clase encargada de la funcionalidad principal del Sistema, consistente en dibujar y guardar los cambios realizados a un Plano ya configurado.
- **MenuClient.cs:** Clase que muestra información relevante para el Cliente una vez autenticado.
- **ChartShowing.cs:** Clase que posibilita mostrar el Plano realizado (si el Usuario es Diseñador) o encargado (si el Usuario es Cliente) y descargarlo en formatos: .bmp, .jpg, .png y .gif.
- **PasswordEditing.cs:** Clase que facilita el cambio de contraseña de todo Usuario (a excepción del Administrador) en cualquier momento desde su Menú correspondiente.
- **IWindowsConfigurable.cs:** Interfaz que viabiliza el comportamiento relativo de *DataEditing.cs*, *PasswordEditing.cs* y *ChartShowing.cs* respecto a que tipo de Usuario ingresa a su *Form*. A su vez,

¹ Discusión en Foro de Aulas del Obligatorio 1: <https://aulas.ort.edu.uy/mod/forum/discuss.php?d=73315>

² Discusión en Foro de Aulas del Obligatorio 2: <https://aulas.ort.edu.uy/mod/forum/discuss.php?d=74591>

permite implementar el comportamiento relativo en ChartDrawing.cs en caso que el Usuario ingresante sea un Diseñador o un Arquitecto (para el caso de este último se le debe preguntar si desea firmar el Plano y en caso afirmativo, firmarlo).

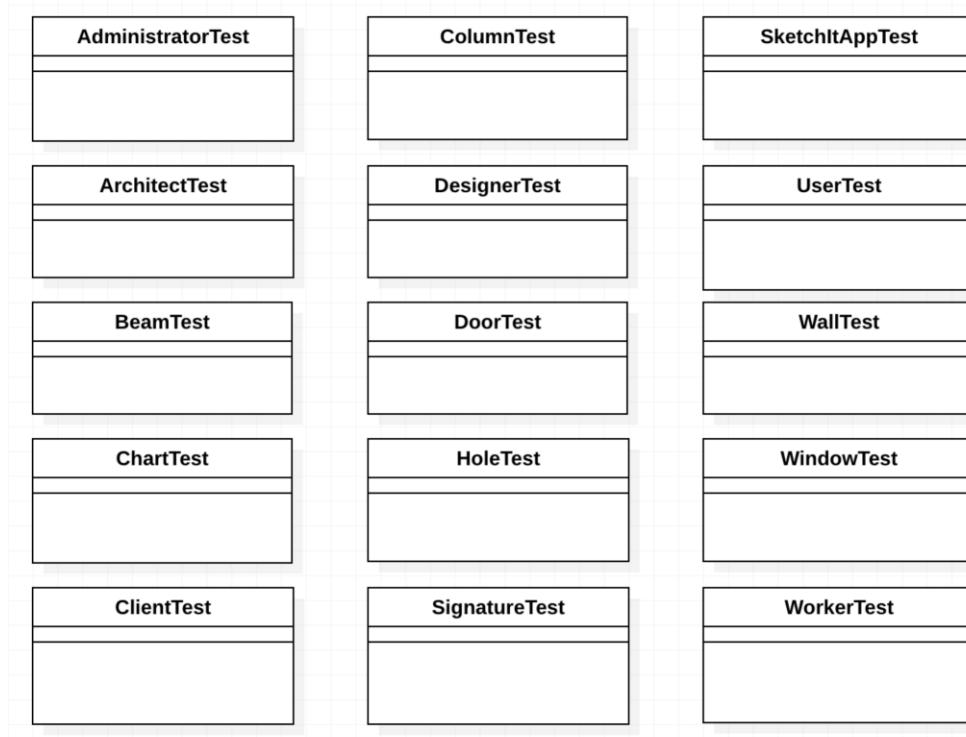


IWindowsChangeable

+ChangeToPreviousForm(Form): void
+AskArchitectToSignChartOrDefault(Form, Chart): void

- **Exceptions:** Este proyecto posee diez excepciones (consideradas como las realmente necesarias) de modo de cumplir con el criterio de *Clean Code* de “Utilizar Excepciones en vez de mensajes de error”. Estas excepciones (con nombres auto explicativos) corresponden a las clases:

- *BlankSpaceException.cs*
- *ChartConfigureException.cs*
- *ChartEmptyException.cs*
- *NoChartToShowSelectedException.cs*
- *NoClientSelectedException.cs*
- *NoSignedChartSelectedException.cs*
- *NoUnsignedChartSelectedException.cs*
- *NoUserSelectedException.cs*
- *TestDataLogException.cs*
- *UserRegistrationException.cs*



En el Proyecto *Tests* se colocaron todas las *TestClass*, *TestInitializers*, *TestMethods* y *TestCleanups* necesitados para establecer un considerado correcto testeo del Sistema, y con una buena cobertura del código. Se crearon una clase de Test por cada clase del Proyecto *Domain* ya descripto, variando la cantidad de *TestMethods* internos de cada *TestClass*. Su correspondiente Diagrama de Clases UML (*Unified Modelling Language*) se encuentra representado en la Figura superior, si bien se entiende que no resulta sumamente útil para el caso actual (por lo que solo se representan las clases y no los métodos, para lo cuales se invita a ir al código). Más del tema Pruebas en la última unidad del documento.

Los nombres de los objetos y métodos utilizados procuraron ser en el transcurso del trabajo de manera concisa y estructurada, manteniendo un orden coherente entre los mismos. Así, si por ejemplo en *ChartDrawing.cs* se utilizó un objeto *Panel chartDraw*, un objeto *Chart drawingChart* y un objeto *Graphic chartGraphic*, entonces en *ChartShowing.cs* se utilizó un objeto *Panel chartShow*, un objeto *Chart showingChart* y un objeto *Graphic chartGraphic*.

De la misma forma, de acuerdo a *Clean Code*, se buscó que los nombres de Atributos, Métodos, Objetos y Clases sean pronunciables, focalizados y fácil de comprender su motivo. Igualmente, los comentarios fueron restringidos a solo en casos totalmente necesarios para la comprensión del código. Finalmente, se preocupó para que los métodos recibieran como máximo (y en contadas ocasiones) 3 parámetros, siendo en su gran mayoría métodos que aceptan ningún o 1 parámetro.

Es importante remarcar que al día de la entrega no se ha encontrado ningún bug notorio en el programa, pero se pretende seguir testeando, teniendo siempre presente que “la ausencia de fallos no es prueba de ausencia”, es decir, que con no descubrir los problemas no significa que no los haya.

Casos de Prueba:

- **Administrador:**

- *Username*: admin
 - *Password*: admin2018

- **Diseñador:**

- *Username*: Fer5050
 - *Password*: Fer.5050

- **Cliente:**

- *Username*: CaroRod3
 - *Password*: Caro.Rod3

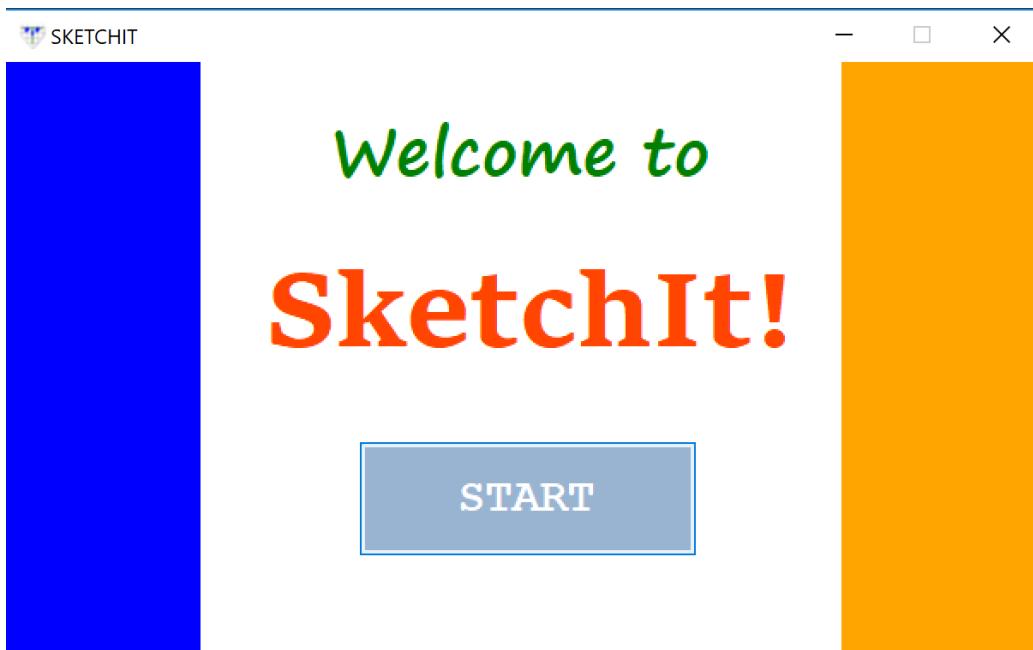
- **Arquitecto:**

- *Username*: Fer55
 - *Password*: Fer.55

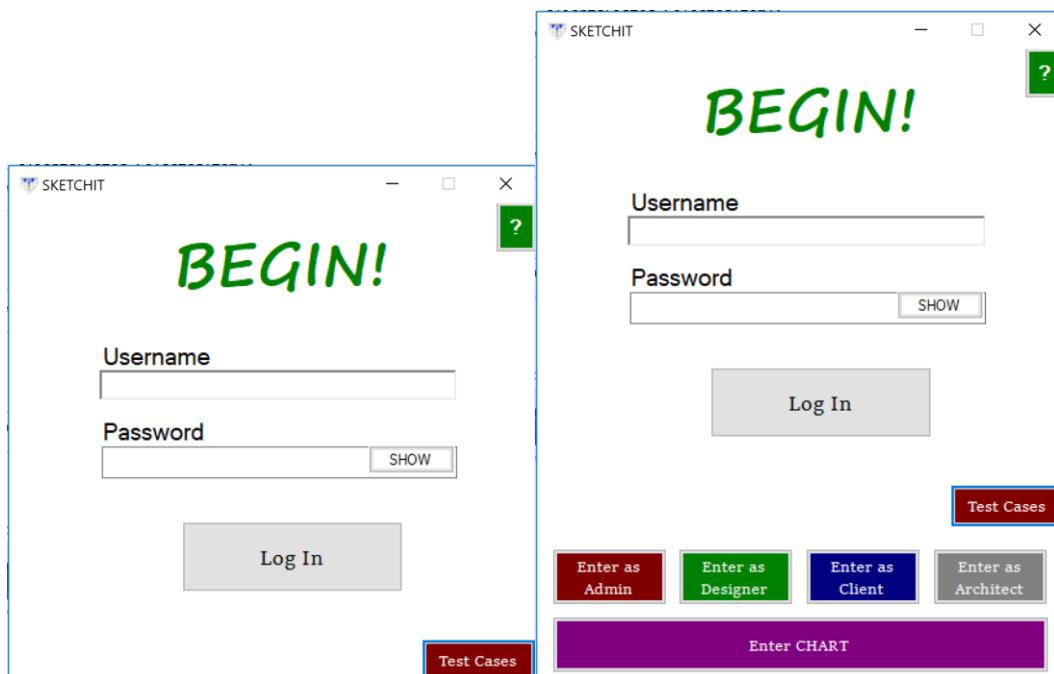
Nota: Los presentados son ejemplos de sesiones de cada tipo de Usuario (a excepción del Administrador que es único). Siempre que se desee se podrá ingresar como otro Usuario, pero se deberá previamente fijar en su correspondiente *Username* y *Password* ingresando como Administrador y observando las Listas de Diseñadores y Clientes en el Sistema.

Mecanismos Generales del Sistema:

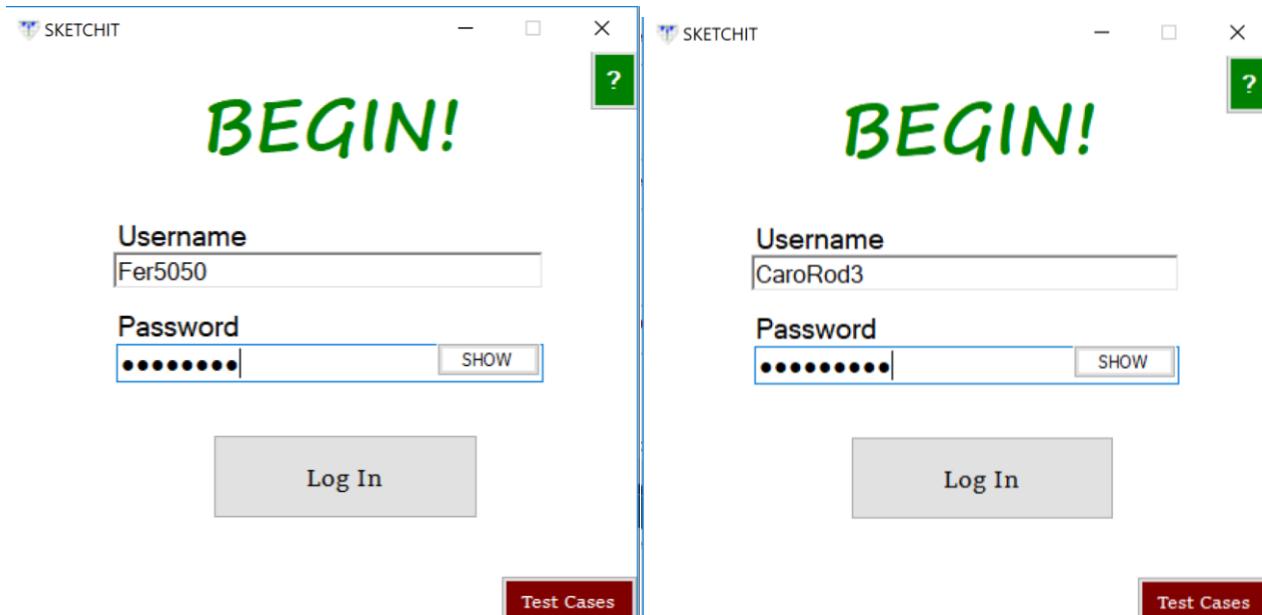
En la presente unidad se explicitarán los mecanismos del sistema en cuanto al funcionamiento de cada sección por la que transcurre el ingresante. Este funcionamiento fue elegido principalmente de manera de continuar con los requerimientos planteados, y basándose en criterios propios y aplicaciones análogas a la creada vistas en el mercado, tales como: *AutoCAD*, *LayOut*, *SketchUp* y *StyleBuilder*.



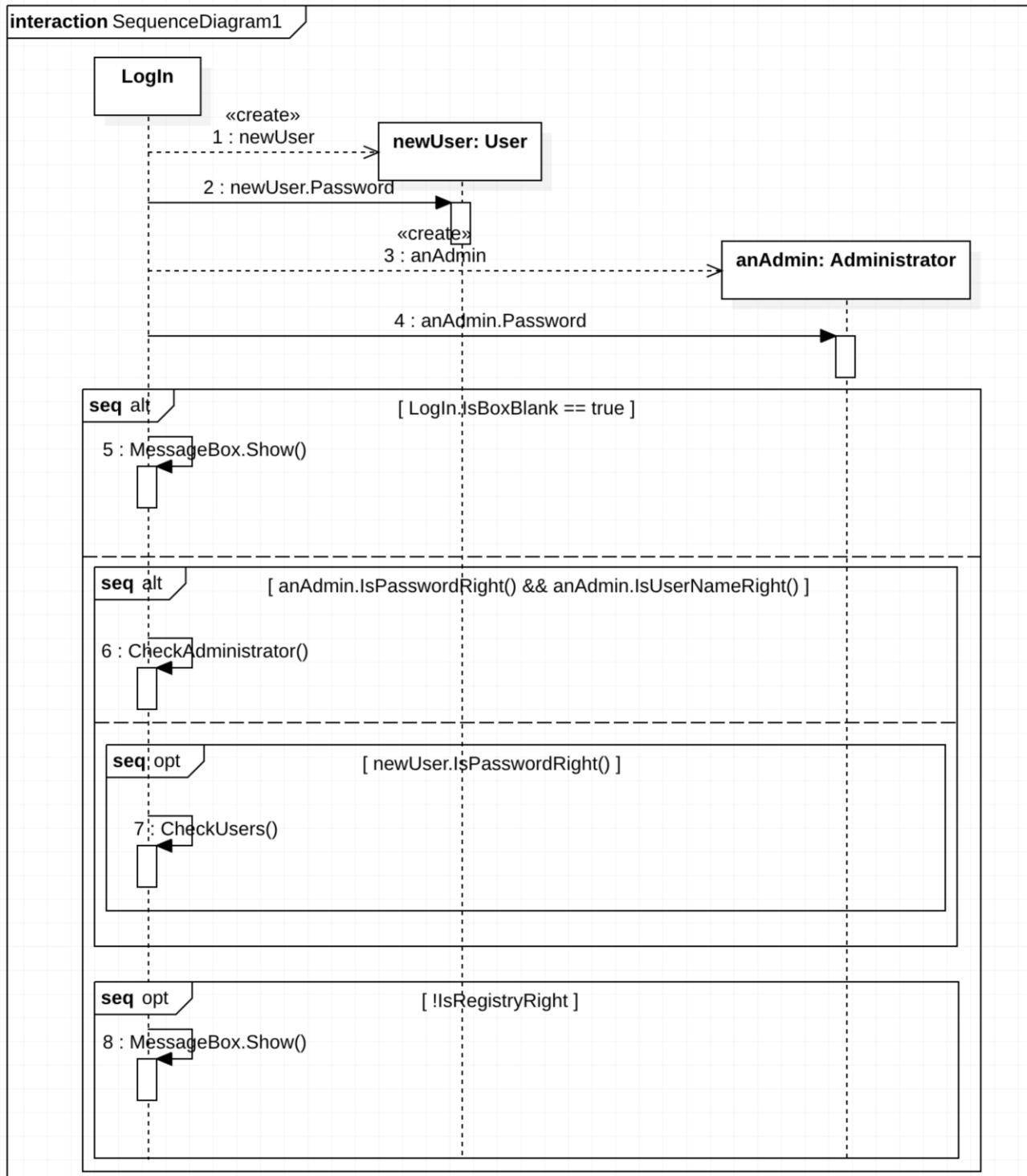
El programa inicia en la *Form* de *Introduction.cs*, donde se instancia un nuevo objeto de la clase *SketchItApp.cs*, el cual será enviado de aquí en más como parámetro por todas las *Forms* de tal forma de conservar y rastrear el estado del Sistema. El programa posee un administrador, al igual que una lista de Usuarios (Arquitectos, Diseñadores y Clientes) y Planos, las cuales al iniciar se encuentran predeterminadamente vacías. Por ello, y de tal manera de implementar persistencia, se utiliza la herramienta de *Entity Framework* de manera de guardar y cargar datos desde una Base de Datos gestionada por el Framework haciendo uso de *Lazy Loading*. Esto implica el proceso mediante el cual una entidad o colección de entidades se carga automáticamente desde la base de datos la primera vez que se accede a una propiedad que hace referencia a la entidad o entidades. Como es solicitado en la letra del Obligatorio, se presenta una Base de Datos vacía al igual que una Base de Datos con datos de prueba para uso de los correctores.

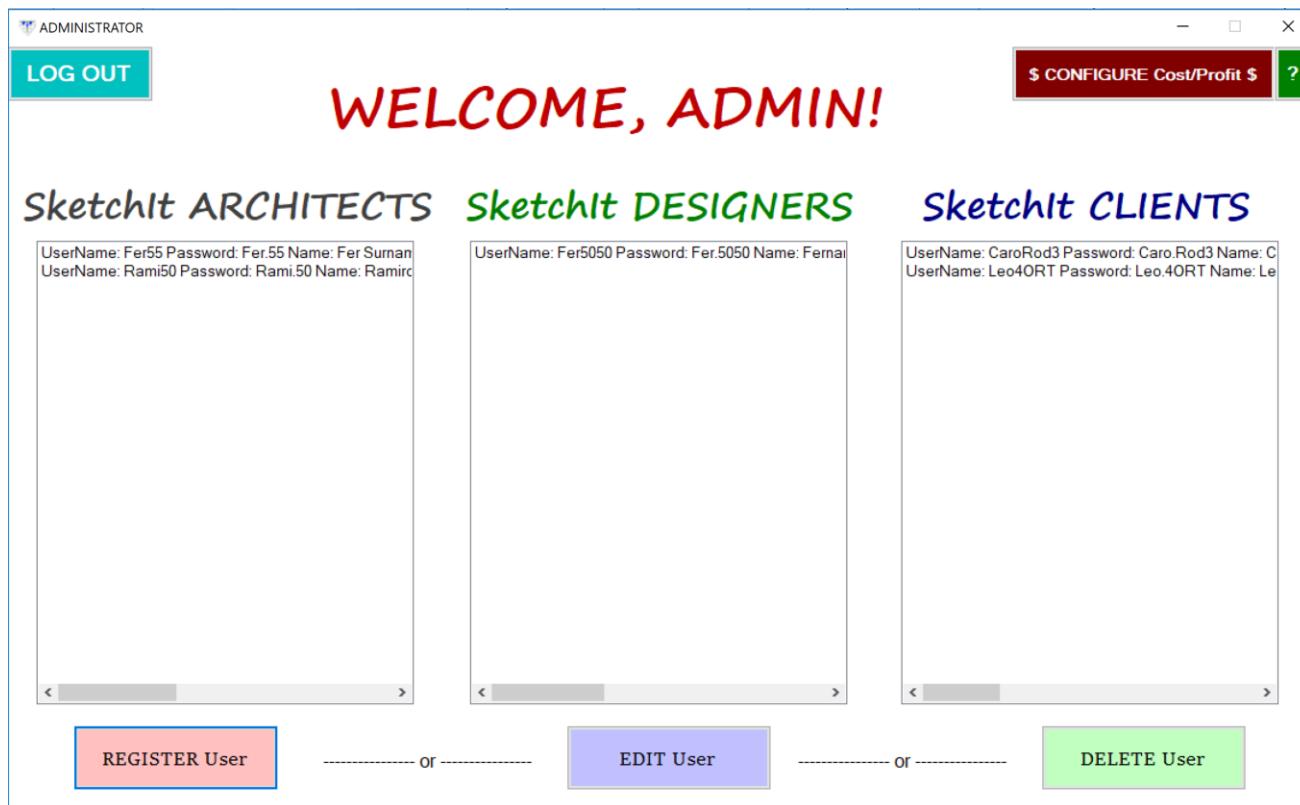


Una vez iniciado el programa, se le solicita al ingresante a identificarse ingresando su nombre de usuario y contraseña, removiendo (como fue solicitado por el docente) la necesidad de tenerse que autenticar previamente como Arquitecto, Diseñador o Cliente. En caso de ser Administrador, el ingresante deberá entrar con *Username* "admin" y *Password* "admin2018". En caso de ingreso erróneo (casillas en blanco o usuario no registrado o no concordancia entre *Username* y *Password*), el sistema le notifica al usuario.



A continuación se presenta el Diagrama de Interacción concerniente a la verificación del *LogIn*:





Al autenticarse el ingresante como Administrador, se le mostrará el Menú correspondiente, en el que se despliega la lista general de Arquitectos, Diseñadores y Clientes registrados al Sistema. Siendo que solo el Administrador puede registrar Usuarios, este puede seleccionar la opción “*REGISTER User*”, luego de lo cual, independientemente de lo que tenga seleccionado de las listas, será enviado a la *Form* de *Registration.cs*. En cambio, si desea editar o eliminar Usuarios, deberá seleccionarlos previamente y modificarlos de a uno. En el caso de “*EDIT User*”, el Administrador será enviado a la *Form* de *DataEditing.cs*, la cual variará su tamaño dependiendo si la edición es para un Arquitecto o Diseñador (con menos atributos), o si es para un Cliente. Finalmente, seleccionando “\$ *CONFIGURE Cost/Profit \$*”, el Administrador podrá fijar los costos y precios de los materiales, para lo que será redirigido a la *Form* de *MaterialConfiguration.cs*. Nótese que los costos y precios de los materiales serán reiniciados a los valores por defecto siempre que se reinicie el programa.

The screenshot shows a Windows application window titled "REGISTRY". Inside, there are three separate registration forms:

- DESIGNER:** Fields for User Name, Password, Name, and Surname. A green button at the bottom says "Register as DESIGNER".
- CLIENT:** Fields for User Name, Password, Name, Surname, Number ID (format i.e. 4.851.112-6), Telephone (format i.e. 098742547), and Address (format i.e. Canelones 1267). A blue button at the bottom says "Register as CLIENT".
- ARCHITECT:** Fields for User Name, Password, Name, and Surname. A grey button at the bottom says "Register as ARCHITECT".

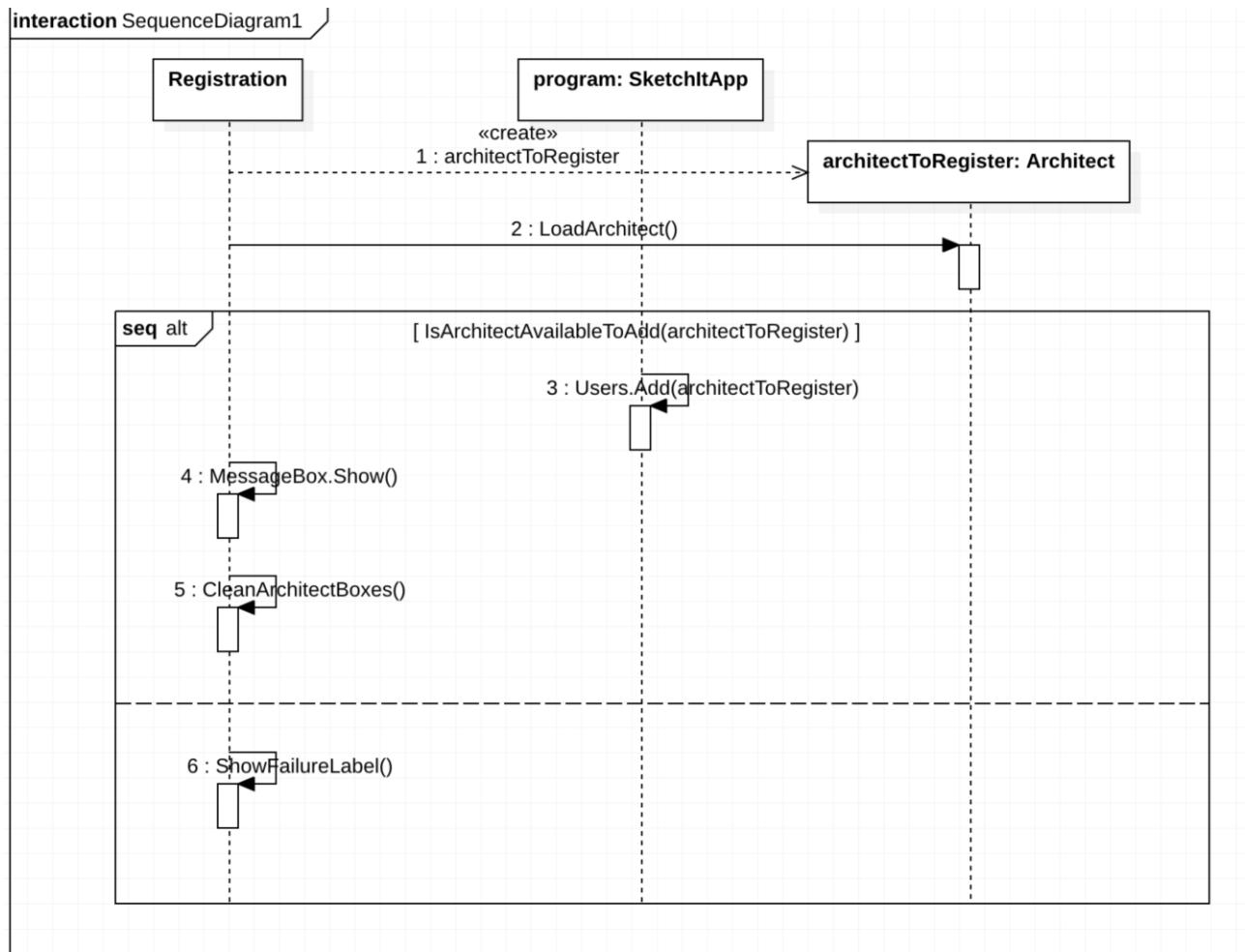
Al desear el Administrador agregar Arquitectos, Diseñadores y/o Clientes, será redirigido a la *Form* que se muestra en la Figura superior. En ella se pueden crear todos los Arquitectos, Diseñadores y Clientes que se quieran, sin necesidad de cerrar y volver a seleccionar la opción desde el Menú del Administrador. Los espacios a completar para registrar un Usuario poseen condiciones particulares para ser considerados como apropiados (además de no tener ningún campo vacío ni estar el Cliente, Arquitecto o Diseñador ya registrado):

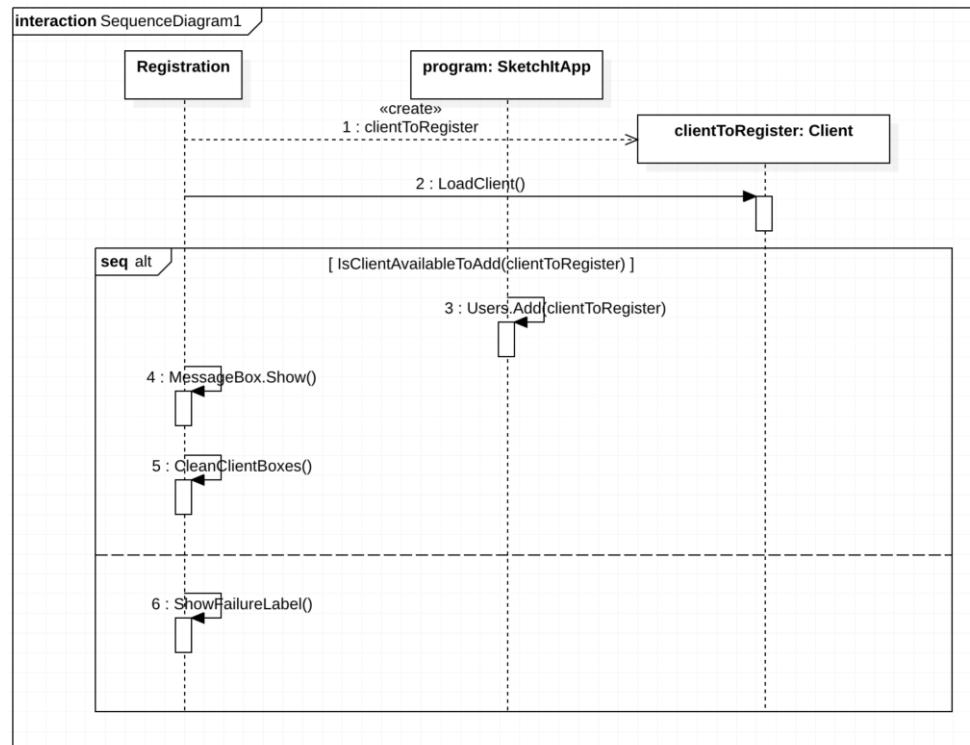
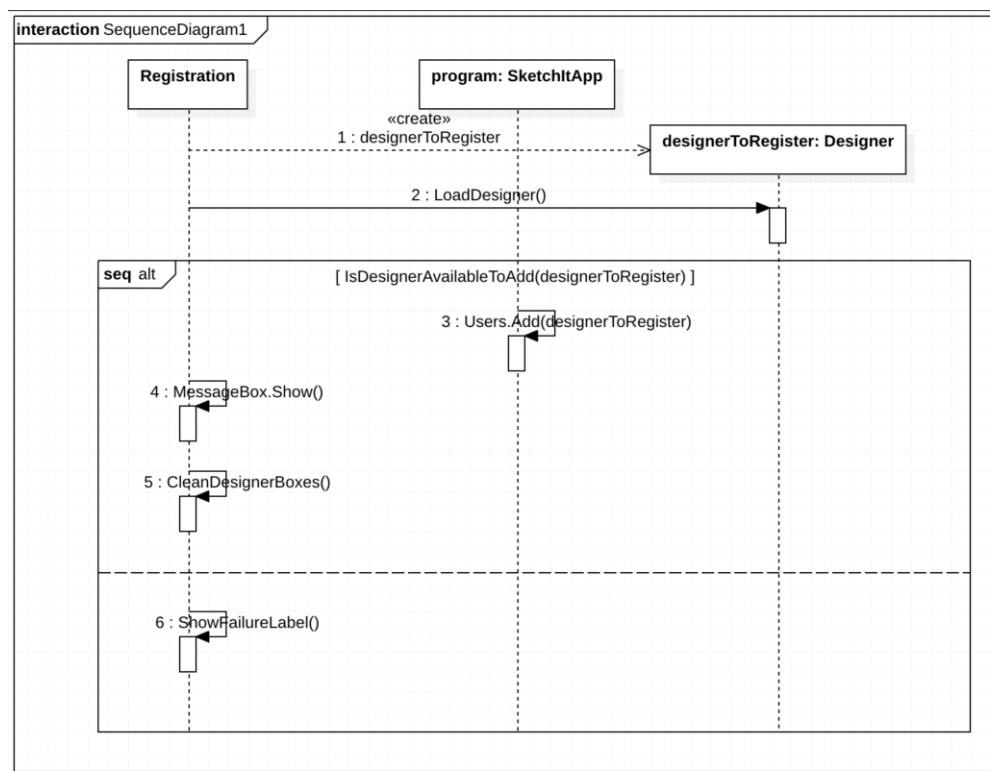
- ❖ **Nombre de Usuario (*Username*):** Sin espacios y con un largo entre 4 y 10 caracteres.
- ❖ **Contraseña (*Password*):** Con al menos un símbolo, un número y una mayúscula, y con un largo mayor a 6 caracteres.
- ❖ **Nombre (*Name*) y Apellido (*Surname*):** Sin espacios.
- ❖ **Cédula de Identidad (*Number ID*):** Con el formato tradicional uruguayo, ya sea con cifra millonaria o no, utilizando puntos y guiones. Todos números y símbolos.
- ❖ **Teléfono (*Telephone*):** Número telefónico móvil que comience con “09” y tenga largo 9 caracteres (todos números).
- ❖ **Dirección (*Address*):** Con al menos un espacio y un número.

Estas verificaciones son también realizadas al momento de cambiar el Administrador o el propio Usuario sus datos personales, de tal manera que no pueda ingresar datos considerados como

incorrectos. Se le notifica al Administrador tanto en caso de registro correcto como de registro incorrecto.

A continuación se muestran los Diagramas de Secuencia al momento de verificar un correcto registro del Arquitecto, Diseñador y Cliente (en los eventos de *Click* en cada Botón: *Register DESIGNER* y *Register CLIENT*):





EDIT DATA

User Name:	CaroRod3
Name:	Carolina
Surname:	Rodriguez
Number ID:	4.453.354-8
Telephone:	097364323
Address:	Malvin 2390

Al buscar editar los datos de los Usuarios, el programa redirige al ingresante a esta ventana (de nombre *DataEditing.cs*), la cual se adecúa si el mismo es autenticado como Arquitecto o Diseñador (*Form* superior izquierda), o como Cliente (*Form* superior derecha), o si el Administrador desea modificar a un Arquitecto, Diseñador o Cliente. Una vez desplegada la *Form*, son rellenados automáticamente los espacios con la información correspondiente del ingresante o del usuario buscado. El Usuario tratante allí podrá modificar valores, luego de lo cual si el ingreso es correcto se lo aplicará al seleccionar “*CHANGE*”. Acto seguido, se le redirigirá al Usuario a la página anterior a la que se encontraba antes de cambiar sus datos.

The screenshot shows a mobile application interface for managing material costs and prices. The title is '\$\$\$\$ MATERIAL \$\$\$', with 'COST' and 'PRICE' headers. It lists five categories: Wall, Beam, Door, Window, and Column. Each category has two input fields for 'Cost' and 'Price'. Below each field is a 'Current' value. To the right of the fields for Door and Window are dropdown menus labeled 'Select DOOR' and 'Select WINDOW' respectively.

	<i>COST</i>	<i>PRICE</i>
Wall: (\$ per Meter)	50 Current: 50	100 Current: 100
Beam:	50 Current: 50	100 Current: 100
Door:	50 Current: 50	100 Current: 100
Window:	50 Current: 50	75 Current: 75
Column:	25 Current: 25	50 Current: 50

Buttons at the bottom include '<- BACK' and '\$ CONFIRM \$'.

El Administrador puede cambiar el costo y precio de los materiales para cada elemento del Plano modificando el contenido de las casillas. Incluso puede variar el costo y precio de cada tipo de Abertura individualmente (identificándolo con el Selector a la derecha), pero nótese que solo podrá variar los valores de un tipo de la misma abertura a la vez (es decir, no podrá variar el dinero asociado a dos tipos de Puerta simultáneamente). Por defecto en caso de no seleccionar ningún tipo de Abertura, se le modificará el dinero correspondiente al tipo predeterminado de esa Abertura.

Si no ingresa ningún valor en alguna casilla o si el valor ingresado no es un numero entero o decimal, se le notifica al Administrador de su ingreso incorrecto. Cada valor ingresado es independiente entre sí, y no se especifica (por lo que no se verifica) que ocurriría en caso que el precio fijado sea menor que el costo (lo cual no sería rentable para el Trabajador y, al fin y al cabo, para el negocio de la aplicación). En caso de variar el Costo y Precio de un Material, automáticamente variarán los de todos los Planos de la aplicación.³

³ Discusión en Foro de Aulas: <https://aulas.ort.edu.uy/mod/forum/discuss.php?d=73409>

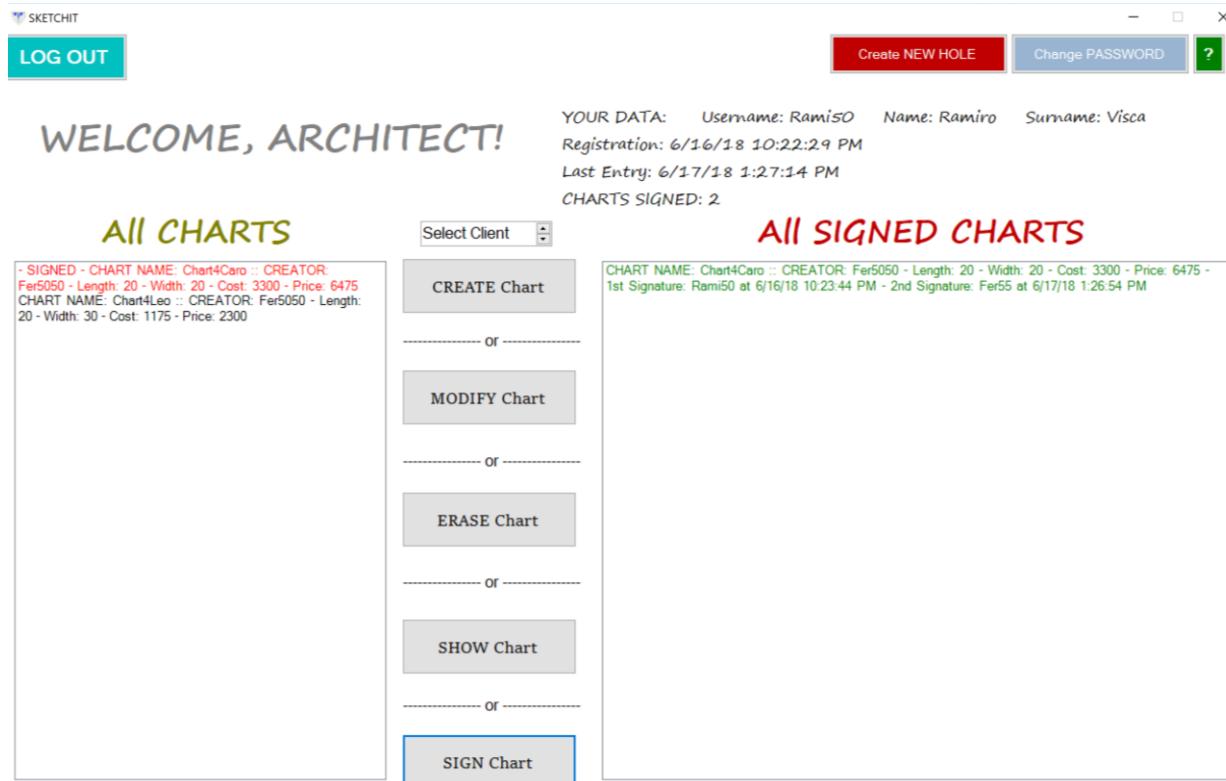


Al autentificarse correctamente el ingresante como un Diseñador, y en caso de ya haber confirmado sus datos personales, se le redirige a la *Form* de *MenuDesigner.cs* mostrada en la Figura superior. En ella se despliegan todos los Planos registrados en la aplicación (diferenciándolos por Firmados y No Firmados), de tal forma que todo Diseñador pueda ayudar a otro en la construcción o diseño de un Plano. Toda intervención que un Diseñador realice en un Plano será recabada por la aplicación y mostrada en su Menú, junto a otros datos personales registrados y ya aceptados.⁴

De la misma forma, se despliegan todos los Clientes existentes en la aplicación, correspondientes a todas las potenciales fuentes de capital para el Diseñador. Desde aquí el Diseñador puede cambiar su contraseña, crear un nuevo Plano (seleccionando primero a su Cliente objetivo), modificar un plano ya existente que no se encuentre firmado con un Primer Firmante (seleccionando un Plano de la lista desplegada), eliminar un plano ya existente tampoco firmado (también seleccionándolo) o visualizarlo y descargarlo en los formatos ya explicitados. Al buscar visualizarlo, se dirige al Diseñador a la *Form*

⁴ Discusión en Foro de Aulas: <https://aulas.ort.edu.uy/mod/forum/discuss.php?d=73635>

ChartShowing.cs, mientras que al elegir la opción de crear un nuevo Plano, se le redirige al Diseñador a la *Form ChartDrawing.cs* mostrada a continuación.



Al autenticarse correctamente el ingresante como un Arquitecto, y en caso de ya haber confirmado sus datos personales, se le redirige a la *Form* de *MenuArchitect.cs* mostrada en la Figura superior. Como se puede observar su formato es similar al del *MenuDesigner.cs* mostrado previamente, pero posee un mayor énfasis en la firma de los Planos. Así, está compuesto por 2 listas de Planos: una lista de Planos firmados (con los 2 Firmantes presentes) y una lista de Planos no totalmente firmados (sin Firmas o con un único Firmante). Ningún Plano puede poseer dos firmas con un único Firmante, pero un mismo Plano puede encontrarse firmado múltiples veces por diferentes Arquitectos. Como es solicitado, las demás funcionalidades del Arquitecto son las mismas que las del Diseñador, pudiendo filtrar las listas mostradas utilizando el Selector de Clientes, el cual también es necesario utilizar en caso que el Arquitecto desee crear un nuevo Plano.

The image displays two side-by-side screenshots of a software application window titled "CREATE YOUR HOLE!".

Left Screenshot (Door Creation):

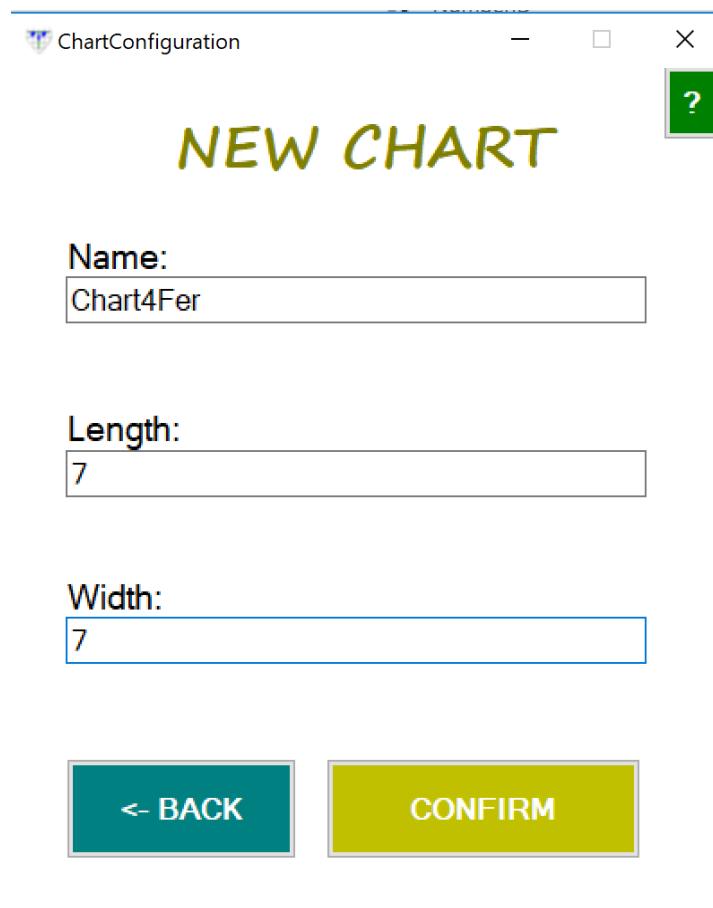
- Labels:** "Door:" (above the first row), "NAME" (bold green), "WIDTH" (bold green), "LENGTH" (bold green), "HEIGHT" (bold green).
- Inputs:** Four input fields for Name, Width, Length, and Height.
- Visual Preview:** Below the inputs are small blue icons representing a quarter-circle (Door) and a vertical rectangle (Window).
- Buttons:** Two large buttons at the bottom: "NEW DOOR" (blue) and "NEW WINDOW" (red).

Right Screenshot (Window Creation):

- Labels:** "Window:" (above the second row), "NAME" (bold green), "WIDTH" (bold green), "LENGTH" (bold green), "HEIGHT" (bold green).
- Inputs:** Four input fields for Name, Width, Length, and Height. In this screenshot, "Name" is set to "GlassDoor", "Width" is 2, "Length" is 1, and "Height" is 1.5.
- Visual Preview:** Below the inputs are larger blue icons representing a semi-circle (Door) and a tall vertical rectangle (Window).
- Buttons:** Two large buttons at the bottom: "NEW DOOR" (blue) and "NEW WINDOW" (red).

Todo Trabajador posee la capacidad de crear un nuevo tipo de Abertura (variando las dimensiones de nuevos tipos de Puertas y Ventanas) para su luego uso en el Plano. Para ello, se dirige al Trabajador a la ventana de *HoleCreator.cs*, presentada en las Figuras superiores. Estas nuevas dimensiones se encuentran limitadas porque el Ancho no puede superar los 3 metros, el Largo no puede superar los 2 metros y el Largo junto a la Altura desde el Piso no pueden superar los 3 metros (el Alto de la Pared). Cada tipo de Abertura es también persistente por la Base de Datos y utilizable entre Planos de distinto Trabajador (independientemente de quien haya definido la Abertura). Las dimensiones de las Aberturas pueden ser enteros o decimales, pero nunca letras o palabras.

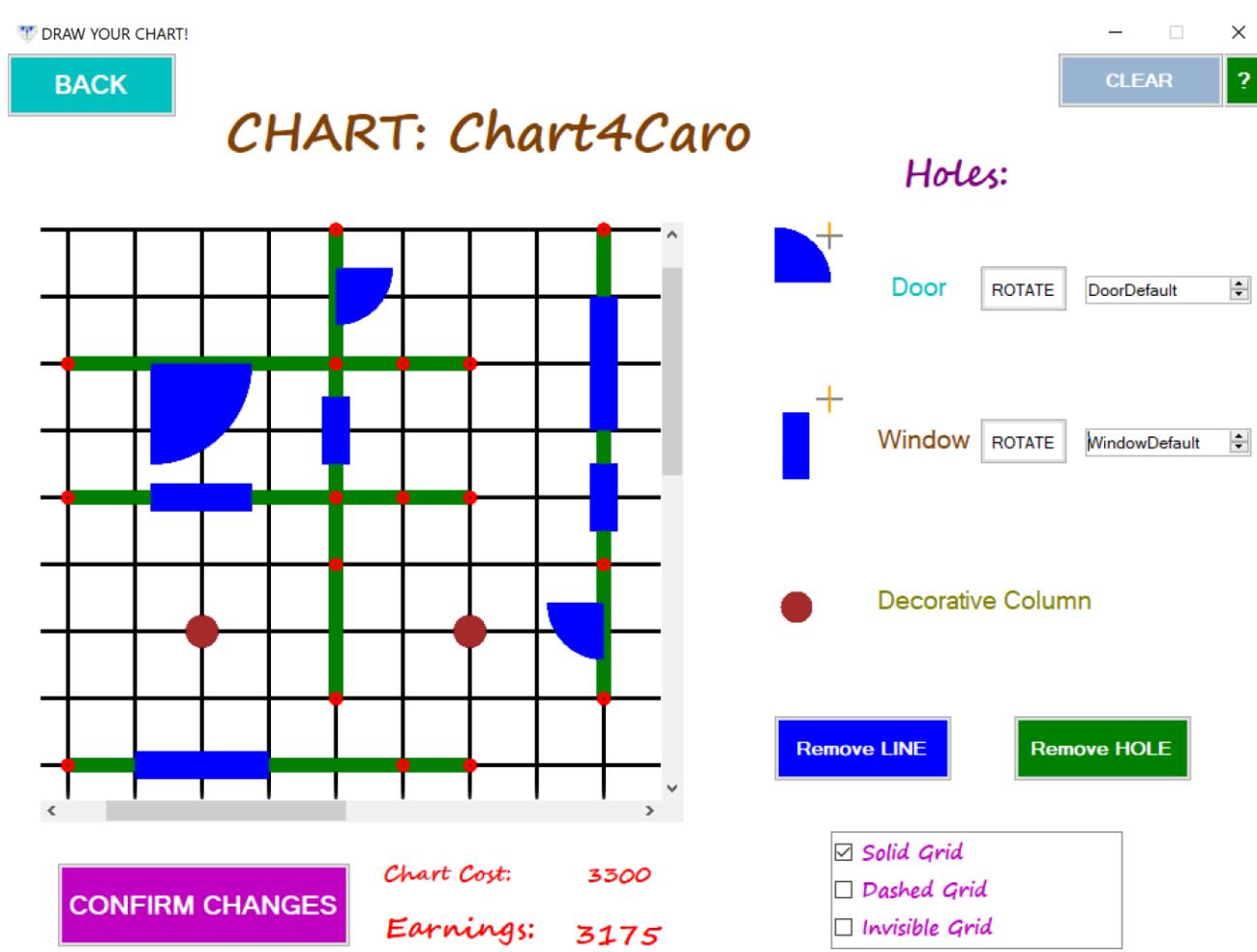
Al momento de utilizarla en el Plano, cada tipo de Abertura es identificada por su Nombre, por lo que es sumamente importante que el Trabajador recuerde las dimensiones de sus Aberturas en relación al mismo. Como se puede observar en las imágenes superiores, se agrega una ayuda visual al Trabajador para que al momento de variar las dimensiones se pueda conocer en tiempo real el nuevo estado configurado.



Al buscar crear un nuevo Plano, el Trabajador creador debe determinar primeramente el Nombre del Plano, su Largo (Alto) y Ancho, estas dos últimas medidas en metros. Tanto el Largo como el Ancho no pueden superar los 100 metros por decisión de diseño, si bien el programa lo soporta (al haber utilizado un *Scroll*, se permitirían dimensiones tan grandes como las que éste admite). En la práctica no se espera que se utilice este programa para distancias más grandes. Por otro lado, se espera que el Nombre del Plano no contenga espacios, aunque se puede repetir entre otros Planos del programa o incluso del mismo Diseñador.⁵

Una vez definido el Plano, el Trabajador podrá ser capaz de trabajar en él, puesto que al seleccionar “CONFIRM” será dirigido a la *Form ChartDrawing.cs* representada posteriormente.

⁵ Discusión en Foro de Aulas: <https://aulas.ort.edu.uy/mod/forum/discuss.php?d=73774>



Una vez definido el Plano que el Trabajador busca crear o modificar, podrá dibujar Paredes al cliquear, mantener apretado (*Hold*) y soltar el botón izquierdo del Mouse (o haciendo *click* con un dedo en un *Trackpad*) para crear una Pared, siempre y cuando no esté otra pared en el mismo lugar ni interseccione ninguna Abertura. Automáticamente le son colocadas las vigas en los extremos, y se verifica que el tamaño de la Pared no sea excesivo, puesto que si es mayor de 5 metros se lo comienza a subdividir en trozos más pequeños y se colocan más columnas en cada subdivisión. Después se debe corroborar que existan intersecciones entre Paredes, luego de lo cual en caso afirmativo se deben realizar nuevamente subdivisiones.

Luego, en caso de buscar agregar Puertas o Ventanas, se debe primer seleccionar una orientación determinada a la correspondiente Abertura, luego de lo cual seleccionándola haciendo doble *Click* con el Mouse (o *Trackpad*) en el símbolo correspondiente y luego cliqueando un punto del Plano se añadirá la Abertura. Sin embargo, como se mencionó en las Descripciones previas, se debe considerar que las

Aberturas solo tienen sentido si van arriba de Paredes ya existentes (a excepción de las Columnas), y cumpliendo distintas condiciones (además de no existir ya una abertura en las coordenadas en las que se la busca agregar). En el caso de la Ventana, se debe verificar que la orientación de la misma se encuentre alineada (paralela) con la Pared a la que se le intenta agregar (ambas verticales o ambas horizontales) y que no exista ninguna Ventana en la mencionada Pared, en un bloque más y en un bloque menos. En el caso de la Puerta, se debe verificar que la Puerta sea perpendicular a la Pared y que no exista ninguna Puerta en un bloque a la redonda (sin contar las esquinas) a sí misma. Las Columnas Decorativas pueden colocarse en todo lugar en que no se encuentre una Pared.

El Trabajador además puede eliminar líneas (Paredes) enteras cliqueando y manteniendo desde una columna a otra, luego de lo cual se eliminan todas las Aberturas y vigas que solo de esa Pared dependiesen. Igualmente, el Diseñador puede eliminar Aberturas, seleccionando individualmente cada coordenada a la cual se le desea eliminar la Abertura. Ambas eliminaciones son excluyentes, es decir, no se puede seleccionar "*Remove LINE*" y "*Remove HOLE*" simultáneamente.

Por decisión del grupo, en caso de seleccionar la opción de remover Abertura y tener seleccionado alguno de las opciones de agregar Abertura, el Diseñador notará que no podrá agregar Aberturas hasta que no se deseccione la opción de remover Abertura. Esto no curre en caso de seleccionar la opción de remover Línea y buscar agregar una Abertura. Aquí la Abertura podrá ser agregada pero no se podrán agregar nuevas Paredes hasta no deseleccionar la opción de remover Línea.

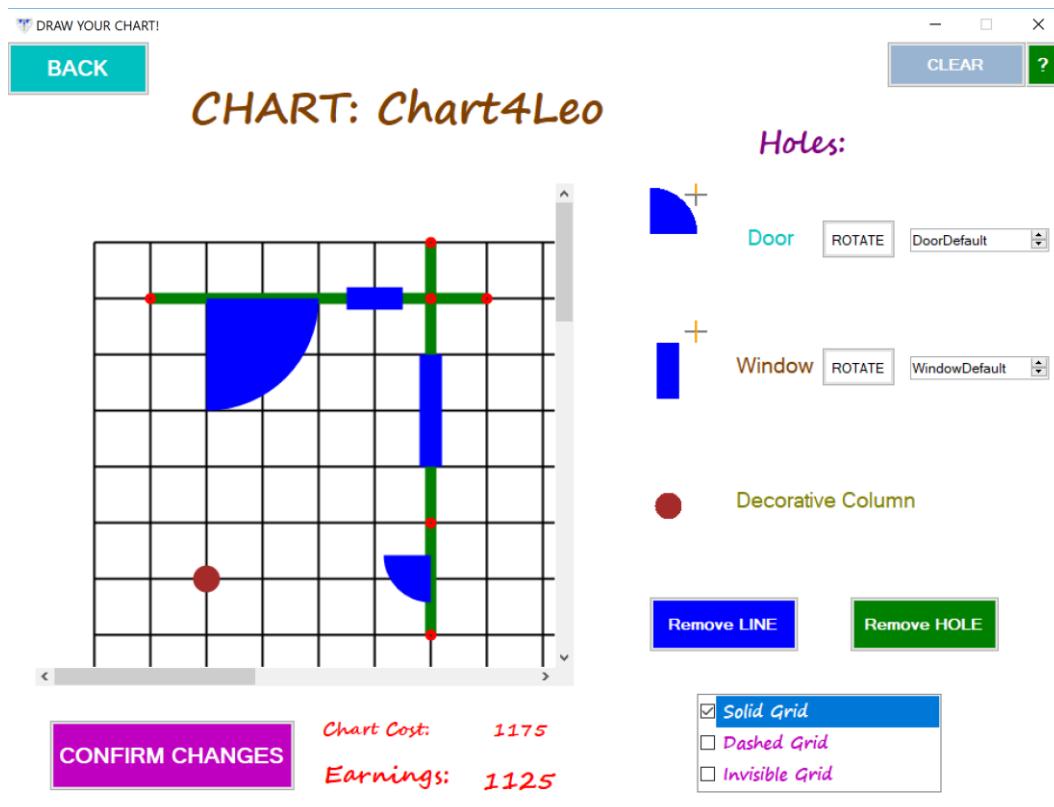
Como funcionalidad adicional se agrega que, en caso de buscar realizar un nuevo Plano desechando el anterior, el Diseñador puede seleccionar *CLEAR*, con lo cual se eliminarán todos los elementos colocados hasta el momento en el Plano.

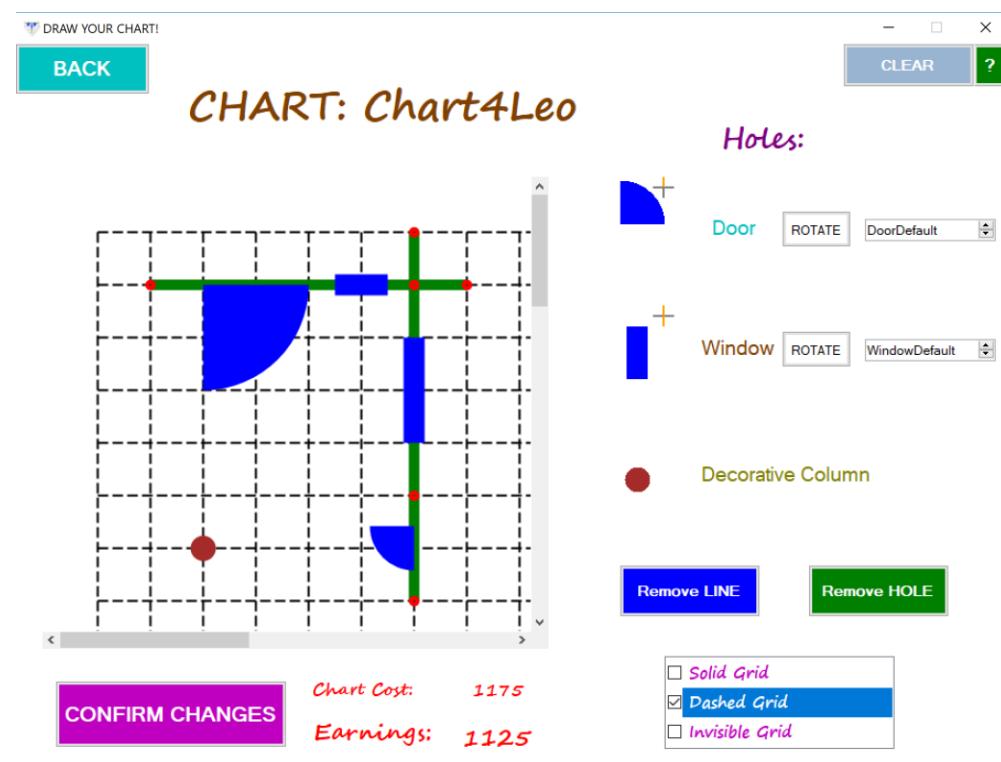
Nótese como al variar la Rotación de la Puerta y la Ventana en los símbolos pequeños (de la derecha de la ventana), existe una variación del indicador de orientación adjunto, conformado por una cruz con colores predeterminados grises, que va cambiando de color a naranja coherentemente con el cambio de orientación del dibujo. A su vez, el Trabajador puede elegir el tipo de Abertura deseada identificándolo por su nombre en el Selector derecho, debiendo recordar previamente las características del tipo de Abertura elegido.

Finalmente, se especifica que el Costo (y más importantemente para el Trabajador, la Ganancia, definida como la diferencia entre el Precio pagado por el Cliente y el Costo de la Construcción) son calculados dinámicamente al agregar o eliminar elementos al Plano. Este Costo y Ganancia se encuentran sujetos a los valores determinados por el Administrador en *MaterialConfiguration.cs*.

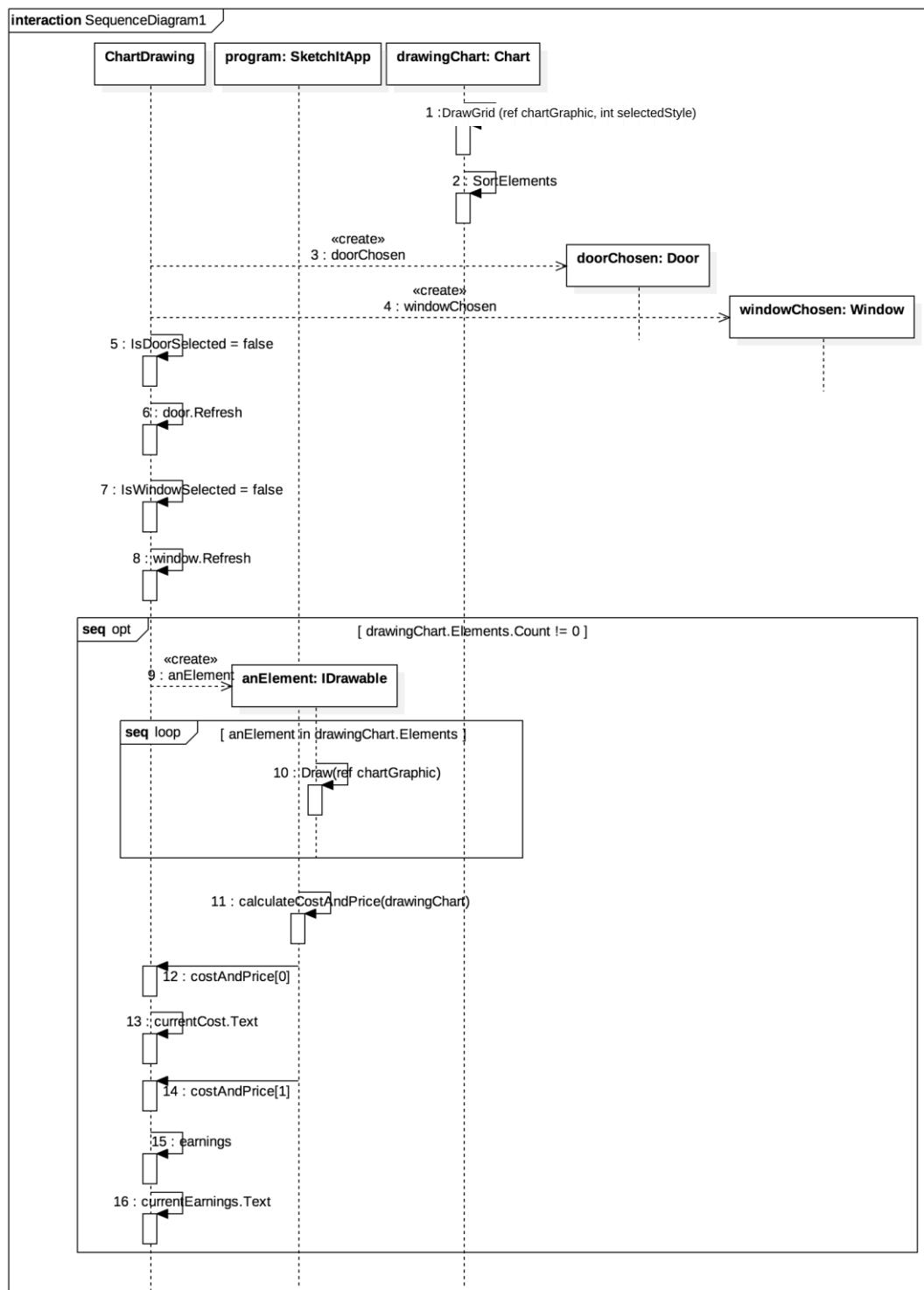
Para una visualización más personalizada del Plano, se agregan las opciones de dibujar la Grilla con líneas continuas, discontinuas (punteadas) o sin líneas (solo el recuadro), pudiéndolo variar dinámicamente en cualquier momento según el Trabajador juzgue conveniente.

Una vez confirmados los cambios, el Plano será actualizado y registrado, posibilitándose su visualización y posible futura descarga. En caso de generar cambios y luego cerrar el programa o seleccionar la opción de Atrás (*BACK*), los cambios no serán guardados ni se tendrá en cuenta para el perfil del Trabajador la ayuda que haya realizado a la construcción o registración del Plano.

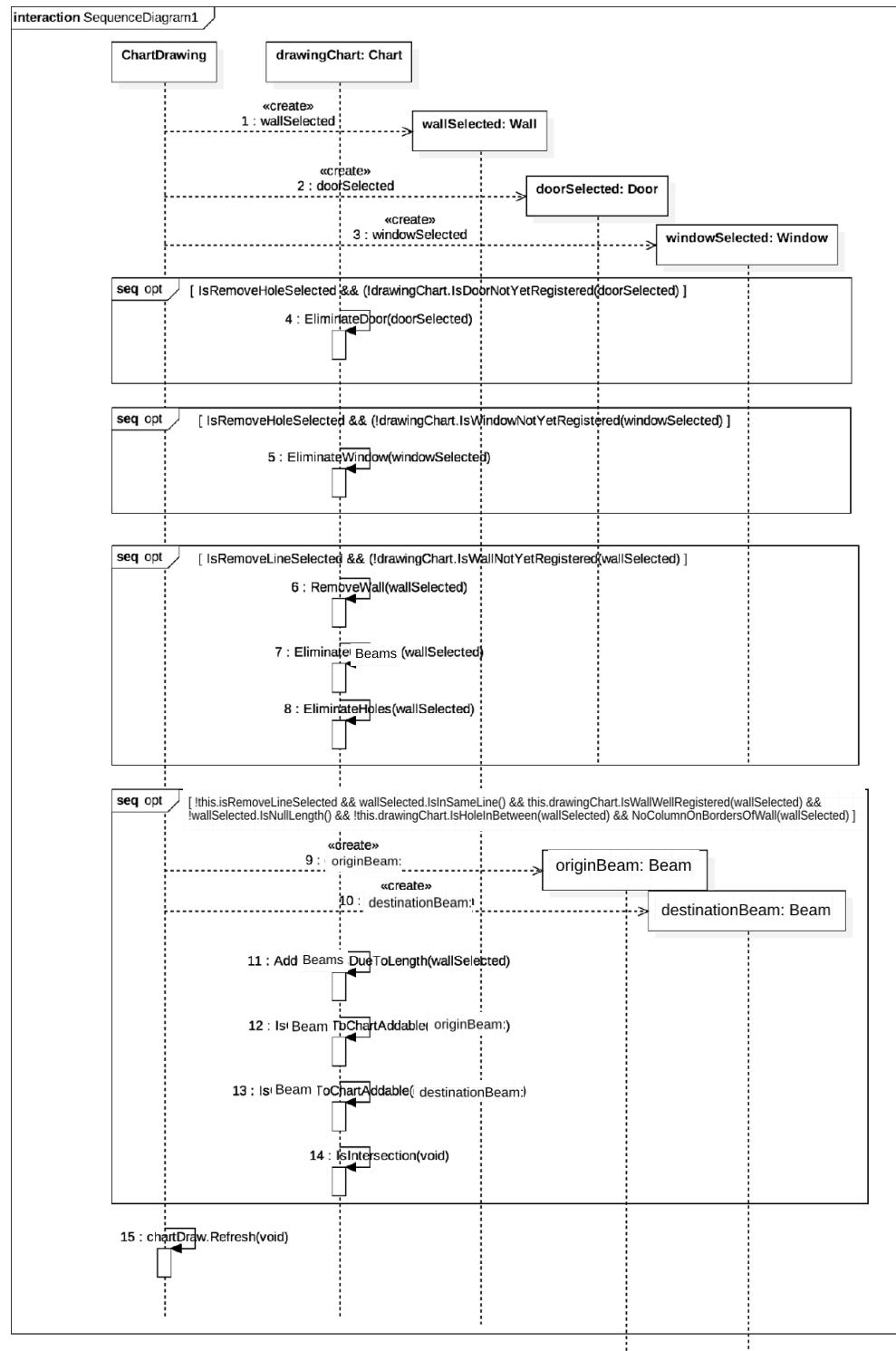




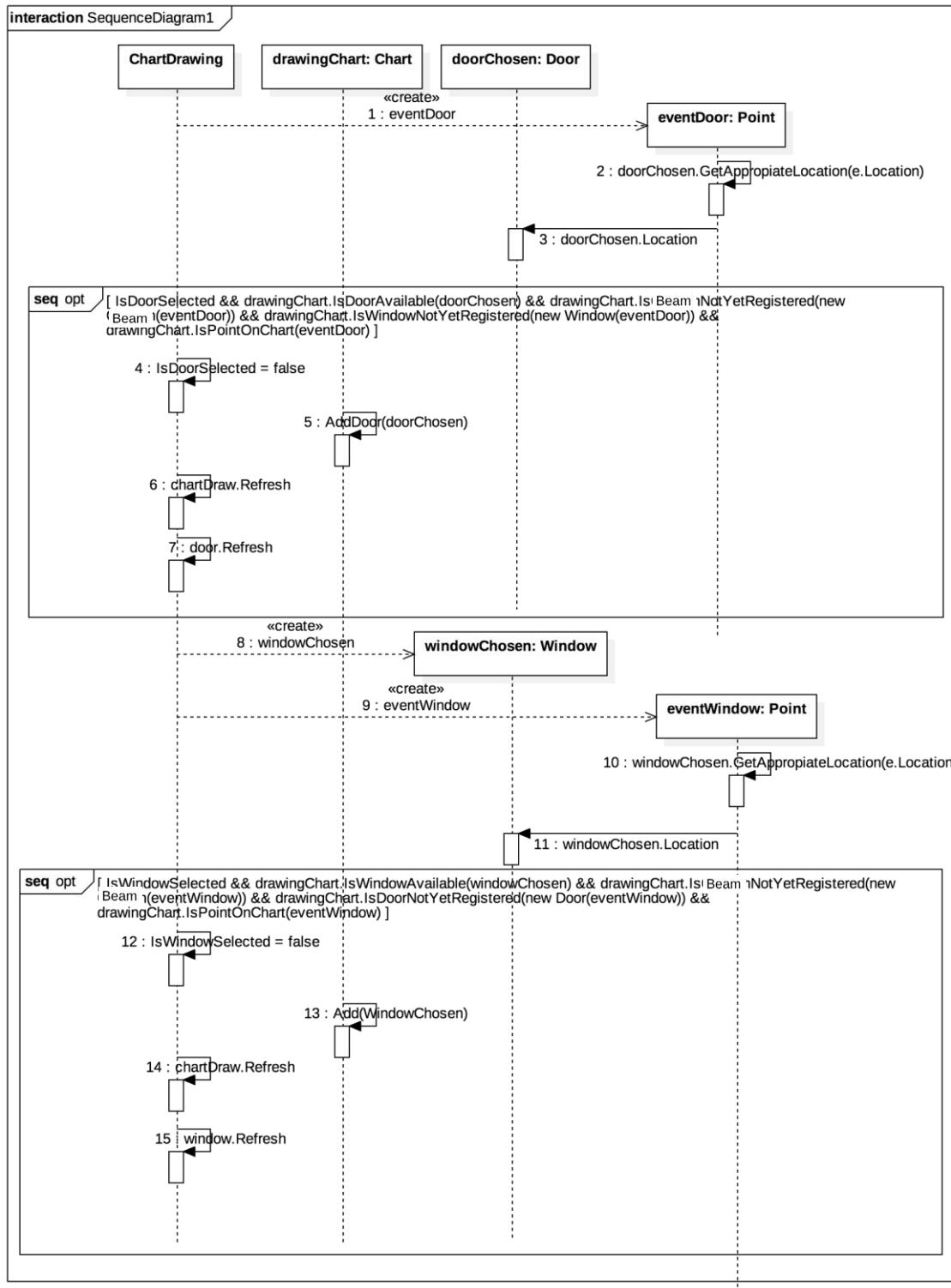
Siguiendo se muestra un Diagrama de Secuencia representando la interacción necesaria al dibujar el Plano (y realizado cada vez que se refresca el Panel), existente en la función *ChartDraw_OnPaint()*:

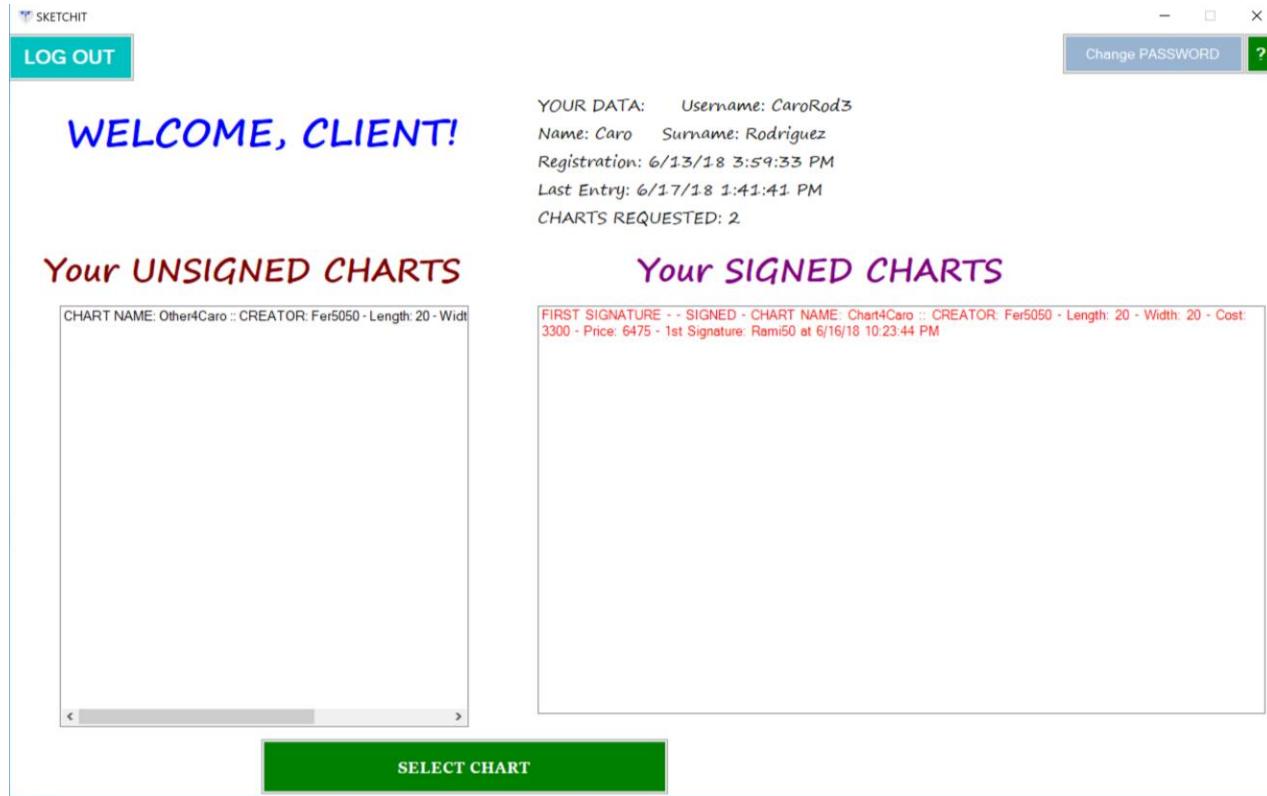


A continuación se presenta un Diagrama de Secuencia referido a la función al momento de hacer Click en el Panel de Trabajo (de nombre *chartDraw*):

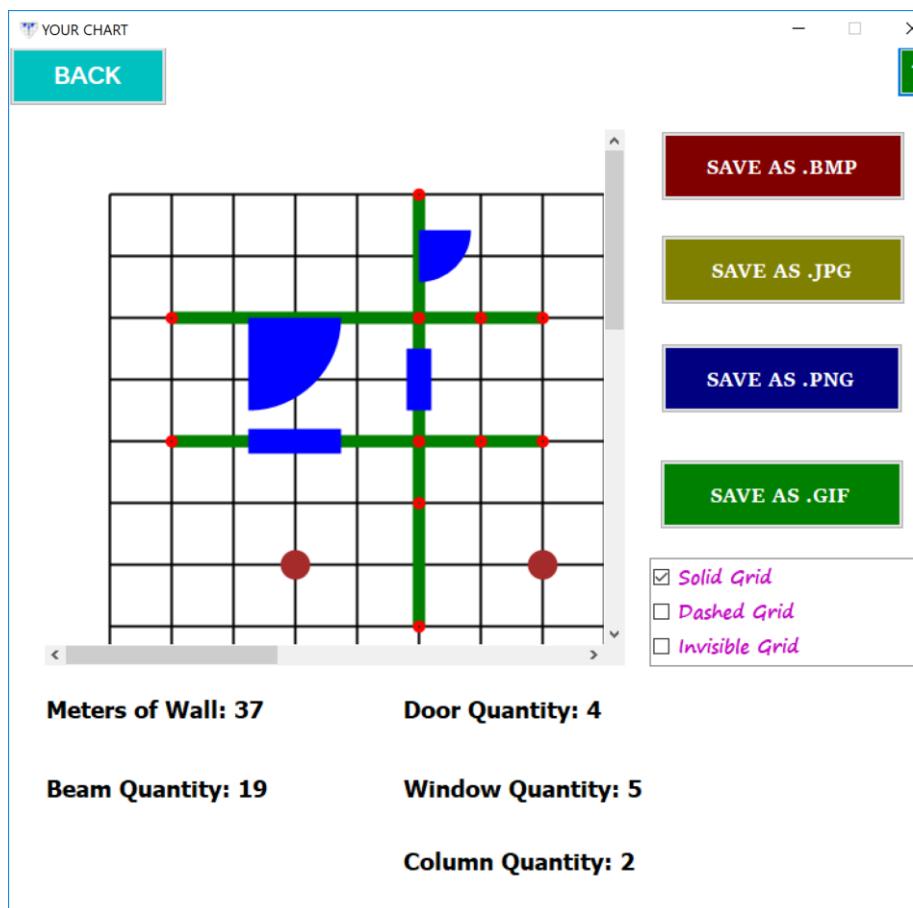


Ahora se presenta un Diagrama de Secuencia referido a la función al momento de agregar Puertas y Ventanas en el Panel de Trabajo (de nombre *chartDraw*):





Al autenticarse correctamente el ingresante como un Cliente, y ya haber entrado anteriormente por primera vez, se redirige al Usuario al *MenuClient.cs*, donde se despliega el Menú del Cliente. En esta *Form* el Cliente puede visualizar su información personal (incluyendo la cantidad de Planos solicitados), cambiar su contraseña, visualizar los planos pedidos y descargarlos en caso que así lo deseé. La visualización de la Lista de los Planos del Cliente es clasificada según sean aquellos firmados (con un Firmante, es decir, a la espera de un Segundo Firmante) y aquellos no Firmados. Para poder visualizar y descargar sus planos (de cualquiera de las dos listas desplegadas), el Cliente debe primero seleccionar de una de las Listas de Planos aquel que deseé, y luego seleccionar la opción “*SELECT CHART*”. Por diseño de la interfaz, es imposible que el Cliente seleccione de ambas listas de Planos simultáneamente. Una vez realizado tal procedimiento, se redirige al Cliente a la *Form* *ChartShowing.cs* presentada a continuación.



Al ingresar el Diseñador o el Cliente a visualizar un plano, se le dirige a una *Form* análoga a la Figura superior. En ella se puede visualizar el Plano seleccionado, pero no se le puede realizar modificaciones. También, se puede descargar el dibujo del Plano en formatos .bmp (*Bitmap Image File*), .jpg (*Joint Photographic Experts Group*), .png (*Portable Network Graphics*) y .gif (*Graphics Interchange Format*). Si el archivo de imagen del Plano ya se encuentra descargado, se sobreescibirá con la nueva descarga realizada. La visualización del formato de Grilla del Plano puede ser variado dinámicamente tal como ocurre al momento de dibujar el Plano, pudiendo incluso descargar el archivo con el formato de Grilla deseado. En todo momento se observa el contador de Elementos del Plano debajo de la Figura.

El formato de archivo BMP es capaz de almacenar imágenes digitales bidimensionales, tanto monocromas como en color, en distintas profundidades de color. En cambio, el formato de archivo JPG o JPEG es capaz de almacenar imágenes digitales con pérdidas pero a una alta tasa de compresión. Se brinda especial interés en este formato de imagen en la materia específica de Telecomunicaciones conocida como “Audio y Video”, puesto que es este formato el que se utilizó como base para la codificación de videos MPEG (*Motion Pictures Experts Group*) y MJPEG (*Motion JPEG*). Luego, el formato

de archivo PNG soporta compresión de imágenes sin pérdidas, y el formato de archivo GIF se ha generalizado en la *World Wide Web* (WWW) debido a su amplio soporte y portabilidad.

Finalmente, todo Arquitecto, Diseñador y Cliente en cualquier momento desde su Menú correspondiente cambiar su contraseña, para lo cual se le solicita previamente su contraseña actual. Solo podrá cambiar su contraseña si su contraseña actual fue ingresada correctamente, y si su nueva contraseña cumple con las condiciones planteadas anteriormente. En caso contrario, se notificará al Usuario en tiempo real, debido a que la etiqueta a la derecha de cada casilla irá cambiando a medida que se cumplan con los requisitos necesitados. No se le permitirá editar la contraseña al Usuario hasta que las tres etiquetas de la derecha de las tres casillas se encuentren en “OK”.

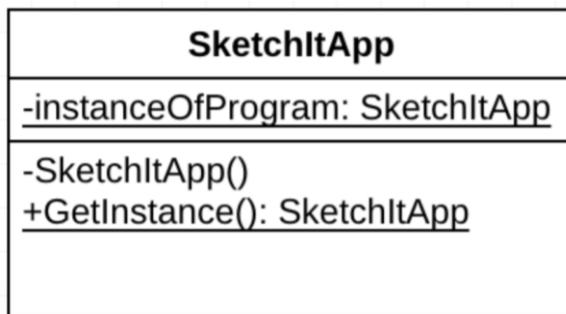
Descripción y Justificación de Diseño:

En la actual unidad se proseguirá a fundamentar el diseño implementado en la aplicación, considerando que lo realizado fue lo que en el momento el equipo creyó mejor. En el transcurso de la creación del programa se debió aplicar *Refactoring* en numerosas ocasiones, principalmente a efectos de adaptarse a las especificaciones que se fueron detallando en el Foro de Aulas sobre el Obligatorio.

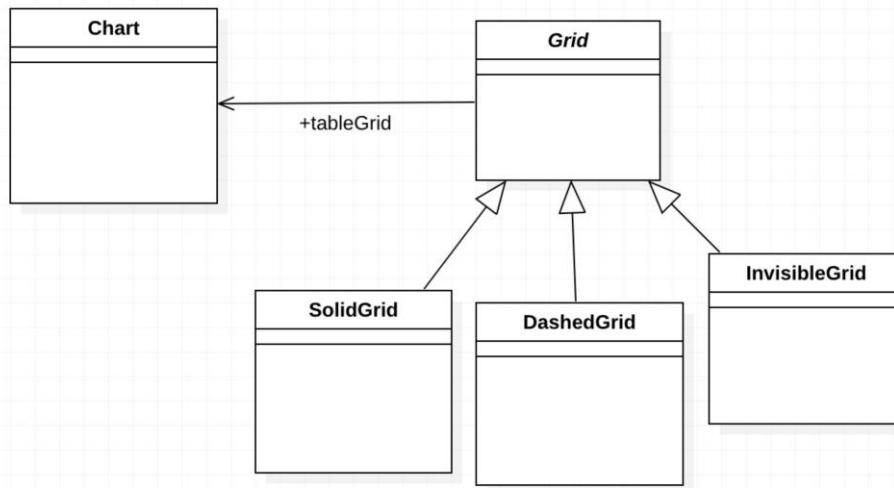
A continuación se presentan las principales decisiones de Diseño que se consideran relevantes en justificar, presentadas en un formato creciente de especificidad y modificadas de tal forma de adecuarse los contenidos de ambos Obligatorios relacionados:

- En todo momento se prefirió definir y utilizar constantes para cada clase, de tal manera de que quede el código más leíble y prolífico. Estas constantes fueron a su vez coherentemente definidas para cada clase, de tal forma de encontrarse todos estos valores que implícitamente intervienen en la clase de forma conjunta a ella.
- La persistencia de los datos del programa fue implementada haciendo uso de la herramienta *Entity Framework* para la gestión de Bases de Datos. Sin embargo, es primordial notar que no se encontró manera de hacer persistir interfaces haciendo uso de esta herramienta, por lo que una vez confirmado por los docentes se procedió a realizar el cambio de las inicialmente interfaces *IUsable*, *IChartable* e *IDrawable* en Clases Abstractas, de las cuales sus clases implementadas pasarían a heredar. Esta fue una gran limitante que tuvimos que afrontar inevitablemente debido a la obligatoriedad del uso de *Entity Framework*. De todas formas se decidió permanecer inalterado el nombre de la entidad de tal manera de ser identifiable en un futuro por los docentes correctores.
- La persistencia de los datos en Base de Datos fue realizada de tal forma de evitar su implementación a modo de Serialización, esto es, cargando toda la información pasada del sistema al inicio de la Aplicación, y guardándola al finalizar. Esta metodología, sin bien útil y eficiente, es poco satisfactoria en caso que se busquen guardar todos los estados intermedios del Programa, y evitar pérdidas de información intermedias. Por ello, se prosigue a guardar datos en la Base de Datos alternadamente mientras que se está ejecutando el programa, a medida que se van actualizando las Listas y Colecciones presentes en la Aplicación.

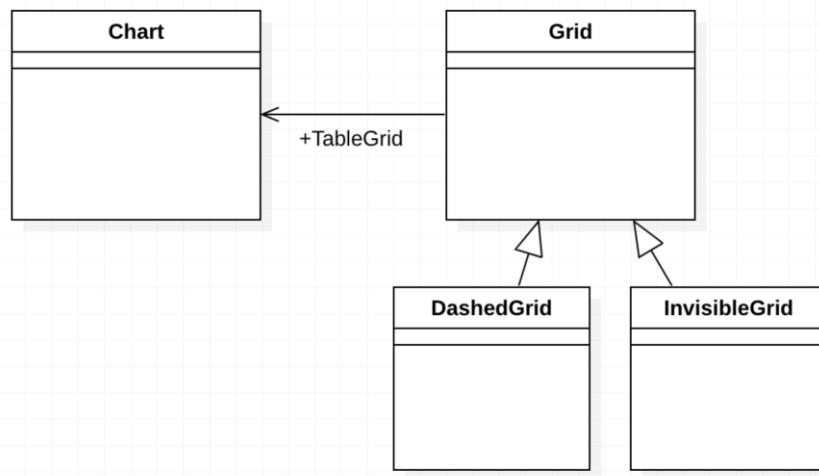
- Igualmente, como fue previamente mencionado, la Base de Datos fue buscada de implementar de tal forma que fuera ella la que dependiera de la Lógica del Negocio y no a la inversa, lo que concuerda con el Principio de Diseño tratado conocido como **Principio de Inversión de Dependencia (DIP: Dependency Inversion Principle)**, al estar las herramientas de Bajo Nivel dependiendo de las de Alto Nivel y no lo opuesto.
- Los archivos de imagen se ubican en la dirección determinada por defecto como de inicio del programa, la cual por simplificación no se modificó (pudiéndose editar en *Properties* de la Solución), dentro de una carpeta denominada *SketchItAppData/ChartImages*, de tal manera de focalizar la información en un directorio y no desperdigarla por sitios múltiples y de poco control. Se prefirió utilizar la dirección de origen como la por defecto del programa, puesto que esta puede ser diferente para cada Usuario en cada Ordenador, y de esta forma también se asegura que esa dirección tenga las licencias de lectura y escritura en memoria necesitadas.

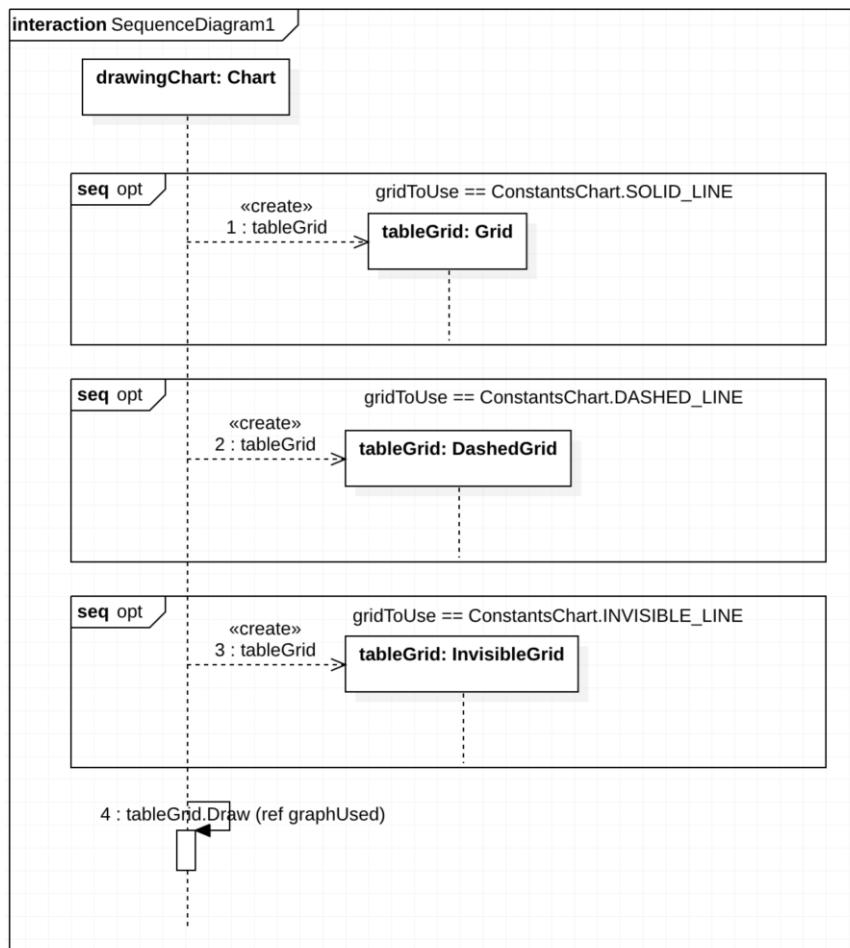


- Al momento de cargar la instancia inicial del Sistema, se prosiguió a utilizar el **Patrón de Diseño Creacional visto en clase conocido como Singleton**, de tal manera de asegurar que solo exista una instancia de la clase Sistema, y brindar un punto de acceso global a la misma. Así, tal como se menciona en la Bibliografía recomendada “*Design Patterns: Elements Of Reusable Object-Oriented Software*”, se utiliza un menor espacio de variables, facilita que la clase Sistema sea en un futuro “*subclaseada*” y permite un número variable de instancias de la clase mencionada.



- Para generar que la Grilla dibujada al momento de crear el Plano sea personalizable, se prosiguió a utilizar el **Patrón de Diseño de Comportamiento conocido como *Strategy***, definiendo una familia de algoritmos, encapsulando cada uno y haciéndolos intercambiables. Esto, guiados por la Bibliografía anterior, genera que se facilite factorizar funcionalidades comunes mediante Herencia al igual que brinda una alternativa a la necesidad de *subclassing* y elimina la necesidad de utilizar condicionales. Sin embargo, a efectos de resultar persistible en *Entity Framework* y existiendo un problema con la numeración de las Id en caso de utilizar un formato con la Clase *Grid* Abstracta como el de la Figura superior, se prosiguió a utilizar una versión modificada del Patrón. En esta modificación, como se indica en la Figura inferior, no se utilizaron Clases Abstractas y, en cambio, se incluyó en *Grid* el comportamiento predeterminado (el mostrar las líneas continuas) que originalmente se encontraba definido en *SolidGrid*.

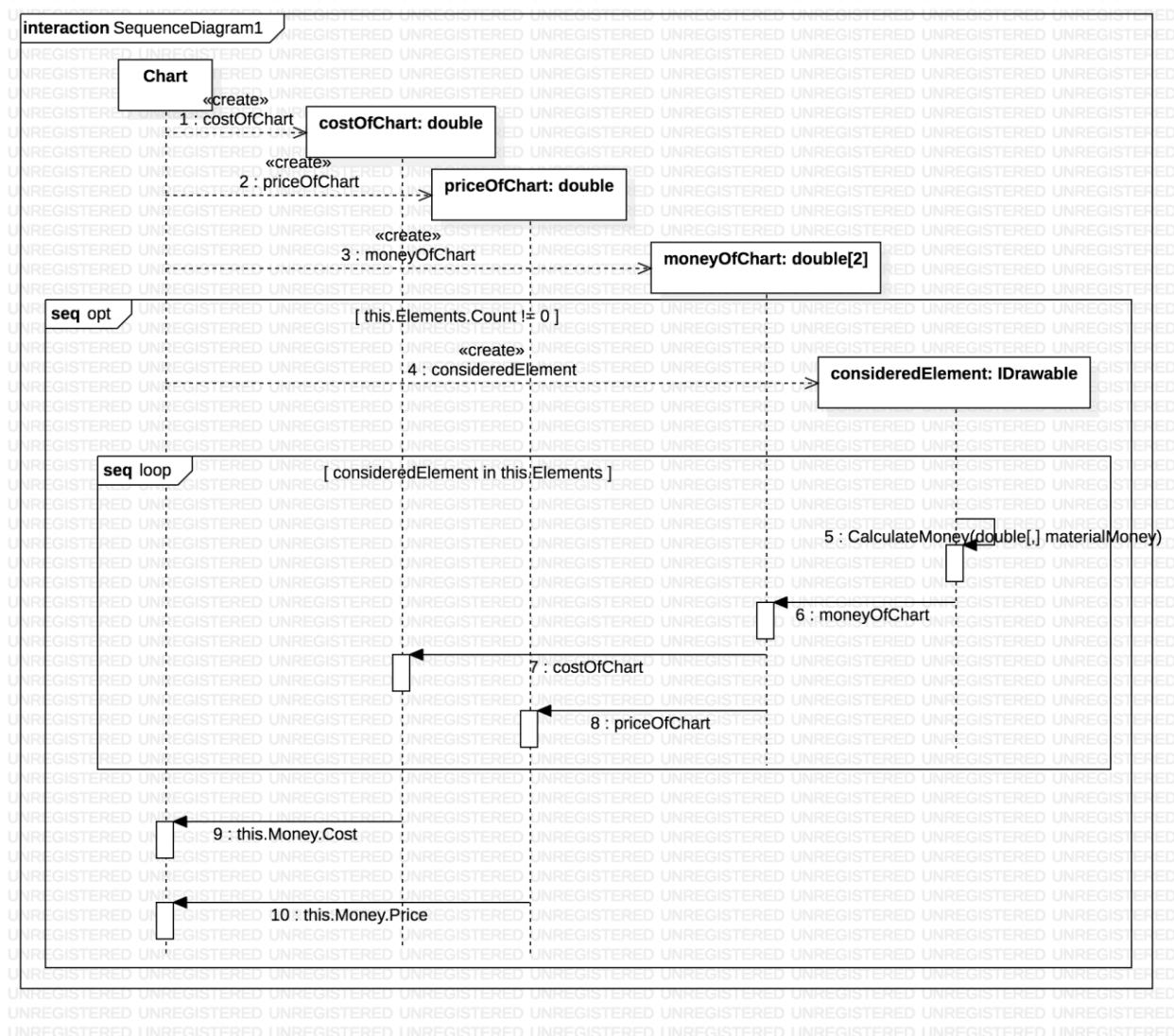




- Se buscó en todo momento que el Sistema cumpliese con el **Principio de Diseño de Abierto Cerrado (OCP: Open Close Principle)**, de tal forma que todos sus módulos deban estar abiertos a la extensión y cerrados a la modificación. Para ello, previendo posibles futuros agregados o cambios a realizar al Sistema, se planteó agregar la mayor cantidad de puntos de articulación a la estructura del Programa, consistentes en interfaces o relaciones entre clases bases y derivadas de tal forma que por llamadas polimórficas se puedan utilizar dinámicamente y sea el Sistema menos tendiente a las modificaciones y reestructuraciones. Este fue el motivo por el que se agregaron las interfaces de *IUsable* e *IChartable* a modo de ejemplo (ya que se pudiesen haber agregado otras muchas más, como por ejemplo entre *User* y *Worker* o *User* y *Client* o *Chart* y *Signature*). Sin embargo, nótese que no se consideró conveniente el agregado de más articulaciones ya que se previó aumentaría en sobremanera la complejidad del código respecto al beneficio de una posible futura mayor escalabilidad, a la vez que se debería modificar todas las interfaces que se agreguen a Clases Abstractas para poder ser persistidas

por *Entity Framework*. Incluso la utilización de una Clase Abstracta entre *SketchItApp* y *Hole* que posibilite la inclusión de otras variaciones de Aberturas o demás elementos no es permitida por *Visual Studios* ya que C# no permite Herencias Múltiples, y la Clase *Hole* ya se encontraba heredando de la ahora Clase Abstracta *IDrawable*.

A continuación se despliega el Diagrama de Secuencia del procedimiento para calcular polimórficamente el Costo y Precio de los Elementos del Plano, correspondiente al método “*CalculateMoney (double[,] materialMoney)*” de la Clase *Chart*, de modo introductorio al contenido presentado en el siguiente punto.



• Los datos tratados donde hubiese dinero involucrado (tales como el Costo y Precio final de un Plano, el Costo y Precio predeterminado de un Elemento del Plano) se prosiguió a modificarlos respecto al Obligatorio 1. Esto fue así ya que, al observar que siempre el Costo y el Precio se encontraban relacionados y en variables separadas, se consideró más prolífico y un mejor uso de los recursos del ordenador si se utilizara un *Array* (a falta de una estructura particular a crear) de nombre “*Money*” que estuviese cuantizado en 2 números decimales por fila: uno para el Costo y otro para el Precio de la entidad referida. Esta fue la razón por la que, a falta de una estructura para matrices en C#, en la Clase *SketchItApp* se prosiguió a hacer uso de “*Multidimensional Arrays*” (notación “[,]”), puesto que fue la única forma visualizada para utilizar matrices en C#. Sin embargo, estas modificaciones tuvieron que cambiarse nuevamente porque los *Arrays* tampoco son persistibles en *Entity Framework*. Por tanto, se crearon clases contenedores (*IntContainer*, *DoubleContainer*, *MoneyContainer*) que contengan los datos que anteriormente contenían los *Arrays*. El caso de *MoneyContainer* pudo haberse programado en base a 5 instancias de *DoubleContainer*, pero se consideró más claro visualmente y con menos probabilidad de error el desarrollar los Contenedores independientemente entre sí, con posibilidad futura de relacionamiento.⁶

• De la misma forma que lo mencionado en el punto anterior, se debió realizar las Clases Contenedoras *PointContainer* y *PointCoordinatesContainer* para los casos en que se usaban instancias de *Point* y (en caso de las coordenadas de las Paredes) *Arrays* de *Point*. Esto fue necesario puesto que ni los *Arrays* ni instancias de *Point* son persistibles en *Entity Framework*. Se reitera nuevamente que se pudo haber formado *PointCoordinatesContainer* con dos instancias de *PointContainer* (punto de origen y punto de destino de la pared), pero no se realizó para evitar equivocaciones en el *Refactoring* y facilitar la visualización del código. Nótese que se intentó facilitar la modificación del código al adecuar los nuevos formatos de atributos a los métodos ya creados y verificados, por lo que la Clase *Point* y los *Arrays* continúan siendo usados en los métodos, no así como atributos.

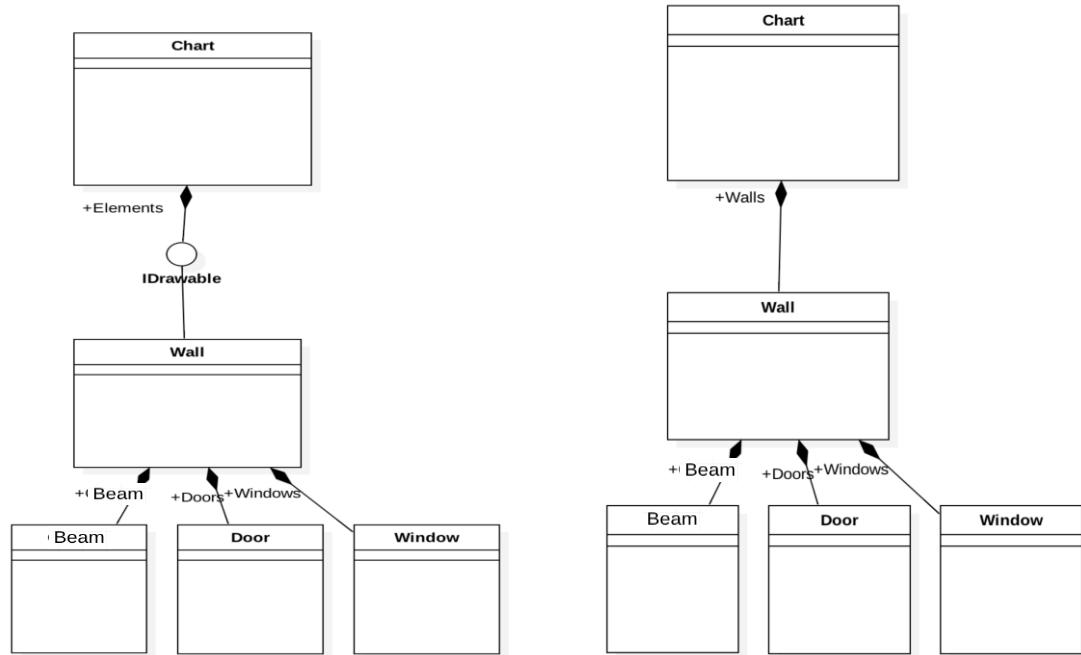
• Haciendo uso de interfaces donde se creyó conveniente se intentó cumplir con el **Principio de Diseño de Sustitución de Liskov** (*LSP: Liskov Substitution Principle*), buscando nunca utilizar RTTI (*Run-Time Type Information*: preguntando por el tipo de objeto) y evitando en lo posible la utilización de *DownCast* (suplantándolo por llamadas a “*new*”). Esto es observable en casos como cuando en el *LogIn*

⁶ Vínculo a Documento de Microsoft sobre Multidimensional Arrays: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/multidimensional-arrays>

se debe diferenciar entre Usuarios, o cuando en *ChartDrawing* o *ChartShowing* se dibujan elementos del Plano, se dibujan en forma polimórfica sin preguntar sobre el tipo de elemento a dibujar.

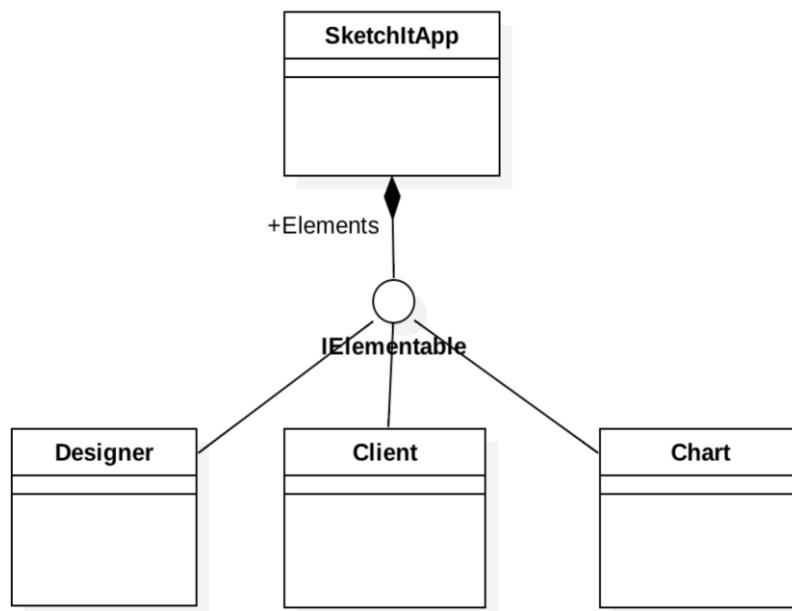
- En la *Form* de *Introduction.cs* se genera una secuencia continua y asíncrona de colores a efectos de brindar un mayor dinamismo al Sistema. Nótese que de la forma que está implementado, esta secuencia es repetida durante todo el tiempo en que la *Form* se encuentra abierta (aún cuando no está visible), lo que consume un procesamiento adicional inútil. Se estudió sin éxito el poder definir una nueva *Thread* al programa sobre la cual efectuar la mencionada secuencia, en la que una vez cambiada de ventana se pueda cerrar.
- La información del Administrador no fue considerada para guardar, puesto que al no poder modificarse se supuso innecesario. Sin embargo, pudo en caso de haberse decidido ser un buen momento para guardar aquí los datos de Costos y Precios de los materiales de elementos del Plano.
- Las clases base *User*, *Worker* y *Hole* fueron creadas a efectos de contener métodos comunes a sus clases derivadas, de tal manera de evitar repetición innecesaria de estos métodos en cada clase hija. A su vez, se plantea la posibilidad de ocurrencia de polimorfismo, al poder un nombre (en este caso un método) poder representar métodos de distintas clases mientras estas estén relacionadas por una abstracción común.
- La contraseña es determinada de tal forma de aportar una mayor seguridad al Usuario, debiendo para ello tener que ingresar al menos un número, un símbolo y una mayúscula. La mayor seguridad estriba en el hecho que aumentan los casos posibles del universo de posibilidades de contraseñas, pudiendo ahora en una carácter de la contraseña dado encontrarse una letra (mayúscula o minúscula), un número o un símbolo.
- Guiándose por la filosofía de *CleanCode*, se priorizó el uso de *Exceptions* al de mensajes de error, para lo cual se crearon excepciones de tal manera de verificar condicionantes a los datos ingresados. Se buscó aumentar la eficiencia de las excepciones al utilizar menos y utilizarlas en la mayor cantidad de casos posibles, puesto que se consideró no necesario definir nombres y casos sumamente concretos y obligar a definir una multitud de excepciones.

- En cuanto a la construcción del Plano, se decidió desde un comienzo que, al no tener por parte del cliente un límite en las dimensiones del mismo y de tal manera de hacerlo escalable, todos debieran tener la misma distancia entre bloques y por tanto la misma correspondencia pixel-metros del Plano. Esto a su vez facilitó en la programación, ya que se evitó tener que hacer cálculos innecesarios como los que serían mandatorios en caso de tener que cada plano adecuarse a un sector fijo y restringido de la pantalla. Además, se evadió el posible desagrado del usuario, justificado en caso que la distancia entre los bloques fuera sumamente pequeña, o incluso de distintas dimensiones de ancho y largo. Nótese el detalle de definir el atributo de Dimensiones del Plano respetando el mismo formato de las herramientas de *System.Drawing*, es decir, primero *Width* y luego *Length* (si bien al Diseñador se le solicitan las medidas en el sentido inverso).



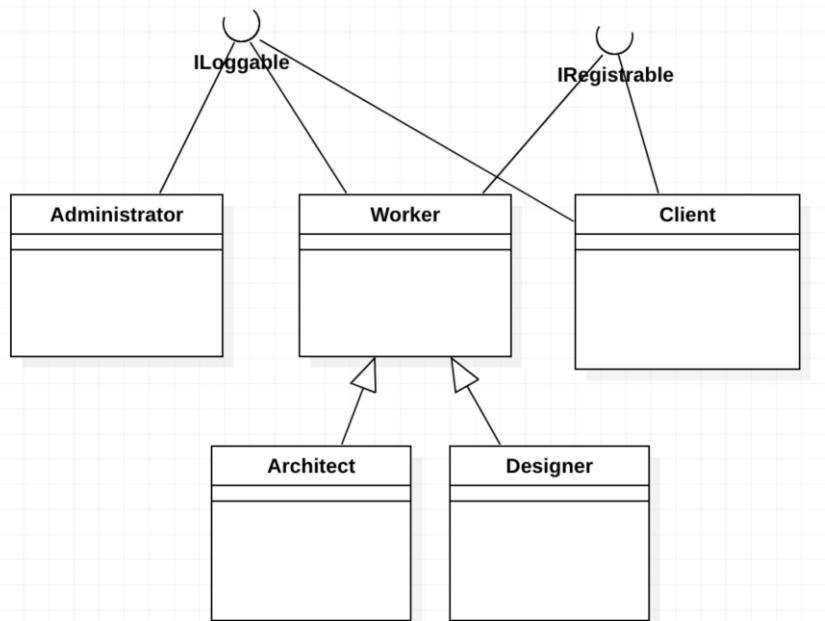
- También se quiere justificar la decisión de las dependencias entre los elementos del Plano. Como se puede observar de los Diagramas UML anteriores, las Clases *Wall*, *Beam*, *Column*, *Door* y *Window* son referenciadas por la Interfaz *IDrawable*, y luego es la Clase *Chart* la cual distribuye sus instancias y características. Esto se decidió así de tal manera de buscar cierta centralización de las responsabilidades en una clase, y disminuir la complejidad del código y las interrelaciones entre clases. Tal es así que se descartaron otras posibilidades como la mostrada en las Figuras superiores, ya que en este caso existirían dos capas de complejidad a las que atender (primero de la clase *Wall* y luego de *Chart*).

- Como se mencionó antes, se utilizó en este trabajo solo un paquete por Proyecto debido principalmente a buscar evitar complejidad innecesaria. Además, al considerarlo, se observa que de haber dividido el Proyecto Dominio en 2 paquetes: por ejemplo *Users* (para *User*, *Administrator*, *Designer* y *Client*) y *Drawing* (para *Chart* y sus elementos), habría existido un conflicto en la Clase *SketchItApp*, ya que inicialmente no pertenecería a ningún paquete. En caso de elegir agregarlo a *Users*, habría que adecuarla para la interacción con *Drawing*, y viceversa.



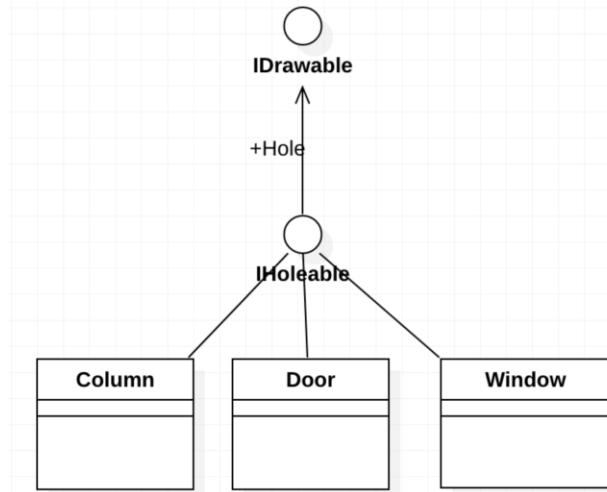
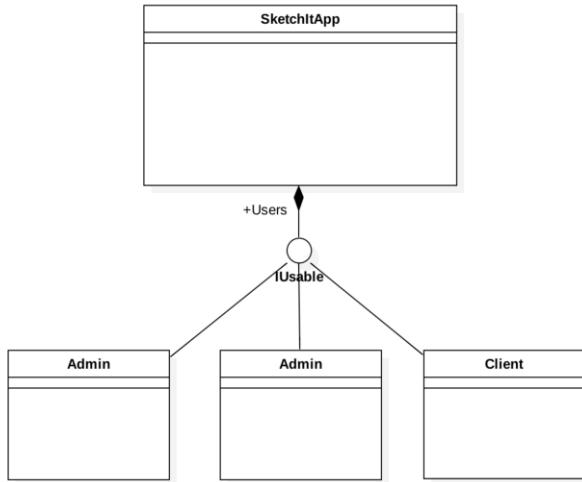
- Actualmente la Clase *SketchItApp* posee como atributos principales el tener un Administrador, una Lista de Usuarios, una Lista de (Tipos de) Aberturas y una Lista de Planos. Esta decisión fue realizada en base a buscar tener una mayor claridad en el código y facilitar cambios futuros, tal como fue mencionado en un punto anterior. Otra posibilidad era la de estructurar según el formato mostrado en la Figura superior, donde la Clase *SketchItApp* tuviera una sola Lista (en vez de 3 Listas) de instancias de la Interfaz *IElementable*, la cual contuviera el contenido de las 3 Listas antes mencionadas. Esta opción fue descartada al considerarla innecesaria y sumamente compleja para el diseño solicitado, ya que cada uno de estos 3 componentes (*Designer*, *Client* y *Chart*) poseen un comportamiento y flujo por la aplicación bien marcado y diferente entre sí, que terminaría por obligar al equipo a definir múltiples funciones vacías en cada clase interviniente, para justificar por la existencia de una función útil en una de las clases componentes.

- Guiándose por el **Principio de Diseño de Segregación de Interfaces** (*ISP: Interface Segregation Principle*), se buscó realizar una reutilización de las *Forms* definidas que poseerían en otro caso funcionalidades similares, para de esta forma generar una mejor (más eficiente) gestión de los recursos del Sistema. De esta forma es que las *Forms* de *DataEditing.cs*, *PasswordEditing.cs* y *ChartShowing.cs* fueron compartidas (con leves cambios) entre los usuarios Arquitectos, Diseñadores, Clientes (las 3 mencionadas) y Administrador (la primera mencionada). Este fue el principal motivante para la creación y utilización de las interfaces *ILoggable.cs*, *IRegistrable.cs* e *IWindowsChangeable.cs*, esta última definiendo la firma de una función que posibilita el retorno a la *Form* previa al ingreso de la *Form* actual, al igual que diferencia el comportamiento consecuente de que un Arquitecto modifique un Plano firmado y luego salga de la Edición, debiéndole solicitar una nueva Firma.

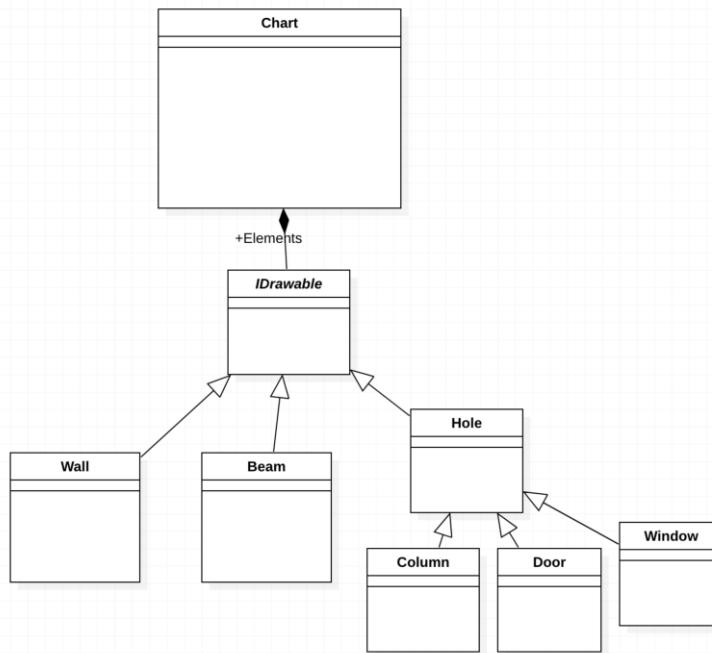


- Las Interfaces fueron creadas bajo el preconcepto de separar cada interfaz por su motivo de existir, el cual fue explicitado en la unidad “Descripción General del Sistema”. Con esto presente, la solución posee 4 Interfaces: 1 para implementación y utilización en el Proyecto Domain (*IDrawable.cs*), 1 para implementación y utilización en el Proyecto GUI (*IWindowsConfigurable.cs*) y 2 para implementación en Domain y utilización en GUI (*ILoggable.cs* e *IRegistrable.cs*). De estas últimas, se decidieron separar por una cuestión de prolijidad, implementándose *IRegistrable.cs* en *Worker* y *Client*, e *ILoggable.cs* en *Worker*, *Client* y *Administrator*. De esta forma, se evitó una vez más la definición de funciones vacías en las clases implementadoras, además de mantener la cadena lógica de utilizar

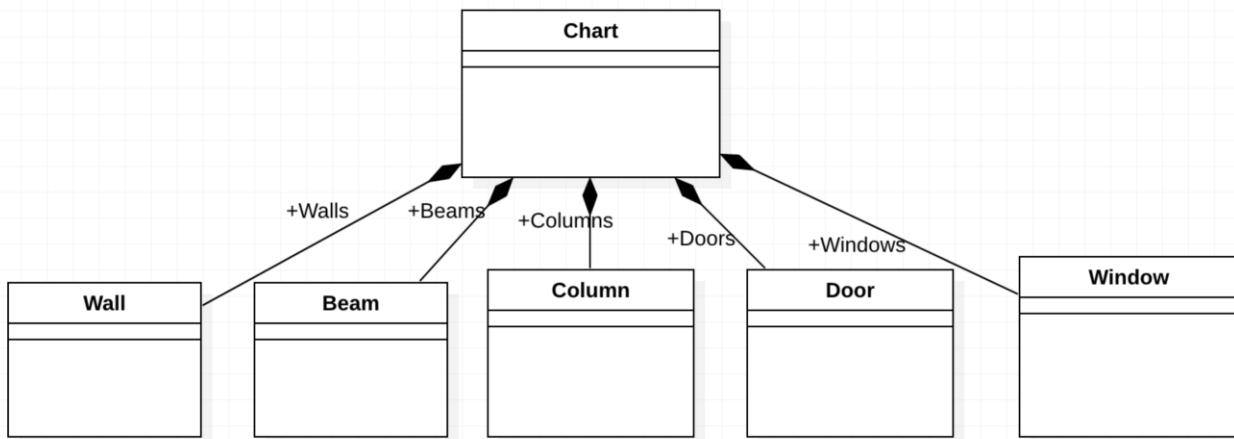
ILoggable.cs para funciones relacionadas al *Log In*, e *IRегистrable.cs* para funciones relacionadas al Registro y Edición de Usuarios.



- Otra alternativas consideradas fueron las presentadas en las Figuras superiores, en las que se intenta sustituir Herencia mediante relación entre las clases base *User* y *Hole* y sus derivadas, con las Interfaces *IUsable* e *IHoleable* respectivamente. Esta nueva configuración, si bien posible e implementable, genera el tener que repetir el código ahora presente en *User* y *Hole* en todas sus clases derivadas, lo cual entendemos es una mala decisión de diseño y de profesionalismo en el código. Sin embargo, entendemos que esta forma permite un control más centralizado de las identidades, al referirse directamente a las correspondientes Interfaces en vez de a cada Usuario o Abertura.



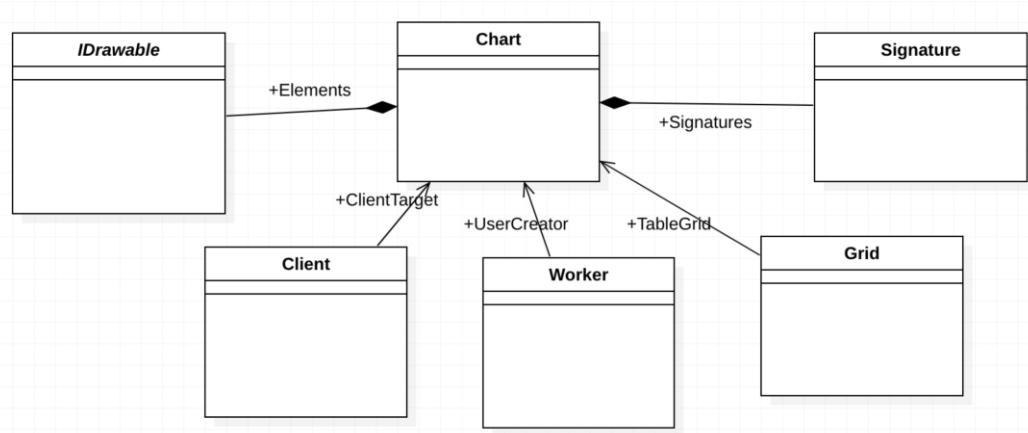
- Actualmente la Clase Chart hace uso del **Patrón de Diseño Estructural conocido como *Facade* (o *Façade*)** al utilizar una Lista de Instancias de Interfaces *IDrawable*, las cuales hacen referencia a instancias de las clases *Wall*, *Beam* y *Hole*. Esto genera que en caso de tener que generar otra Abertura (tal como ocurrió en el caso de las Columnas Decorativas) se la pueda agregar fácilmente haciéndola heredar de la Clase base *Hole*, haciendo entonces que implícitamente implemente a la Interfaz *IDrawable*, definiendo los métodos apropiados. Esto por otro lado genera problemas al momento de querer agregar elementos a la Lista, puesto que esta debe diferenciar que elemento se encuentra agregando, lo cual se buscó solucionar generando un método diferente para el agregado de cada tipo de elemento. También, provoca la necesidad de tener que ordenar los elementos de la Lista *IDrawable*, para que al momento de dibujar los elementos en el Plano no se solapen entre sí (ya que se dibujan en orden en que se encuentran posicionados dentro de la Lista). Esto inevitablemente provoca que se haya definido la firma de un método *IsWall()* dentro de Interfaz *IDrawable*, que identifique que elementos son Paredes y las traslade para el inicio de la Lista, de tal forma que sean dibujadas primero y luego todas las Columnas y Aberturas. Al final, la desventaja más notoria de la utilización de la Interfaz *IDrawable* es la necesidad de establecer funcionalidades de Interfaz Gráfica en el Proyecto del *Domain*.



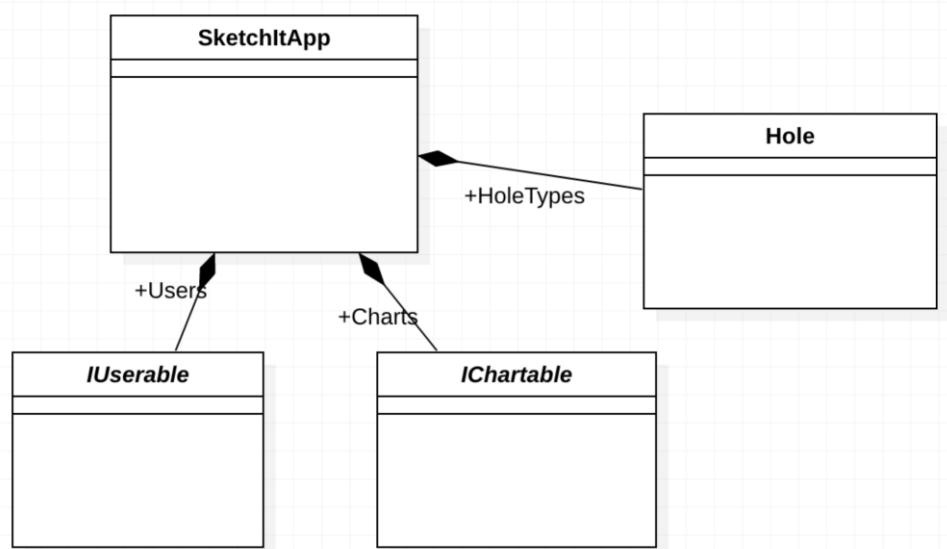
- Sin embargo, respecto al punto anterior, se consideró previamente realizar una arquitectura de diseño análoga a la mostrada en la Figura superior, en la que la Clase Plano (*Chart*) poseía una Lista de Paredes, una Lista de Vigas, una Lista de Columnas, una Lista de Ventanas y una Lista de Puertas. Esto poseía la ventaja de ser más sencillo de programar y diferenciar entre los distintos elementos, pero no era eficiente al tener que la Clase *Chart* administrar 4 Listas en simultáneo, debiendo continuamente que iterar entre todas las listas al dibujar un Plano con sus elementos (lo cual se realiza cada vez que se produce un cambio en el Plano). A la vez, en este caso al necesitar agregar una Abertura, si bien se podría igualmente heredar de *Hole* y tener los métodos de esta clase base, se deberá definir todo un comportamiento específico nuevo para esta nueva Abertura, a la vez que agregar una sexta Lista a la Clase *Chart*.

- En Conclusión, se hace referencia a que, tal como se previó en el Obligatorio anterior luego de un análisis del diseño final del trabajo, el principal limitante al momento de realizar modificaciones, si bien dependiendo de que modificaciones estas sean, fue en la adecuación de la ventana de las *Forms* necesarias. Esto no hubiese sido necesario en caso de haber buscado reutilizar *Forms*, en vez de realizar una diferente, por ejemplo, para cada Menú. Tal como fue predicho en el Obligatorio 1:

- **Al agregar un nuevo tipo de Usuario (Caso Arquitecto):** Se deberá crear un nuevo Menú correspondiente (salvo reusar alguno de los Menús ya realizados) y muy posiblemente, modificar las transiciones entre *Forms* que generan las características y funcionalidades del nuevo Usuario en la Aplicación.
- **Al agregar un nuevo tipo de Abertura (Caso Columna Decorativa):** Se deberá modificar la ventana de la *Form ChartDrawing.cs* de manera de posibilitar la selección e inserción de la nueva Abertura.
- Sin embargo, nótese que globalmente el diseño presentado en el Primer Obligatorio fue lo suficientemente escalable como para unas relativamente pocas modificaciones al momento de implementar las nuevas funcionalidades solicitadas en el Segundo Obligatorio. Si se observa, muchas de las modificaciones realizadas y mencionadas como más notorias fueron hechas de forma de facilitar una futura mejor mantenibilidad del Sistema, y atenerse a los Principios y Patrones de Diseño explicitados y recomendados por los docentes.



- La posibilidad de colocar Columnas decorativas ya se encontraba siendo anunciada en el Primer Obligatorio, y solo se debió agregar una clase derivada más de la Clase *Hole* y definir correctamente el método que lo dibuje en el Gráfico correspondiente. En el caso particular, se debió a su vez modificar el nombre de toda una Clase, puesto que la ahora conocida Clase *Beam* había sido llamada en el Primer Obligatorio como Clase *Column*. Aún así, como ya fue notado, la mayor complicación resultó al momento de añadir esta nueva funcionalidad en la Interfaz Gráfica, debiendo generar la opción y verificar todas las condiciones de relacionamiento con los demás elementos del Plano.
- Se cree que la personalización de la Grilla del Plano, si bien directamente implementada utilizando el Patrón de *Strategy*, nunca pudo haber sido previamente predicha por el diseñador del software para el diseño concreto de la Aplicación, puesto que consisten en características concretas e independientes de toda lógica del Programa. En lo mismo puede referenciarse a la existencia de Firmas en un Plano.
- El agregado de un nuevo estilo de Usuario como fue el Arquitecto fue lo más complejo en agregar, tanto por tener que crear nuevas *Windows Forms* como por la cantidad de clases que se debieron agregar y relacionar en consecuencia a efectos de poder cumplir con sus nuevas funcionalidades requeridas. Por ello se buscó reestructurar todo aquello mencionado relacionado con las clases de Usuarios, para así similares modificaciones puedan ser más sencillas de implementar en el futuro



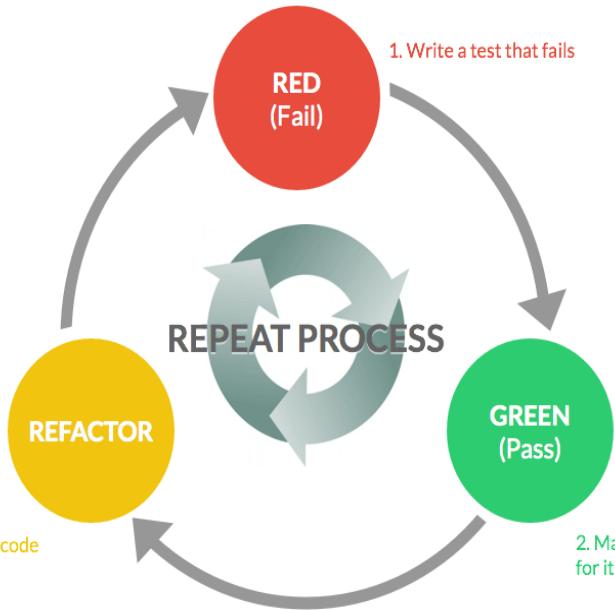
● Por último, el posibilitar al Usuario poder definir nuevos Tipos de Aberturas fue complejo de incluir a la lógica del programa, si bien no de extensión y elementos a incluir. Se debió agregar la opción en el Menú del Arquitecto y del Diseñador para que puedan crear Aberturas, crear una nueva *Windows Form* que se encargue de definir y registrar estas nuevas Aberturas por parte del Ingresante, y agregar una nueva opción al momento de dibujar el Plano para elegir sobre el tipo de Abertura a colocar. Estas variaciones del comportamiento predeterminado del Sistema pudieron haber sido previstas, pero no en concreto respecto a lo que realmente se termine precisando. Haciendo uso de una Lista de Interfaces en vez de una Lista de (Tipos de) Aberturas en la Clase *SketchItApp*, se podrá tratar de forma polimórfica a todo tipo similar de modificación que se busque agregar en el futuro. Sin embargo, utilizando *Entity Framework* y debiendo definir a *IDrawable* como una Clase Abstracta de la cual herede *Hole*, *Visual Studio* no permitió poder colocar otra Clase Abstracta (en defecto de Interfaz) entre *SketchItApp* y *Hole* para poder definir cualquier otro tipo de Abertura, ya que el entorno no posibilita Herencia Múltiple. Por tanto, se debió permanecer en la relación mostrada en la imagen, utilizando en *SketchItApp* una Lista de *Hole* directamente. La gran mayoría de las posibilidades de interacción entre distintos tipos de Aberturas fueron abarcadas, pero no se descarta la posibilidad de algún caso sin resolver.

- Nótese nuevamente que se pudieron haber agregado una mayor cantidad de interfaces, pero se optó por disminuir la complejidad y facilitar la compatibilidad con *Entity Framework*. Esto se encuentra claramente resaltado en los diseños presentados anteriormente en los Diagramas UML del proyecto *Domain*, en donde se observan algunos de los cambios necesitados para alcanzar tal compatibilidad.
- Respecto a la Base de Datos resultó muy complejo el tratamiento con entidades como la *IDrawable* presentada, por lo que se debieron realizar múltiples modificaciones tales como fueron mencionadas. Se concluye por tanto que, arribando a un resultado que no nos dejó satisfecho, se ha utilizado un diseño no completamente compatible con aquel soportado por *Entity Framework*, si bien se considera que el utilizado es sumamente eficaz por los motivos mencionados. Se considera que se realizó lo mejor con las herramientas obtenidas en el tiempo disponible.

- Por tanto, se adjunta a la entrega final del trabajo una versión sin Base de Datos pero Serializable, con todas las funcionalidades incluidas y funcionales, habilitando el siguiente vínculo: https://drive.google.com/open?id=13Fs4KDOdiMpy8iq30elVlHubbnrp_an . Esta versión no será modificada luego del día de la Entrega.

Nota: Para los Diagramas realizados en esta unidad se utilizó *StarUML* v. 3.0.0 (actualizando desde el formato utilizado para el Obligatorio 1: *StarUML* v. 2.8.1), el cual se descubrió dibuja en forma inusual la relación de Referencia respecto a una Interfaz. En estos casos (tales como el caso de la Figura entre *Designer*, *Client* y *Chart* para con *IElementable*), la flecha la realiza continua, cuando en realidad debería ser punteada. A su vez, nótese como las Clases Abstractas son representadas con letra Cursiva (*Italic*).

Cobertura de Pruebas Unitarias:



Date	Author	Commit
15 May 2018 17:11	Carolina <carorod>	07ba06
15 May 2018 17:08	Carolina <carorod>	504856b
15 May 2018 17:06	Carolina <carorod>	2f3d0b1
15 May 2018 17:02	Carolina <carorod>	4fcde11
15 May 2018 16:56	Carolina <carorod>	44a8de8
15 May 2018 10:56	Carolina <carorod>	61394cb
15 May 2018 10:55	Carolina <carorod>	d8fec0
15 May 2018 10:52	Carolina <carorod>	89deeba
14 May 2018 19:25	Carolina <carorod>	6c29862
14 May 2018 19:14	Carolina <carorod>	b0b06d5
14 May 2018 17:44	Carolina <carorod>	2bdc8bf
14 May 2018 17:38	Carolina <carorod>	93b74a1
14 May 2018 17:35	Carolina <carorod>	b3bae55
9 May 2018 16:53	Carolina <carorod>	3e5020a
9 May 2018 16:52	Carolina <carorod>	e2e0c76
8 May 2018 17:44	Carolina <carorod>	d0fd565

En el transcurso del trabajo se trabajó utilizando el procedimiento o modalidad TDD (*Test-Driven Development* o *Test-Driven Design*), realizando las pruebas antes de realizar los métodos necesitados. Se necesitó por tanto pensar en todas las posibles salidas y comportamientos que los métodos deberían modificar, de manera de tener los métodos una respuesta cercana a la esperada. Para ello, se debió utilizar un ciclo *Refactoring, Red, Green, Blue* de tal manera de testear cada posible situación encontrada que sea posible de ocurrir en el Sistema. En consecuencia se realizaron un total de 338 pruebas, todas exitosas, que lograron abarcar un 98.079% del código, habiendo testeado únicamente el Proyecto *Domain* como consigna del Obligatorio. Esta cobertura fue conseguida sin hacer uso de las exclusiones utilizadas en el anterior Obligatorio a efectos de no testear por ejemplo los métodos de *Draw*. Esto, si

bien como se observa en el código se pudo realizar, no se considera lógico puesto que en algunos casos se debió probar métodos que solo retornaban un valor, y si ese valor era retornado (*null* o *false* generalmente), al igual que en otros casos se debió “probar” métodos vacíos. Se aprovechó la oportunidad y para este Obligatorio se logró efectivamente el mecanismo para probar los métodos no probados totalmente en el Obligatorio anterior.

The screenshot shows the Test Explorer window in Visual Studio. It displays a list of 338 passed tests under the category 'Passed Tests'. One specific test, 'TestAddWallLengthNull', is highlighted with a blue selection bar at the bottom of the list. The test names and their execution times are listed as follows:

- TestAddBeamsDueToLength: 2 ms
- TestAddBeamsDueToLengthWhenNotNecessary: < 1 ms
- TestAddChartsSigned: 1 ms
- TestAddClientsHelped: < 1 ms
- TestAddColumn: < 1 ms
- TestAddColumnWhenNotWellRegistered: < 1 ms
- TestAddDoor: < 1 ms
- TestAddDoorWhenNotWellRegistered: < 1 ms
- TestAddLengthToOriginWhenHorizontalCoord0To1: < 1 ms
- TestAddLengthToOriginWhenHorizontalCoord1To0: < 1 ms
- TestAddLengthToOriginWhenVerticalCoord1To0: < 1 ms
- TestAddWallAlreadyRegistered: < 1 ms
- TestAddWallLengthNull**: < 1 ms
- TestAddWallTrue: < 1 ms
- TestAddWindow: < 1 ms
- TestAddWindowWhenNotWellRegistered: < 1 ms
- TestApproximateCoordinate: < 1 ms
- TestApproximateLinesDown: < 1 ms
- TestApproximateLinesUp: < 1 ms
- TestArchitectAddChartsAndClientsHelped: < 1 ms
- TestBeamIsBigDoorOfPoint: < 1 ms
- TestBeamIsBigDoorOfPoint: < 1 ms
- TestBeamIsPointOnHole: < 1 ms

The screenshot shows the Code Coverage Results window in Visual Studio. It displays coverage statistics for the project 'ferhernandez_FERNANDOHERF66E 2018-06'. The table provides the following data:

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
ferhernandez_FERNANDOHERF66E 201...	126	1.921%	6432	98.079%
domain.dll	108	3.213%	3253	96.787%
tests.dll	18	0.563%	3179	99.437%

Para documentar el desarrollo del proyecto en TDD se utilizo la herramienta *SourceTree*. A lo largo del todo el proyecto se tuvieron numerosos problemas con dicho programa, llevando a tener que eliminar el repositorio dos veces.

En nuestro repositorio las clases de las cuales mas se documentó (Todas las clases fueron realizadas con TDD) su desarrollo en TDD fueron *User*, *Client*, *Designer* y *Administrator*.

Para cada prueba se tomaron 3 etapas, *RED STAGE* que es cuando se piensa la prueba y se escribe antes de su método y se verifica que dicha prueba falla. Luego esta *GREEN STAGE*, en esta etapa el objetivo de la prueba es que pase con los mínimos cambios. Y por último esta *BLUE STAGE* donde se realiza *Refactoring* la prueba y el código para que pase satisfactoriamente.



- BLUE STAGE. TestIsNumberIDRightTrueWithNew pass successfully
- GREEN STAGE. TestIsNumberIDRightTrueWithNew.
- RED STAGE. TestIsNumberIDRightTrueWithNew.

En la Figura superior se muestran la cobertura de las pruebas por Clases del Proyecto *Domain*. Como se puede observar, se pudo llegar a una completitud de testeo de todos los casos de prueba realizados, los cuales se pueden observar en las propios métodos de prueba por clase. Cada *TestMethod* fue programado de forma de abarcar un caso particular de prueba. En los casos particulares en que no se llegó a abarcar en testeo (concernientes al mínimo de la Clase *Chart*, al mínimo de la Clase *SketchItApp* y al mínimo de la Clase *User*) fueron realizados por falta de un caso concreto de prueba que realizar. Aquello que más deteriora la cobertura de las pruebas es la no cobertura (por imposibilidad) de los métodos de las Clases Abstractas (antes Interfaces) *IDrawable*, *IChartable* e *IUsable*. Sírvase ver la captura extraída en la Página siguiente.

Nótese que el no llegar al 100% de la cobertura en tests.dll ocurre debido a unos paréntesis que, al probar excepciones, nunca llegaron a ser alcanzados, tal como es denotado en la imagen ilustrativa debajo, extraída a modo de captura de pantalla de nuestro programa.

Code Coverage Results					
Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)	
fer hernandez_FERNANDOHERF66E 201...	126	1.921%	6432	98.079%	
domain.dll	108	3.213%	3253	96.787%	
Domain	108	3.217%	3249	96.783%	
Administrator	0	0.000%	29	100.000%	
Architect	0	0.000%	36	100.000%	
Beam	2	3.571%	54	96.429%	
Chart	40	2.761%	1409	97.239%	
Chart.<>c	0	0.000%	1	100.000%	
Chart.<>c_DisplayClass59_0	0	0.000%	2	100.000%	
Chart.<>c_DisplayClass61_0	3	50.000%	3	50.000%	
Chart.<>c_DisplayClass62_0	0	0.000%	6	100.000%	
Chart.<>c_DisplayClass72_0	0	0.000%	2	100.000%	
Chart.<>c_DisplayClass91_0	0	0.000%	2	100.000%	
Client	12	5.195%	219	94.805%	
Column	2	4.878%	39	95.122%	
DashedGrid	0	0.000%	29	100.000%	
Designer	0	0.000%	47	100.000%	
Door	2	0.870%	228	99.130%	
DoubleContainer	2	25.000%	6	75.000%	
Grid	0	0.000%	27	100.000%	
Hole	0	0.000%	85	100.000%	
IDrawable	2	100.000%	0	0.000%	
IntContainer	2	25.000%	6	75.000%	
InvisibleGrid	0	0.000%	29	100.000%	
MoneyContainer	12	50.000%	12	50.000%	
PointContainer	2	33.333%	4	66.667%	
PointCoordinatesContainer	2	20.000%	8	80.000%	
Signature	4	6.154%	61	93.846%	
SketchItApp	6	2.564%	228	97.436%	
SketchItApp.<>c	0	0.000%	6	100.000%	
SketchItApp.<>c_DisplayClas...	0	0.000%	3	100.000%	
SketchItApp.<>c_DisplayClas...	0	0.000%	3	100.000%	
SketchItApp.<>c_DisplayClas...	0	0.000%	2	100.000%	
SketchItApp.<>c_DisplayClas...	0	0.000%	2	100.000%	
SketchItApp.<>c_DisplayClas...	0	0.000%	2	100.000%	
SketchItApp.<>c_DisplayClas...	0	0.000%	2	100.000%	
SketchItApp.<>c_DisplayClas...	0	0.000%	3	100.000%	
User	2	1.869%	105	98.131%	
Wall	2	0.580%	343	99.420%	
Window	2	1.258%	157	98.742%	
Worker	9	15.517%	49	84.483%	
Domain.Exceptions	0	0.000%	4	100.000%	
AlreadyExistingUserException	0	0.000%	2	100.000%	
InvalidOperationException	0	0.000%	2	100.000%	
tests.dll	18	0.563%	3179	99.437%	