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

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, que en nuestro caso fueron dos: **featureTDD** y **feature**, en las cuales se trabajó en ellas para implementar los diferentes requerimientos pedidos en la letra, se ahondará más adelante sobre ellas.
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.

Algo a tener en cuenta para los próximos obligatorios es enfocarnos más en la metodología gitflow y crear por cada Requerimiento Funcional una rama y posteriormente realizar un merge a la rama develop.

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.

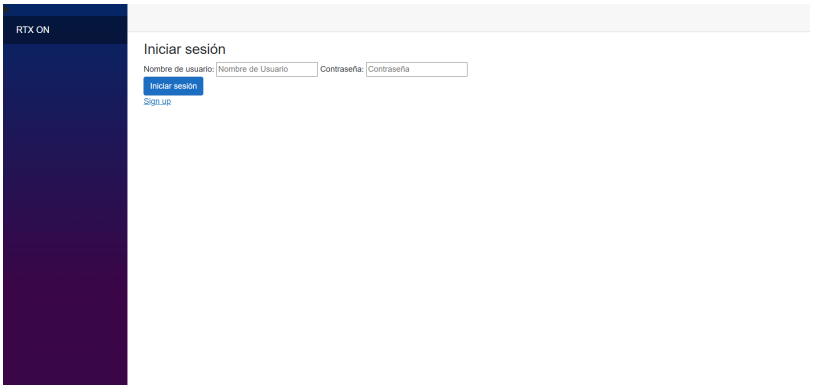
Descripción del sistema

Características de la página:

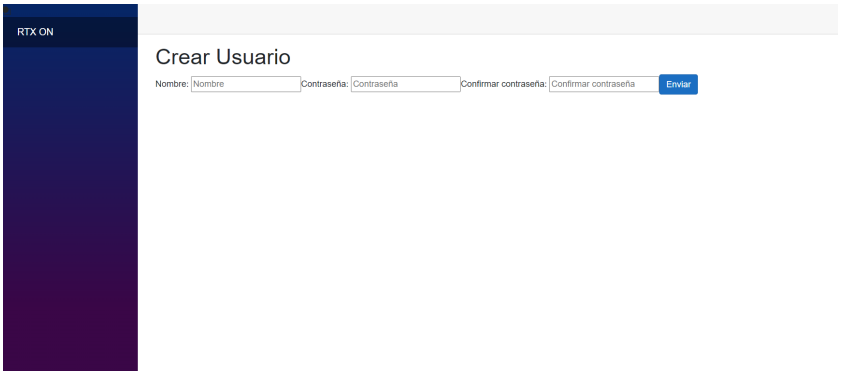
En la letra dada se nos pide que realicemos una página donde se pueda crear imágenes 3D utilizando la técnica de renderizado llamada Ray Tracing.

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.

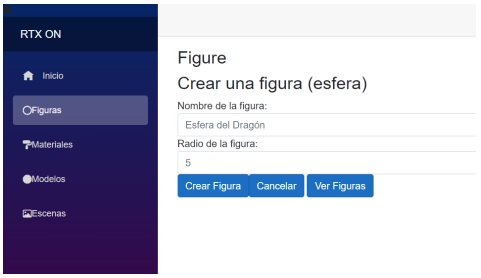
Vista de ingreso:



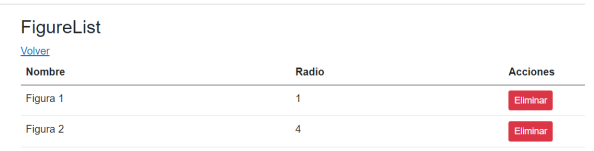
Vista para generar usuario:



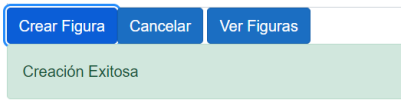
Luego de su ingreso exitoso, se lo redirigirá a una sección donde podrá crear una figura, agregarle su nombre y un radio específico. También en esta división se puede observar 3 botones los cuales sirven para “Crear” la figura anteriormente nombrada, otro para “cancelar” que lo redirigirá a una sección donde podrá “Cerrar sesión” o “comenzar nuevamente”.



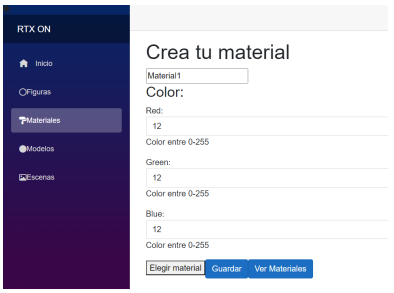
Por último se cuenta con un botón “Ver Figuras” lo cual nos lista las figuras con sus características anteriormente colocadas por el usuario,
Además esta sección contiene un botón de “eliminar” para quitar la figura a elección del cliente.



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.

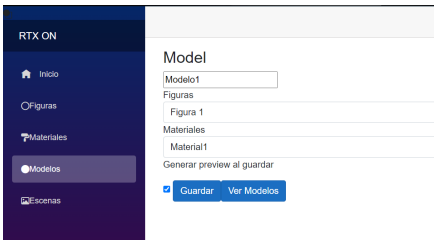


También se puede observar al lado izquierdo de la pantalla un nav que nos permitirá navegar entre las diferentes secciones, que nombraremos más adelante.



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 (en este Obligatorio tendremos solamente uno llamado Lambertiano).
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.

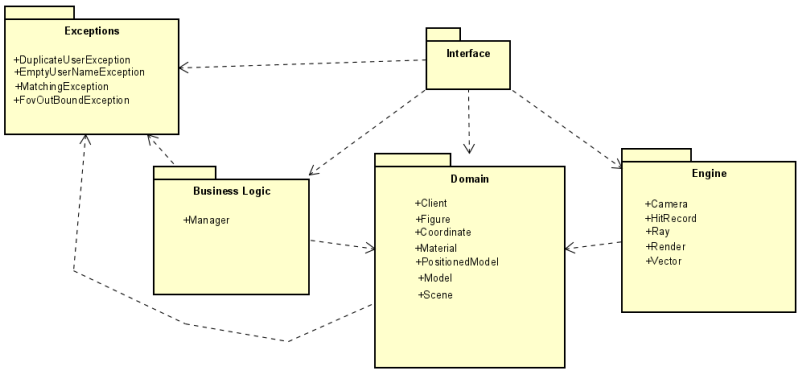
Por medio de un hipervínculo se podrá acceder a una sección que nos permitirá personalizar la escena, mediante la creación de modelos posicionados pasándole por medio de un **Input** valores de coordenadas.

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.

Creación de diagramas:

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

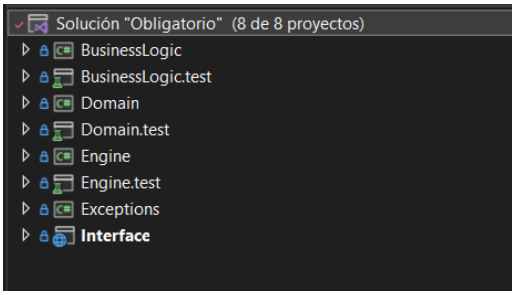
Diagrama de Paquetes:



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.

En nuestro proyecto nos encontramos con 8 paquetes, los cuales los podemos ver en la siguiente foto:



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 **BussinessLogic** 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.

También vamos a tener tres paquetes con sus terminaciones **.test** los cuales van a contener nuestras pruebas tdd de cada uno de los paquetes nombrados anteriormente más el paquete Engine.test.

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.

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.

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

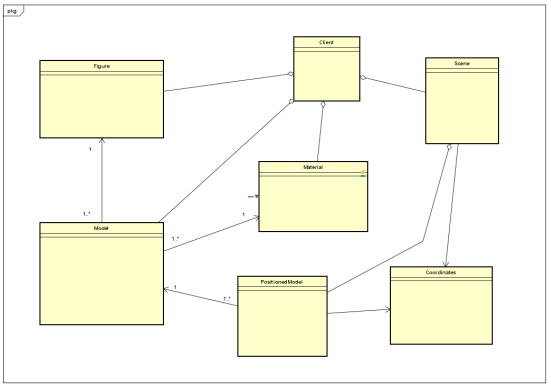
También, notamos que la relación entre paquetes es de **Dependencia**, deja claro que un paquete depende del otro, dándonos a entender que si se efectúa un cambio en un paquete este puede alterar el comportamiento del otro.
Por ejemplo la relación de dependencia que tenemos del paquete BusinessLogic con Domain se da ya que en la clase Manager perteneciente al paquete BusinessLogic contiene objetos Client el cual es perteneciente a Domain.
Así mismo podemos notar la relación de dependencia entre Domain y Engine partiendo desde la clase “Render”, creando un vínculo de dependencia con la clase “Scene”.

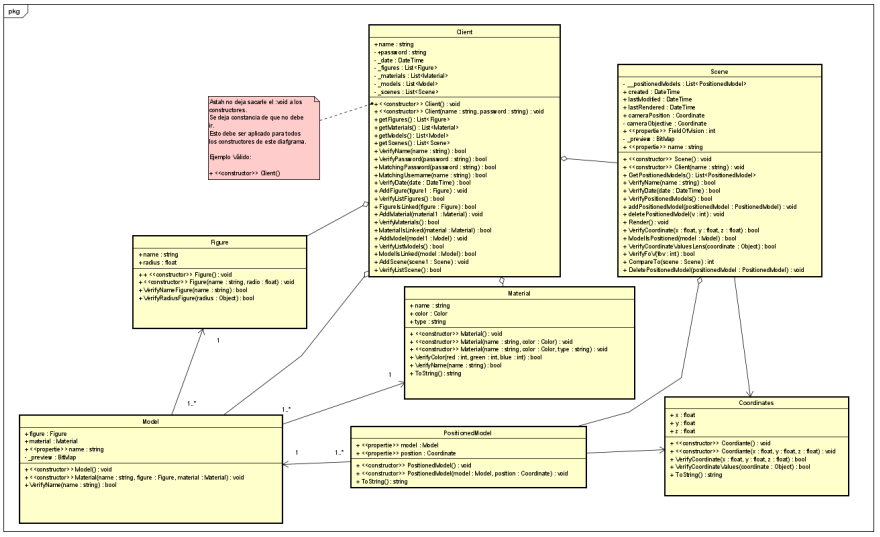
Por último, no menos importante podemos notar la dependencia que se realiza hacia el paquete Exceptions con los paquetes Interface y BusinessLogic, esto se da ya que se utilizan try and catch en varias clases de los paquetes anteriormente nombrados.

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.

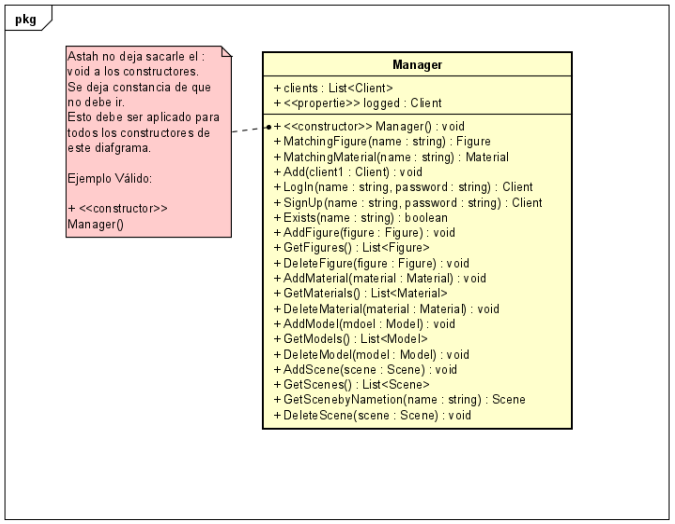
Se utilizó la relación **Agregación** en lugar de la **Composición** para vincular la clase “Cliente” con “Figure”, “Scene”, “Material” y “Model” ya que al momento de crear un cliente este no necesariamente debe tener una figura, escena ,material o un modelo, sabemos que el uso de composición nos estaría 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.

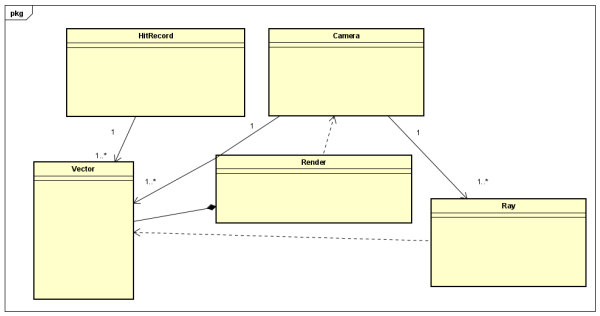
Otra sección importante a explicar son las relaciones entre “Model”, “PositionedModel”, “Coordinate” las cuales están relacionadas con Asociación,dándonos a entender que la clase “PositionedModel” puede tener un mismo modelo pero en diferente posición, encargándose de esto la clase “Coordinate”. Por 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”

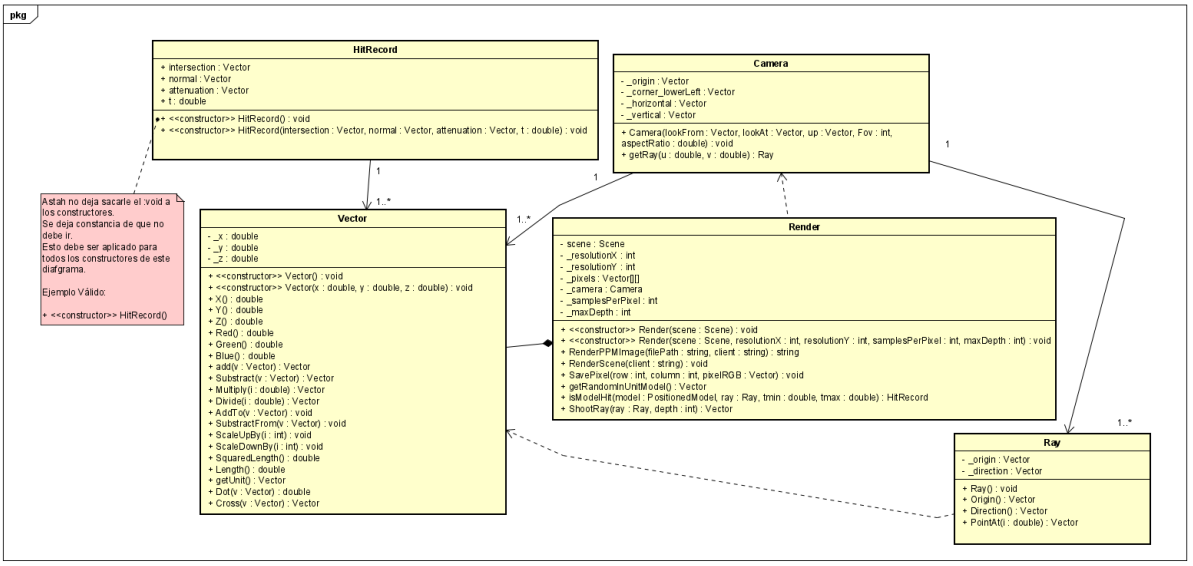
BusinessLogic::



Implementar a futuro: Se separaran los métodos de Manager para obtener un bajo acoplamiento.

Engine:





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”.

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:

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**.

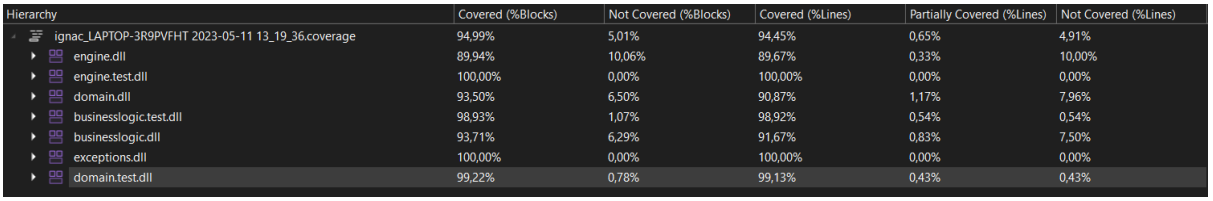
Hablaremos sobre el uso de las ramas que utilizamos para representar TDD;
Estas fueron dos, **branch: feature-tdd** donde representamos puntualmente cada fase con sus commits correspondientes.

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 linea de codigo tambien superando el 90% (Lines) como se puede contemplar en la siguiente imagen:

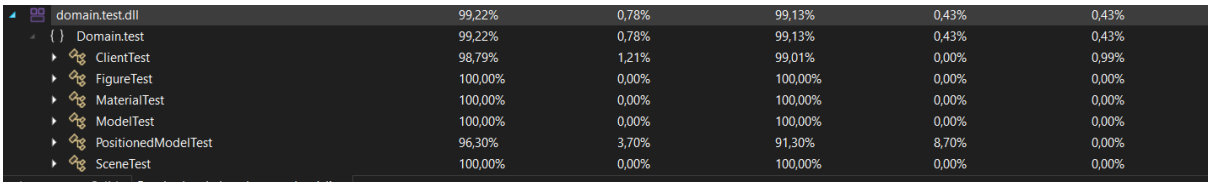


Hierarchy	Covered (%Blocks)	Not Covered (%Blocks)	Covered (%Lines)	Partially Covered (%Lines)	Not Covered (%Lines)
ignac_LAPTOP-3R9PVFHT 2023-05-11 13_19_36.coverage	94,99%	5,01%	94,45%	0,65%	4,91%
engine.dll	89,94%	10,06%	89,67%	0,33%	10,00%
engine.test.dll	100,00%	0,00%	100,00%	0,00%	0,00%
domain.dll	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.

Reporte Evidencia:

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_LTIlM

Respuesta al Archivo Errores Comunes en los Obligatorios:

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.