

Universidad ORT Uruguay

Facultad de Ingeniería

Obligatorio Diseño de Aplicaciones 1
Entregado como requisito para la obtención del título
de Ingeniería en Sistemas

Ignacio Ramírez (259413)

Pablo Torres (257145)

Docentes:

Pablo Geymonat - Joaquín Anduan - Joaquín Mendez

Repositorio en:

https://github.com/ORT-DA1-2023/257145_259413

Grupo N5A – 2023

Índice

Para la segunda entrega se modificaron los distintos puntos del índice previamente realizados para la entrega número uno.

Para mantener una constancia de lo realizado en el primer obligatorio y poder hacer comparaciones se colocó la información anterior en el anexo.

Metodología de trabajo:	3
Clean code:.....	3
Descripción del sistema:	4
Características de la página:.....	4
Creación de diagramas:	6
Diagrama de Paquetes:.....	6
Diagrama de Clases:.....	7
Domain:.....	7
BusinessLogic:.....	9
Engine:.....	10
DataAccess:.....	11
Diagrama de Interacción:.....	12
Test-driven development (TDD):	13
Cobertura:	14
Fundamentos de diseño de software:.....	15
Patrones GRASP/Principio SOLID:.....	15
Patrones de diseño:.....	16
Mapeo de Objetos Relacionales:.....	17
Entity Framework:.....	17
EagerLoading/Lazy Loading:.....	17
Mapeo de herencias:.....	18
Mecanismo general del sistema:	19
Errores corregidos del Obligatorio 1:	19
Anexo:	20
Diagramas Entrega 1:.....	20
Diagrama de Paquetes: (Anterior).....	20
Domain: (Anterior).....	20
BusinessLogic:(Anterior).....	21
Diagrama Engine: (Anterior).....	21
Reporte Evidencia: (Anterior)	21
Respuesta al Archivo Errores Comunes en los Obligatorios: (Anterior)	22

Metodología de trabajo:

El equipo optó por realizar el trabajo vía Discord, en algunas ocasiones compartiendo pantalla y otras de manera asincrónica ayudándonos con la Herramienta **Github** que nos permite trabajar y hacer cambios de manera independiente en el proyecto.

Además utilizamos un flujo de trabajo definido por esta herramienta llamada **GitFlow**, la cual nos dice que debemos tener dos branches llamadas **master** y **develop**.

También para trabajar en diferentes funcionalidades se creó features branch, las cuales van a contener la implementación de cada requerimiento pedido por la letra.

Una vez culminada estas implementaciones se mergeo a la rama develop donde contenemos las funcionalidades estables, por último cuando tuvimos nuestra versión final del proyecto se hizo una subida/merge a la branch master.

Hay que tener en cuenta que para la segunda entrega se profundizó en esta metodología (gitflow) y se creó por cada requerimiento funcional una rama.

Este fue un cambio primordial ya que para la primer entrega no se realizado correctamente.



Clean code:

Algo sumamente importante que tuvimos en cuenta antes de comenzar a trabajar fue definir una serie de principios los cuales se conocen como “Principios de clean code”.

Se realizó en esta etapa para que sea fácilmente de leer y poder llevar un orden en el código en nuestro equipo de trabajo.

Algunos principios que usamos fueron:

- Contener la mínima cantidad de comentarios posibles, solamente si fuese esencial para el código.
- Tener el código en el idioma inglés.
- Los nombres de métodos y variables tienen que ser claros como descriptivos, que se pueda entender con una lectura, además esto nos da facilidad al cambio a priori.
- Usar **Regla Step-Down Rule** (Cada función debe ser seguida por la del siguiente nivel de abstracción).
- Usar excepciones.

- Tener en cuenta que cada función contenga una sola responsabilidad y que reciba pocos parámetros.
- Los nombres de clases son objetos de la vida real, además de ser sustantivos.
- Los nombres de funciones tienen que ser verbos.

Descripción del sistema

Características de la página:

En la letra dada se nos pide que realicemos una página donde se pueda crear imágenes 3D utilizando la técnica de renderizado llamada Ray Tracing.

Primeramente lo que el usuario ve en nuestra plataforma es una landing page, donde se puede loguear si ya cuenta con un nombre de usuario y una contraseña válida, en caso contrario podrá acceder mediante un hipervínculo a otra sección de nuestro sitio web donde podrá crear un usuario.

Vista de ingreso:

Vista para generar usuario:

Luego de su ingreso exitoso, se lo redirigirá a una sección donde podrá crear una figura, agregarle su nombre y un radio específico. También en esta división se puede observar 3 botones los cuales sirven para “Crear”

la figura anteriormente nombrada, otro para “cancelar” que lo redirigirá a una sección donde podrá “Cerrar sesión” o “comenzar nuevamente”.

Por último se cuenta con un botón “Ver Figuras” lo cual nos lista las figuras con sus características anteriormente colocadas por el usuario,

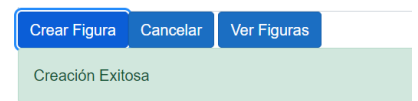
FigureList

[Volver](#)

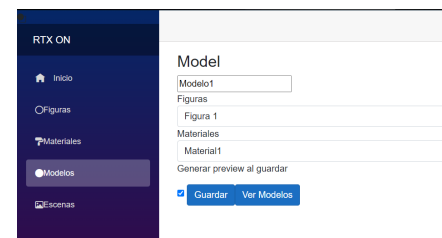
Nombre	Radio	Acciones
Hola12	1	Eliminar
Figura1	2	Eliminar
Figura2	5	Eliminar

Además esta sección contiene un botón de “eliminar” para quitar la figura a elección del cliente.

Hay que tener en cuenta que todos estos pasos de creación son validados con un mensaje, lo cual ayuda al usuario a saber si se realizó con éxito el paso efectuado.



También se puede observar al lado izquierdo de la pantalla un nav que nos permitirá navegar entre las diferentes secciones, que nombraremos más adelante.



Luego de crear una figura el usuario puede crear un material, donde se le pedirá el nombre, el color (RGB) y seleccionar el tipo de material a elección el cual puede ser Lambertiano o Metálico, si se elige la última opción se le dará al usuario un input para poder colocar el valor de difuminado.

Como en la sección anterior el usuario podrá ver una lista de sus materiales y eliminar a elección los mismos desde otra sección.

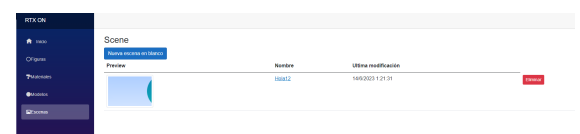
De la misma manera a través del **Nav** el usuario podrá acceder a Modelos, donde se le pedirá un nombre para el modelo a crear, además se le hará elegir una figura y un material anteriormente creado por él a través de un **Select**.

También tendremos un **checkbox** donde se podrá elegir generar una preview del objeto creado.

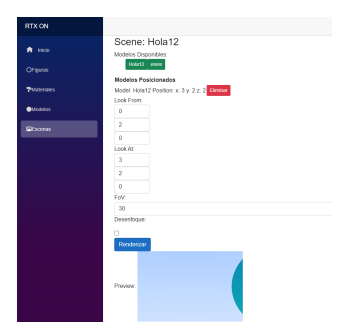
Mismamente que las secciones anteriores el cliente podrá acceder a una lista y podrá eliminar a elección sus modelos.

Por último tendremos una sección llamada Escenas, donde se muestra un botón, el cual crea una escena en blanco con un nombre específico.

Al instante de crearla se mostrará en una lista el nombre y su última modificación además de un botón de eliminar.



Por medio de un hipervínculo se podrá acceder a una sección que nos permitirá personalizar la escena, mediante la creación de modelos posicionados pasándole por medio de un **Input** valores de coordenadas.

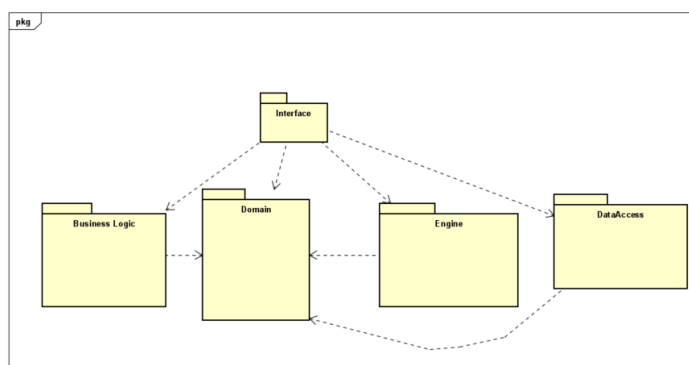


Por último en esta sección tendremos un **botón** el cual nos permitirá renderizar escena y poder verla en diferentes ángulos ya que podremos cambiar las coordenadas de la cámara, además de poder guardar la escena en 3 tipos de formatos (PNG,JPG,PPM).

Creación de diagramas:

(Todos los diagramas se encuentran en la carpeta Diagramas para su mejor visualización)

Diagrama de Paquetes:



Se optó por dividir nuestro proyecto en diferentes paquetes ya que es de gran importancia tener un bajo **Acoplamiento** entre nuestros diferentes componentes que integran nuestro sistema, esto nos va a dar una ventaja a futuro, ya que vamos a tener un código fácilmente escalable y fácilmente mantenible con respecto a códigos que no usen esta técnica.

Además podemos notar que hay una **Cohesión** entre los componentes que tenemos en cada paquete ya que se relacionan entre sí, esto nos ayudará a entender mejor el código y obviamente a su mantenibilidad.

Hablaremos sobre los principales paquetes contenidos en nuestro sistema:

Tenemos el paquete llamado **Domain**, el cual contiene entidades que son sumamente importante para el proyecto, “Nuestro pilar” del cual vamos a partir. Es importante que solamente contenga los objetos de uso con sus características.

El paquete **BusinessLogic** es el cual se va a encargar de darle funcionamiento a nuestro sistema, es decir este se encarga de las operaciones que se le realizan a los objetos nombrados anteriormente, como por ejemplo es el encargado de contener los métodos “deleteFigure”, “addFigure” a una lista, “login” entre otros.

De igual manera se creó un paquete llamado Interface el cual va a contener todo lo relacionado con la interfaz de usuario que nos va a proveer **Blazor** (herramienta de desarrollo web) ayudándonos a darle funcionalidad a nuestros objetos.

En esta biblioteca de software contaremos con componentes como botones, selects, checkbox y todo lo necesario para poder crear una página interactiva con el usuario, además de contener todo los elementos visuales que se le mostrarán a nuestro “cliente”.

Tendremos el paquete **Engine**, el cual es sumamente importante a la hora de cumplir con el objetivo de este Obligatorio ya que va a contener todo lo relacionado con motor gráfico, con el podremos realizar la renderización pedida ya que contiene herramientas para cumplir con esto.

Para la segunda entrega se agregó el paquete DataAccess, el cual va a cumplir un rol fundamental para poder darle funcionamiento a nuestra base de datos.

Por último tendremos un paquete que se va a encargar del uso de **Excepciones**, gracias a esto las podemos hacer personalizadas con mensajes no genéricos para cada uno de ellas. Además es un gran beneficio a la hora de programar ya que estamos centralizando acciones similares en un mismo sector dándonos un beneficio a priori a la hora de implementar cambios en ellas y/o aumentar la cantidad, a su vez brindándonos un menor tiempo de búsqueda.

También, notamos que la relación entre paquetes es de **Dependencia**, deja claro que un paquete depende del otro, dándonos a entender que si se efectúa un cambio en un paquete este puede alterar el comportamiento del otro.

Por ejemplo la relación de dependencia que tenemos de los paquetes Domain y Engine se da ya que en la clase “Render” se crea un vínculo de dependencia con la clase “Scene”.

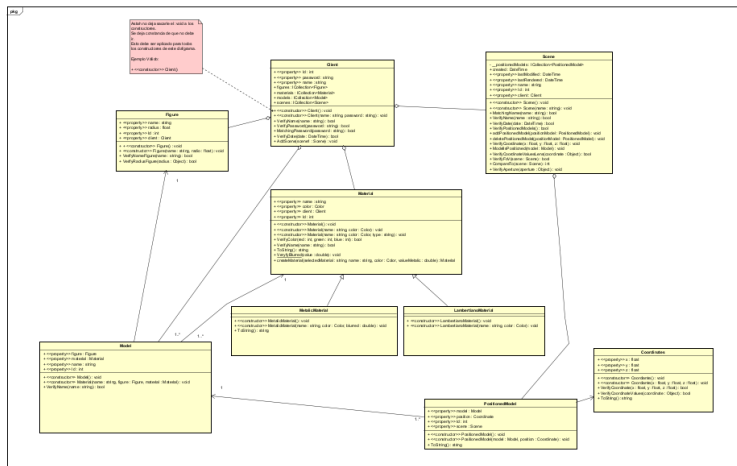
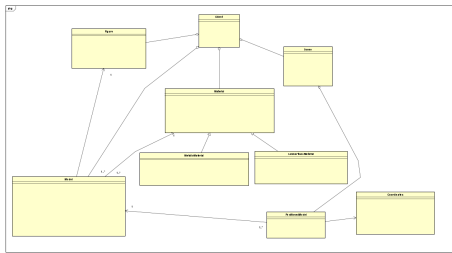
Por último, no menos importante podemos notar la dependencia que se realiza hacia el paquete Exceptions con los paquetes Interface y BusinessLogic, esto se da ya que se utilizan try and catch en varias clases de los paquetes anteriormente nombrados.

Cabe destacar que esta separación de “Responsabilidades” nos van a ayudar a futuro como anteriormente se mencionó.

Diagrama de Clases:

Domain:

(Se adjunta UML sin Atributos ni métodos como dice documentación).



Nos dimos cuenta que la clase “Client” concentra gran parte de la lógica de este proyecto, por lo tanto comenzamos observando sus atributos y los diferentes tipos de relaciones con las demás clases.

Se utilizó la relación **Agregación** en lugar de la **Composición** para vincular la clase “Cliente” con “Figure”, “Scene”, “Material” y “Model” ya que al momento de crear un cliente este no necesariamente debe tener una figura, escena ,material o un modelo, sabemos que el uso de composición nos estaría obligando a tener por lo menos un objeto de estas clases. También lo podemos notar con una simple frase “El cliente puede o no tener una figura” así para todas las clases anteriormente nombradas.

También podemos observar en este UML el uso de **Asociación** sobre las clases “Model”, “Figure”, “Material”, esto refiere a que modelo conoce o está asociada con las clases “Figure” y “Material” pero no recíprocamente.

Además podemos observar que la **Multiplicidad** de esta Asociación es de 1 .. * con una **Navegabilidad** yendo hacia al 1 dándonos a entender que cada objeto de la clase “Model” puede estar asociado con uno o varios objetos de las clases Material y Figure. También se puede deducir que se tendrá un atributo y no una colección.

Otra sección importante a explicar son las relaciones entre “Model”, “PositionedModel”, “Coordinate” las cuales están relacionadas con Asociación,dándonos a entender que la clase “PositionedModel” puede tener un mismo modelo pero en diferente posición, encargándose de esto la clase “Coordinate”. Por último para cerrar este UML creamos una clase “Scene” donde se alojan

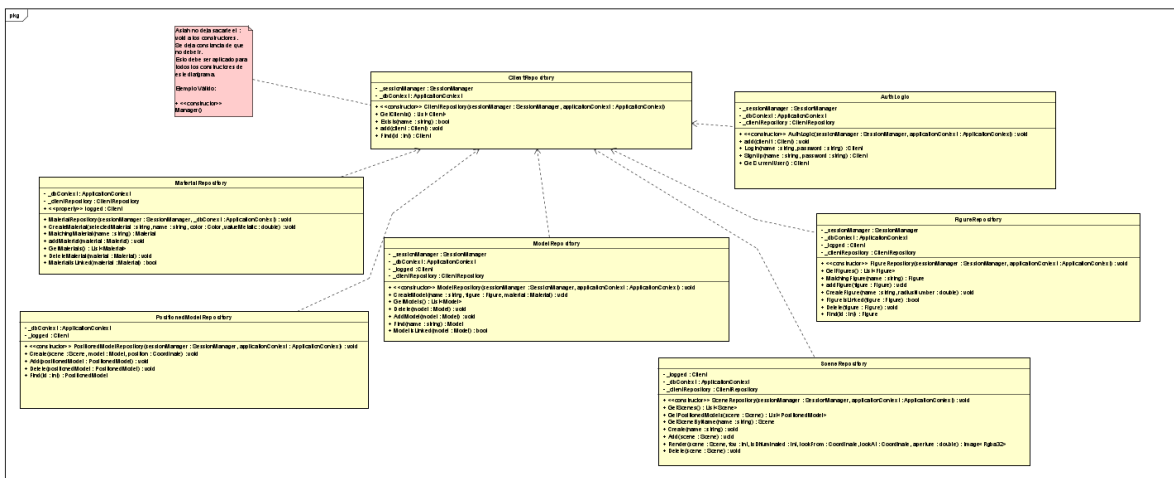
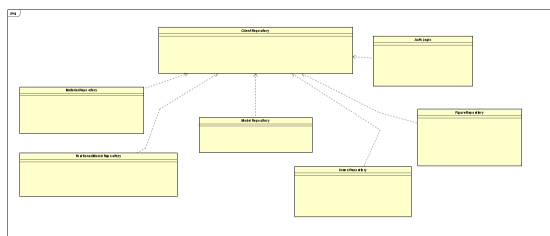
los diferentes tipos de modelos posicionados refiriendo al uso de una **Agregación** entre la clase nombrada anteriormente “Scene” con “Positioned Model”

Cambios importantes creados en este paquete fue la creación de la jerarquía (herencia) sobre la clase material “naciendo” de ella las clases hijas “MetalicMaterial” y “LambertianoMaterial”, se ondara sobre este tema en la [sección de Patrones de diseño](#).

Además se realizó un cambio sumamente importante en la clase Client dado que esta estaba sumamente acoplada, este tema ahondaremos en la [sección GRASP/ SOLID](#).

BusinessLogic:

(Se adjunta UML sin Atributos ni métodos como dice documentación).



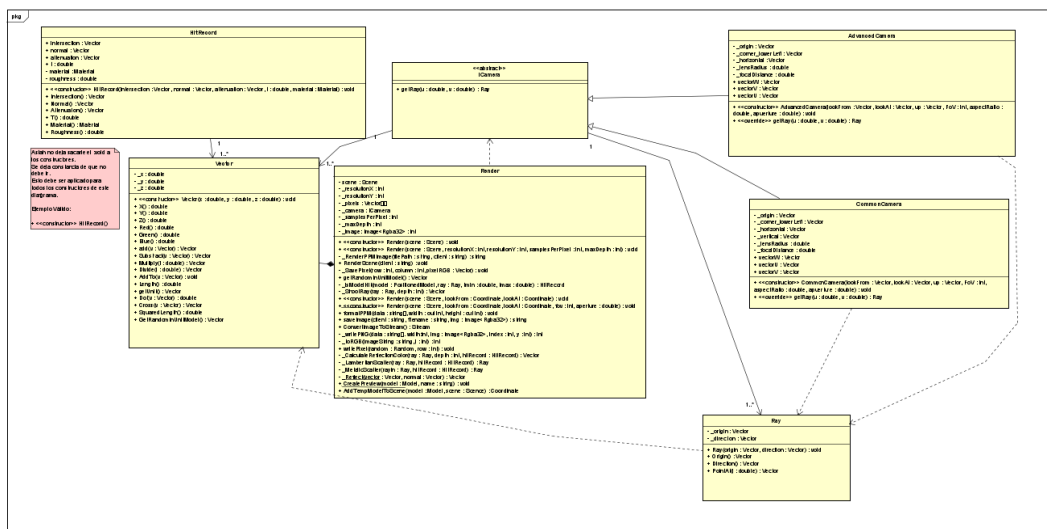
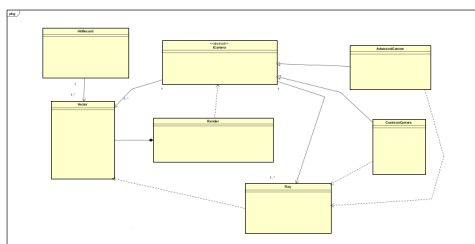
Se puede observar que este diagrama fue actualizado en la segunda entrega ya que se contaba anteriormente con un paquete “BussinessLogic” que contenía una clase “Manager” la cual estaba infringiendo un alto acoplamiento, por lo tanto se optó por dividirlo en clases como por ejemplo: las clases “MaterialRepository”, “FigureRepository” entre otras ayudandonos a quitar este problema.

Así mismo se puede observar una relación de dependencia entre las clases nombradas anteriormente con la clase “ClientRepository” dado que estas contienen un atributo del tipo “ClientRepository” haciendo que dependan de ella.

Ahondar en esta [sección](#) sobre porque se bajo el acoplamiento de esta clase.

Engine:

(Se adjunta UML sin Atributos ni métodos como dice documentación).



Podemos observar que existe una relación **Asociación** entre la clase “Camera” y la clase “Ray”, esto se da porque la clase Camera contiene un método que retorna una instancia de tipo “Ray”.

También la **Multiplicidad** nos está dando a entender que una instancia de Cámara puede crear múltiples instancias de “Ray” pero esto no es recíproco o bidireccional, ya que una instancia de Ray se asocia con solo una de la clase “Camera”.

Contamos también con una relación de **Dependencia** entre las clases “Camera” y “Render”, lo podemos notar observando el código de la clase “Render” ya que este cuenta con un atributo de tipo “Camera”, por lo tanto podemos decir que “Render” depende de la clase “Camera”.

Así mismo esto pasa entre las clases “Camera” y la clase “Vector” ya que Camera **utiliza** a Vector para representar las posiciones como se puede notar en este fragmento de código:

```
“private Vector _corner_lowerLef;  
private Vector _horizontal;  
private Vector _vertical;”
```

También notamos una dependencia entre las clases “HitRecord” y “Vector” ya que se puede observar como la clase “HitRecord” utiliza la clase Vector, haciendo uso de sus instancias en sus atributos.

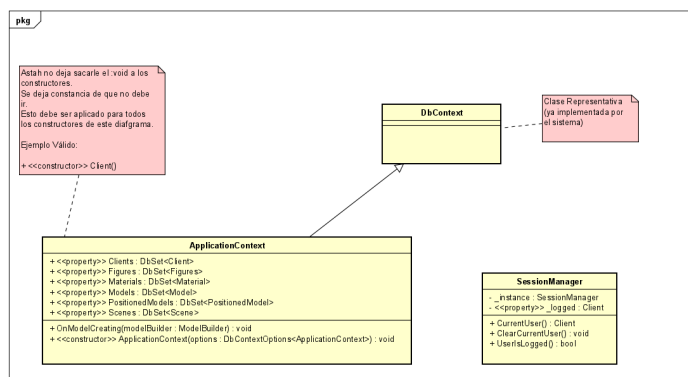
Por último podemos observar una **Composición** entre las clases “Vector” y “Render”, dado que al tener un atributo de tipo “Vector” en la clase “Render” y saber que va a estar compuesta por una o más instancias de “Vector” se formará un lazo de vida muy estrecho por lo tanto se optó por elegir esta relación en vez de una dependencia. “Render está compuesto por Vector, Render no viviría sin Vector”.

Podemos observar que se realizaron cambios nombrados y pedidos en la corrección de la primera entrega ya que se desacoplo algunos métodos desfragmentandolos en funciones más pequeñas para que estas solo **contengan únicamente una responsabilidad**.

Además se implementó una jerarquía de clases ya que se cuenta con dos tipos de cámaras, una llamada “AdvancedCamera” y “CommonCamera”, esto lo hacemos para no repetir código entre las distintas clases que generamos.

Los porqués de estas acciones estarán detalladamente en la [sección Grasp/Solid](#).

DataAccess:



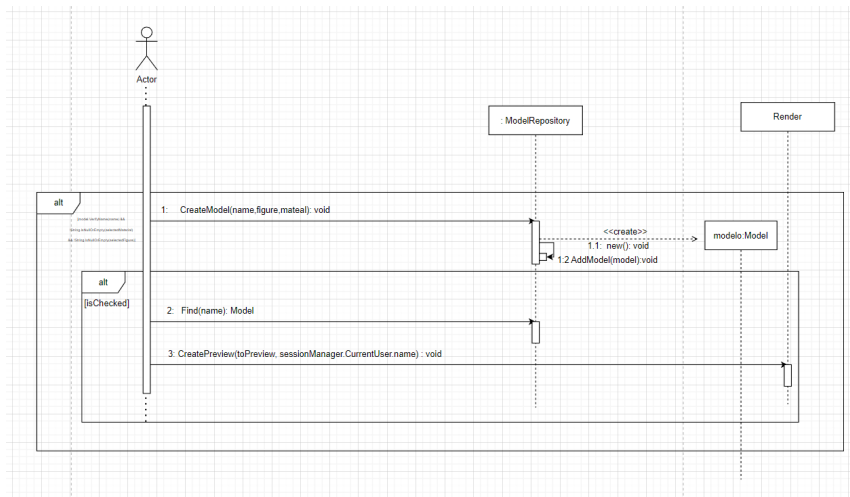
Se observa que se agregó un nuevo paquete conteniendo dos clases nuevas en esta segunda entrega que no se relacionan entre sí, pero son sumamente importantes a la hora de la creación y el uso de nuestra base de datos. En la sección de [Entity Framework](#) se hablará más de base de datos y porque se utilizó esta forma de almacenamiento de datos.

Diagrama de Interacción:

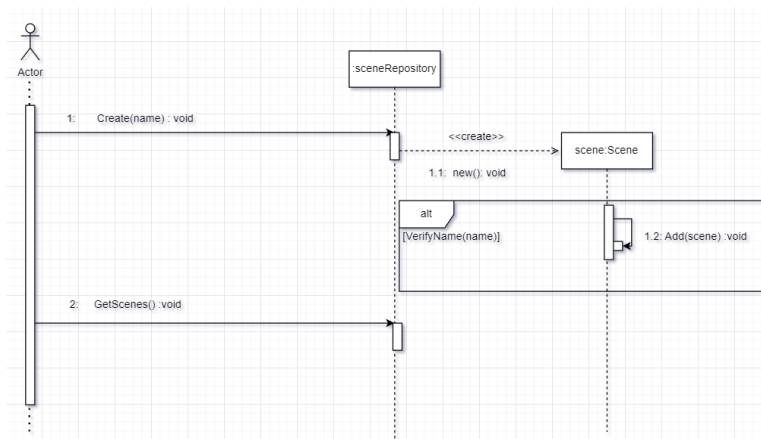
Se utilizaron diagramas de secuencia para poder representar la interacción entre los diferentes participantes en una duración de tiempo.

Se eligieron estas funcionalidades ya que son sumamente importantes a la hora de utilizar el sistema, ya que partimos de la interfaz de usuario y luego como podemos observar en cada uno de estos se introducen a clases importantes dándonos las funcionalidades pedidas en estas entregas.

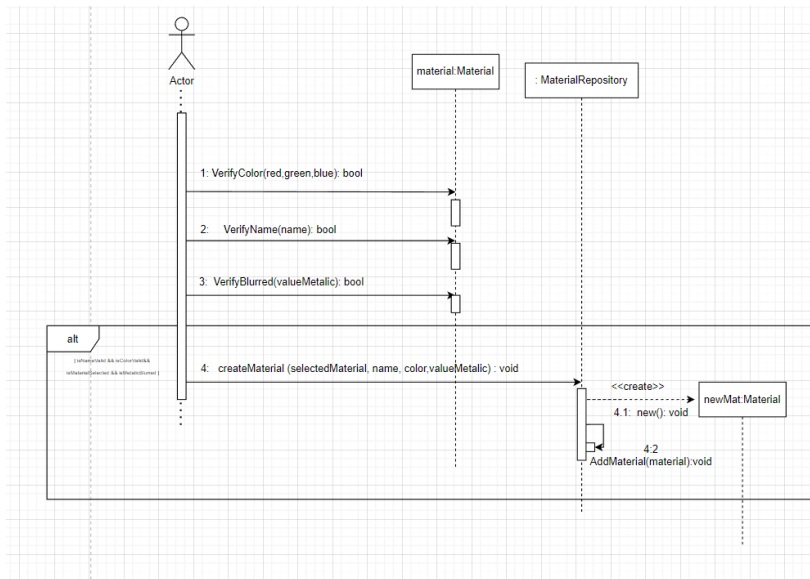
1)



2)



3)



Test-driven development (TDD):

Utilizamos TDD como **estilo de programación** para obtener un mayor beneficio a la hora de programar a futuro. Siempre teniendo en cuenta las **cuatro leyes** fundamentales que este define:

- 0• Nunca escribir código de producción si no se escribió el código de una prueba unitaria.
- 1• Escribir el código mínimo y suficiente de la prueba para que sea una prueba fallida.
- 2• Escribir solo el código mínimo y suficiente de producción para que la prueba pase.
- 3• Refactorizar constantemente tanto el código de prueba como el de producción!

Podemos contemplar el uso de estas reglas sobre nuestro código con ayuda de estas imágenes:

Feature: RF4, Usuario puede crear Figura. Clase Manager - Fase refactor PolTowers committed 3 weeks ago	4951e91	<>
Feature: RF4, Usuario puede crear Figura. Clase Manager - Fase Green PolTowers committed 3 weeks ago	7b18b31	<>
Feature: RF4, Usuario puede crear una figura. Clase Manager - Fase red PolTowers committed 3 weeks ago	3eb2f08	<>
Feature: RF2, Usuario puede registrarse. Clase Manager - Fase Refactor PolTowers committed 3 weeks ago	6ad1867	<>
Feature: RF2, Usuario puede registrarse. Clase Manager - Fase Green PolTowers committed 3 weeks ago	541d4b9	<>
Feature: RF2 Usuario puede registrarse. Clase Manager - Fase red PolTowers committed 3 weeks ago	ebc82fa	<>

Básicamente se fue haciendo un commit en cada etapa, se realizó en la **Fase Red** que representa el punto 1 anteriormente nombrado, luego se realizó un commit en la **Fase Green** representado por el

punto 2 y por último la etapa de **Refactor** donde podemos arreglar nuestro código usando **Clean Code**.

Hablaremos sobre el uso de las ramas que utilizamos para representar TDD; Estas fueron dos, **branch: feature-tdd** donde representamos puntualmente cada fase con sus commits correspondientes.

Y **branch: feature** donde se obvia el paso de hacer un commit por cada regla para agilizar la programación, de igual manera se respetaron las leyes nombradas anteriormente.

Cobertura:

Este tema va de la mano con Test-driven development donde nos mide la proporción de código utilizada por dichas pruebas.

Se llegó obtener un porcentaje por arriba de los 90% (Blocks) en cada proyecto, además de cubrir la cobertura de línea de código también superando el 90% (Lines) como se puede contemplar en la siguiente imagen:

Hierarchy	Covered (%Blocks)	Not Covered (%Blocks)	Covered (%Lines)	Partially Covered (%Lines)	Not Covered (%Lines)
ignac_LAPTOP-3R9PVFHT 2023-05-11 13_19_36.coverage	94,99%	5,01%	94,45%	0,65%	4,91%
engine.dll	89,94%	10,06%	89,67%	0,33%	10,00%
engine.test.dll	100,00%	0,00%	100,00%	0,00%	0,00%
domain.dll	93,50%	6,50%	90,87%	1,17%	7,96%
businesslogic.test.dll	98,93%	1,07%	98,92%	0,54%	0,54%
businesslogic.dll	93,71%	6,29%	91,67%	0,83%	7,50%
exceptions.dll	100,00%	0,00%	100,00%	0,00%	0,00%
domain.test.dll	99,22%	0,78%	99,13%	0,43%	0,43%

Se habló con el profesor y se decidió que no era necesario llegar al 90%(Lines) en el paquete Engine.

Esto nos demuestra el buen uso de la metodología TDD. Ya que en cada proyecto se realizaron pruebas exhaustivas en los posibles casos que podrían surgir de nuestras funcionalidades.

También esta herramienta que nos proporciona Visual Studio Code nos permite ver en el interior de cada proyecto, es decir en las clases que lo integran.

domain.test.dll	99,22%	0,78%	99,13%	0,43%	0,43%
Domain.test	99,22%	0,78%	99,13%	0,43%	0,43%
ClientTest	98,79%	1,21%	99,01%	0,00%	0,99%
FigureTest	100,00%	0,00%	100,00%	0,00%	0,00%
MaterialTest	100,00%	0,00%	100,00%	0,00%	0,00%
ModelTest	100,00%	0,00%	100,00%	0,00%	0,00%
PositionedModelTest	96,30%	3,70%	91,30%	8,70%	0,00%
SceneTest	100,00%	0,00%	100,00%	0,00%	0,00%

Hay que tener en cuenta que es muy difícil llegar al 100% ya que hay casos puntuales los cuales son difícil de contemplar.

Fundamentos de diseño de software:

Patrones GRASP/Principio SOLID:

Como se mencionó en la anterior entrega, se había creado una clase “Manager” contenida en el paquete “BusinessLogic” con algunos problemas desde el punto de vista de **Patrones GRASP**, es decir esta contenía un **alto acoplamiento** y una **baja cohesión** entre sus métodos y responsabilidades restringiendonos los cambios a futuro ya que un mínimo cambio en alguna entidad esta afectará a nuestra clase directamente.

Además se podía observar una baja cohesión ya que esta clase maneja distintas responsabilidades de diversas entidades.

Por lo tanto se optó por pensar una opción que sea más factible a la hora de trabajar, aquí entra el uso de **Principios Solid**.

Nos centramos específicamente en primeramente en la clase “Manager” ya que contenía estos tipos de errores mencionados anteriormente;

Aplicamos **Principio de Responsabilidad Única** comúnmente conocido como **SRP**, ya que dividimos manager en diferentes clases, cada una de estas con una responsabilidad única valga la redundancia sobre una única entidad, esto se puede observar en el UML dado que la clase se dividió en “ModelRepository”, “SceneRepository”,etc

También, se tuvo un problema similar en la entrega número uno, se nos corrigió el hecho de contener una clase “Client” altamente acoplada por lo tanto aplicamos el principio anteriormente nombrado **SRP** llevando responsabilidades a las diferentes clases contenidas en el paquete Business Logic.

Con esto, obtuvimos un aumento significativo en la cohesión y un bajo acoplamiento en cada una de las clases creadas además de bajar notablemente la cohesión.

Hablando un poco más sobre los patrones **GRASP** se puede notar el uso patrón **Experto** ya que utilizamos la propia información que conocemos sobre las clases, esto se observa simplemente en los métodos “isValid()” contenidos en varias de las clases del dominio que implementamos, otra forma de darnos cuenta del uso es la forma que corroboramos los duplicados de nuestras entidades, no se corroboró con un método en las clases del dominio ya que las propias clases no contienen esta información ya que es un nivel más macro, por lo tanto utilizaremos otros expertos para validar esto.

Patrones de diseño:

Otro cambio que se aplicó fue sobre la clase “Material”, no obstante hay que destacar que está no tenía errores, simplemente quisimos utilizar Patrones de Diseño para implementar lo dado en clase.

Utilizamos el **Patrón Factory method**, donde en la clase “Material” creamos un método comúnmente llamado “Método Fábrica” que hace referencia a nuestro “createMaterial (...)” donde se crearán las instancias de los diferentes materiales a través de la implementación de un switch, con esto se podrá insertar nuevos tipos de materiales rápidamente, ya que bastaría crear una clase hija de Material y agregarlo en las opciones del switch.

Algo a tener en cuenta es que sabemos por definición que el principio de **Open-Closed** nos dice que está abierto a la extensión y cerrado a la modificación. A futuro se estaría modificando el código por lo tanto estamos incumpliendo lo dicho anteriormente.

Para justificar lo hecho, podemos decir que estamos centralizando/encapsulando la creaciones de entidades en un solo lugar llamado método fábrica obteniendo una forma sencilla y rápida de crear nuevas instancias según la selección del cliente, además que al pedirnos una pequeña cantidad a implementar de nuevos tipos de materiales esta forma nos brinda una rapidez a la hora de codificar.

A su vez hay que tener en cuenta que **Solid** son principios y no leyes, es decir podemos utilizarlos para obtener una cierta “ganancia” en un sector y por ende perder en otro.

Formato del patrón utilizado: (Puntos importantes sobre la escritura previa)

- Nombre del Patron: Factory Method
- Cuando se aplica: Cuando una clase no puede anticipar el tipo de objeto que va a crear.
- Intención: Podemos crear nuevas clases hijas sin modificar una cantidad importante de código, además nos da una forma de encapsular las creaciones de nuestros objetos.
- Participantes: Clase “Material”, MetalicMateria y LambertianoMaterial.

Siguiendo con los Patrones de diseño implementados en este obligatorio podemos nombrar un patrón que se realizó para la primera entrega, pero en el curso no se hondo hasta ese entonces, este es el **Patron Singleton**, el mismo se utilizó en la clase Program.cs en la siguiente sentencia: “builder.Services.AddSingleton<SessionManager>();”

” dándonos a entender que se creara una sola instancia de la clase durante toda la aplicación además de poder utilizarla desde cualquier parte de código donde fuese referenciada utilizando `@using Interface.DataAccess;` .

Esto nos da un mayor beneficio a la hora de llevar en práctica el tener “código limpio” evitando tener una cantidad significativa de instancias creadas haciendo referencia a lo mismo incumpliendo las reglas de clean code.

Formato del patrón utilizado: (Puntos importantes sobre la escritura previa)

- Nombre del Patron: Singleton
- Cuando se aplica: Cuando necesitamos tener solamente una instancia de la clase, en este caso de la clase “SessionManager” donde otras clases van a tener que utilizarlas.
- Intención: Nos asegura tener una sola instancia de esta clase y proporcionar acceso global a la misma.
- Participantes: “SessionManager” y todas las que utilizan su `@using`.

Mapecto de Objetos Relacionales:

Entity Framework:

Para la segunda entrega de este obligatorio, uno de los requerimientos más importantes fue la implementación de una base de datos relacional. Para esto se utilizó un ORM precisamente “Entity framework” donde nos permitirá trabajar con una representación de base de datos virtual de objetos, el cual luego de aceptar una serie de “transacciones” se transformaran a tablas en una base de datos relacionales.

EagerLoading/Lazy Loading:

Para cargar los datos se utilizó el enfoque **Eager Loading**, como podemos observar por ejemplo en la clase “MaterialRepository.cs”

```
public List<Material> GetMaterials()
{
    return _dbContext.materials.Where(m => m.client.Id == logged.Id).ToList();
}
```

Estamos cargando todos materiales asociados al lugar donde se invoca esta función por ejemplo en la clase “MaterialList.razor”:

```

@foreach (var material in materialRepository.GetMaterials())
{
    <tr style="width: 100px; height: 100px; ">

        @if (material is MetallicMaterial)
        {
            var metallicMaterial = (MetalicMaterial)material;
            <td>@metallicMaterial.name</td>
        }
        else
        {
            var LambertianMaterial = (LambertianoMaterial)material;
            <td>@LambertianMaterial.name</td>
        }
    }
}

```

Se observa que se nos devolverá una List<Material> **cargando todos los materiales de una sola vez** a diferencia de Lazy Loading .

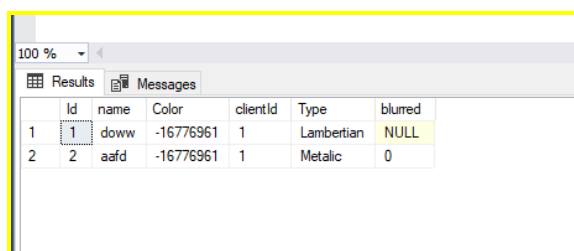
También hay que tener en cuenta que se usa **Lazy Loading**, en las partes de código que no se trabaja con algoritmos similar a lo anteriormente especificado, ya que se encuentra por defecto en visual Studio.

Mapecto de herencias:

Como se pudo observar en el respectivo UML tenemos una herencia en nuestro código, esta se da sobre las clases hijas “LambertianoMaterial” e “MetalicMaterial” siendo la clase padre “Material”. A la hora de utilizar Entity Framework optamos por usar **TPH (Table per Hierarchy)**, esta mapea todos nuestros tipos de “hijos” en la tabla Material creando y mostrando una columna donde nos da a conocer el tipo de este objeto, a la misma se le llama comúnmente columna Discriminadora.

Optamos por utilizar esta forma de mapeo ya que al tener una pequeña cantidad de clases hijas en nuestra jerarquía nos evitará crear tablas asociadas adicionales las cuales nos dificultará la comprensión de nuestra base de datos.

Además no menos importante, es que esta forma ya viene por defecto en Visual Studio, solamente se necesita agregar pocas líneas lo cual para estos requerimientos pedidos fue una forma rápida y sencilla de implementar.



Id	name	Color	clientId	Type	blurred
1	doww	-16776961	1	Lambertian	NULL
2	aafd	-16776961	1	Metalic	0

Mecanismo general del sistema:

Como resumen hablaremos un poco más sobre los mecanismos y las decisiones tomadas en el mismo.

Primeramente la interfaz de usuario tiene un vínculo sumamente importante con el dominio ya que estos son los encargados de comenzar con el funcionamiento del sistema pudiendo así implementar los requerimientos que nos piden en la letra, este vínculo se hace a través de los componentes que nos dejara implementar Blazor, los mismos están asociados a un controlador de eventos los cuales son los encargados de “llevar” la información a las diferentes partes de nuestro sistema.

Como se pudo ver tenemos una sección (paquete) donde se contienen las excepciones, estas van a jugar un rol importante a la hora de corroborar y capturar situaciones no deseadas por el sistema, otorgándonos así una forma rápida de poder mostrarle al usuario un mensaje de error o evitar que se “caiga” nuestro sistema.

A su vez en esta segunda instancia al implementar una base de datos pudimos almacenar persistentemente los datos dándonos una ventaja significativa con respecto a la primera entrega ya que vamos a poder guardar nuestros datos sin tener que volver a colocarlos luego de cerrar nuestra página.

A destacar esto es un leve repaso por nuestro sistema, los detalles de cada punto se pueden observar en cada sección de nuestra documentación.

Errores corregidos del Obligatorio 1:

Luego de obtener el feedback de los profesores sobre el obligatorio 1 intentamos arreglar los errores nombrados por los mismos.

Puntos:

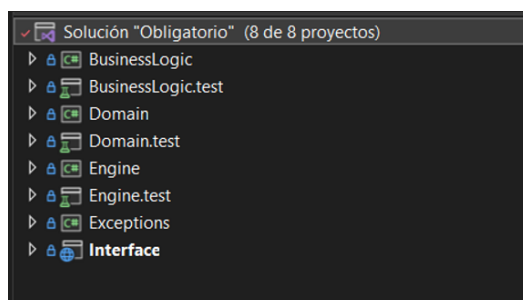
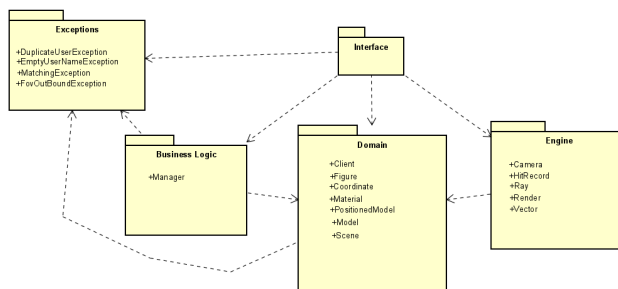
- Quitamos todos los comentarios irrelevantes en el código ([Ahondar en sección](#)).
- Desacoplamos la clase “Cliente” como la clase “Manager”. ([ahondar en su sección](#)).
- Se usaron los Principios/Patrones SOLID y GRASP ([ahondar en sus secciones](#)).
- Uso de Patrones de diseño ([Ahondar en su sección](#)).
- Utilizamos refactorización en diferentes métodos de clases distintas, por ejemplo en el método “Render” ([Ahondar en sección](#)).
- Uso de Gitflow ([Ahondar en sección](#)).

Algo a destacar es que no se pudo implementar los casos de prueba de la base de datos por falta de tiempo y conocimiento.

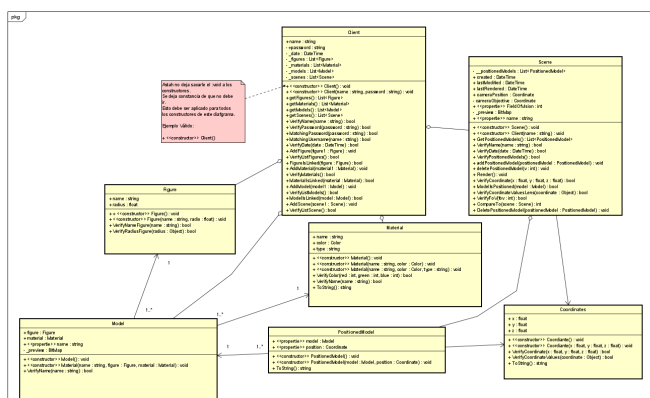
Anexo:

Diagramas Entrega 1:

Diagrama de Paquetes: (Anterior)



Domain: (Anterior)



BusinessLogic:(Anterior)

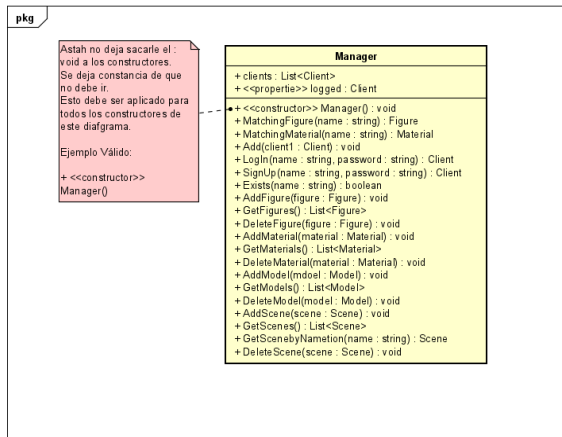
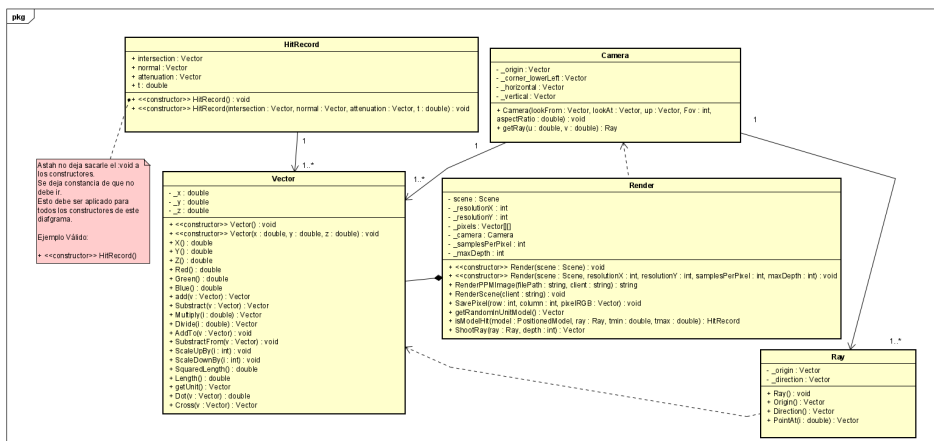


Diagrama Engine: (Anterior)



Reporte Evidencia: (Anterior)

El equipo decidió realizar un video explicativo de como funciona el sistema, además de dejar constancia que no contiene ninguna falla.

Link Youtube:

https://youtu.be/MoRks_LTIIM

Respuesta al Archivo Errores Comunes en los Obligatorios: (Anterior)

Pregunta:

¿Hay algún mecanismo de extensión ya pensado para soportar nuevos tipos en el futuro? Si hay, documentarlo. Si no hay, explicar por qué.

Respuesta:

Sí, realizamos el código pensando en tener una buena cohesión entre los componentes de nuestro proyecto para que sea más fácil nuevas implementaciones y/o cambios a futuro.

Además nuestras clases están diseñadas para soportar nuevos tipos de datos en un futuro por ejemplo la creación de un nuevo Tipo de Material.