Universidad ORT Uruguay
Facultad de Ingeniería

Obligatorio de Diseño de Aplicaciones 1

Autores: Gabriel Guerra, Leandro Olmedo

2023

Índice

Descripción general del sistema

El propósito del sistema es generar escenas 3D renderizadas mediante Ray Tracing, utilizando diferentes figuras, modelos y materiales para los modelos. Para lograr dicho propósito, el sistema proveerá al cliente las herramientas e insumos necesarios para crear dichas escenas de forma sencilla y eficiente.

Decisiones centrales del diseño

Premisa principal

A la hora de diseñarlo el principal objetivo en el proyecto es crear un producto mantenible, se busca crear un producto fácilmente expandible y modificable a largo plazo.
De tener éxito en dicha premisa, la consecuencia será un producto en el que la productividad del equipo de desarrollo para mantener el software no se verá afectada a lo largo del tiempo, o por lo menos se minimizará la posibilidad de que surjan problemas que afecten a dicha productividad.

Cómo alcanzar la premisa

Para cumplir con la premisa se ha optado por utilizar programación orientada a objetos, desarrollo orientado a pruebas y las heurísticas proporcionadas por “Clean Code”.

Separación lógica

A partir de estos principios se tomó la decisión de separar lógicamente el software en diferentes secciones según los roles que cumplen las distintas partes del código. Para realizar dicha división se ha hecho uso de diferentes namespaces y clases, encapsulando el código y separando las responsabilidades en conjuntos más pequeños.
Mediante la separación de responsabilidades y la abstracción del sistema se busca hacer que el sistema se comporte de forma clara, coherente y con complejidad reducida; adicionalmente en caso de errores se logra que estos al ocurrir afecten a una parte mucho menor del sistema y que estos sean encontrables con más facilidad.

Programación orientada a objetos

Utilizar programación orientada a objetos hace más fácil la tarea de dividir las responsabilidades del programa en secciones lógicas y provee herramientas que permiten la reutilización de código. Esto es gracias al encapsulamiento en clases aisladas entre sí, el polimorfismo y otras varias posibilidades que ofrece hacer el uso de clases y objetos.

Test Driven Development

El uso de TDD (Test Driven Development) apoya el objetivo de mantenibilidad. El uso de TDD nos permite crear tests con mucha frecuencia y con buena cobertura, dichos tests permiten identificar rápidamente funciones o características que han dejado de funcionar por un motivo u otro, este beneficio incrementa a medida de que el sistema va creciendo en características, funcionalidades, dependencias y cantidad de código.
En un código cada vez más grande, es cada vez más difícil encontrar los problemas, TDD ayuda a reducir los problemas y a llegar a un sistema más robusto.

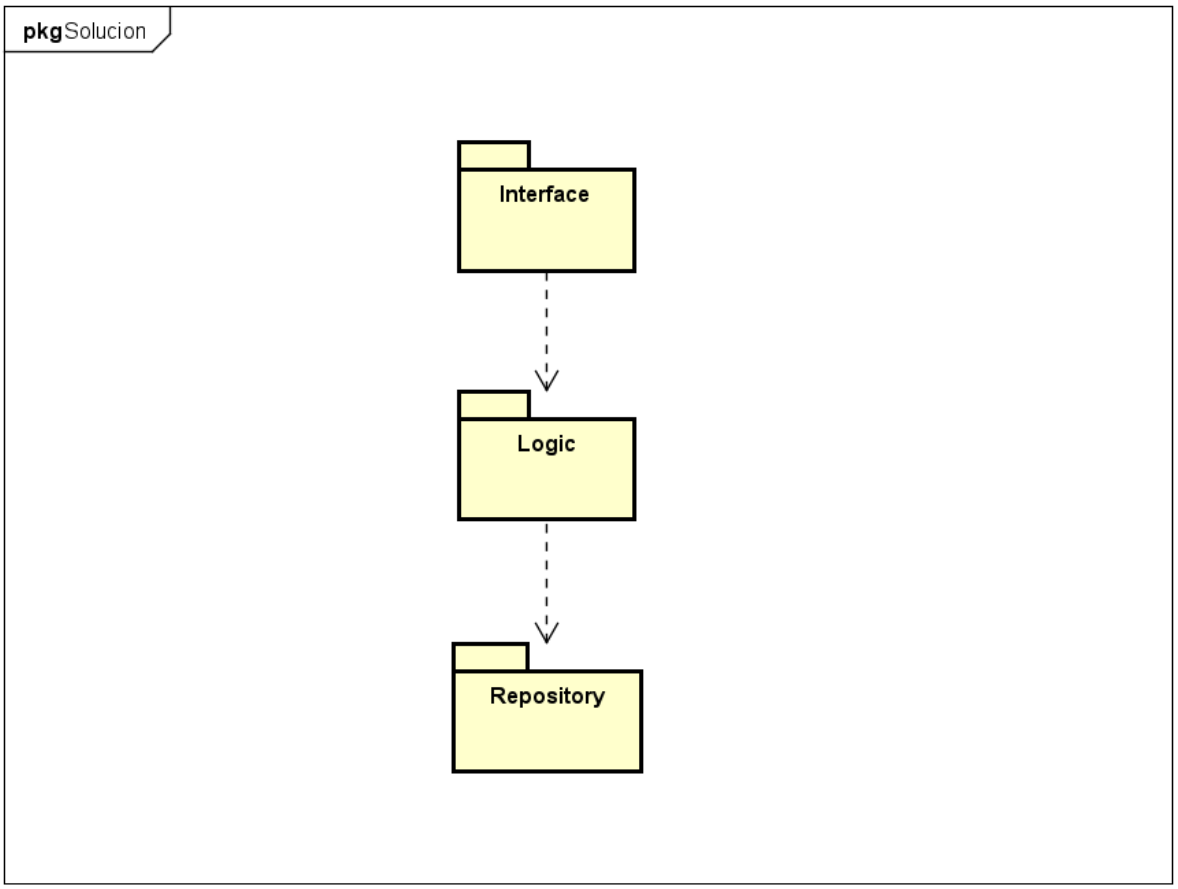
Clean Code y refactoreos

Además como ya se mencionó, durante la implementación, y a medida de lo posible, se aplican técnicas de “Clean Code” y “refactoreo” de código. Esto es con el objetivo de incorporar buenas prácticas en la programación, volviendo el código más legible, fácil de entender y fácil de modificar sin inconvenientes mayores.

Estructura de paquetes y namespaces

Interfaz, lógica y datos

A la hora de dividir lógicamente el sistema se decidió abstraerlo en tres capas diferentes, cada una con su propio rol: Interfaz, Lógica y Datos.



El usuario interactúa directamente con la interfaz y sólo con la interfaz.
Cuando la interfaz necesita realizar operaciones que involucren la lógica de negocio ésta se comunica con la "lógica" del sistema.
Cuando la lógica necesita almacenar, recuperar o eliminar datos, ésta llama a la parte de datos para que lo haga.

Interfaz: Medio por el cual el usuario interactúa con el sistema.

Lógica: Contiene la lógica principal del sistema, realiza controles, realiza operaciones y da órdenes a la capa de datos.

Datos: A este namespace lo llamamos "Repositorio" por su función. Tiene la función de almacenar, consultar, borrar datos y proveer datos cuando se lo piden.

Restricciones a la interacción entre capas

Nótese que la interfaz no interactúa directamente con la capa de datos, sino que usa a la lógica como intermediario. Evitar dicha interacción hace que se reduzca la cantidad de dependencia entre clases y hace que haya menos dudas sobre quién se encarga de validar, controlar o manipular los datos.

Se evitan dependencias circulares

Al realizar el diseño se evitó hacer dependencias circulares. Por ejemplo: Si bien Lógica depende del repositorio y hace llamadas de sus métodos, el repositorio no depende de Lógica para funcionar.
Evitar dependencias mutuas ayuda a evitar errores en cadena cuando una de las dos clases se rompe por algún motivo. Además ayuda a mostrar con más claridad cómo se comporta el programa, que dos clases llamen a métodos de la otra mutuamente añade una complejidad extra de la que estar atento, lo cual no es deseable.

Subdivisiones de interfaz, lógica y datos

La interfaz, la lógica y los datos a su vez se dividen en otro nivel de abstracción.

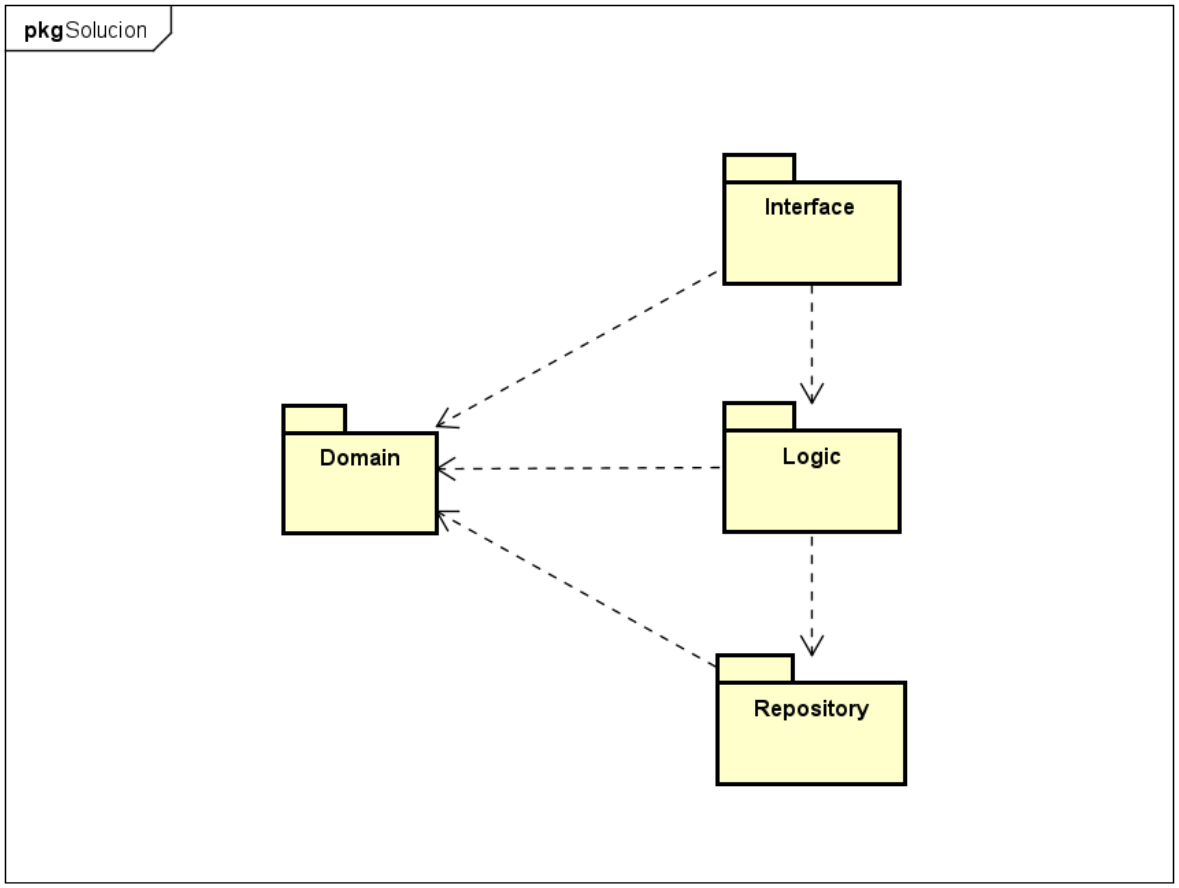
En la interfaz las ventanas y los elementos de la misma se distribuyen en un montón de clases, la gran mayoría auto-generadas por Windows Forms.

La lógica se divide en varias sub-lógicas, cada una encargada de un tema distinto. Al haber en el sistema figuras, materiales, modelos, escenas, etc, serían demasiadas responsabilidades si fueran asignadas todas a una lógica sola, por lo que se implementan varias, cada una se encarga de una de esas responsabilidades.

La capa de datos se divide en varios “repositorios”, cada uno almacena y manipula un tipo específico de datos como pueden ser figuras, clientes, materiales o modelos. La lógica del respectivo tipo hace uso de los métodos del repositorio que corresponde.

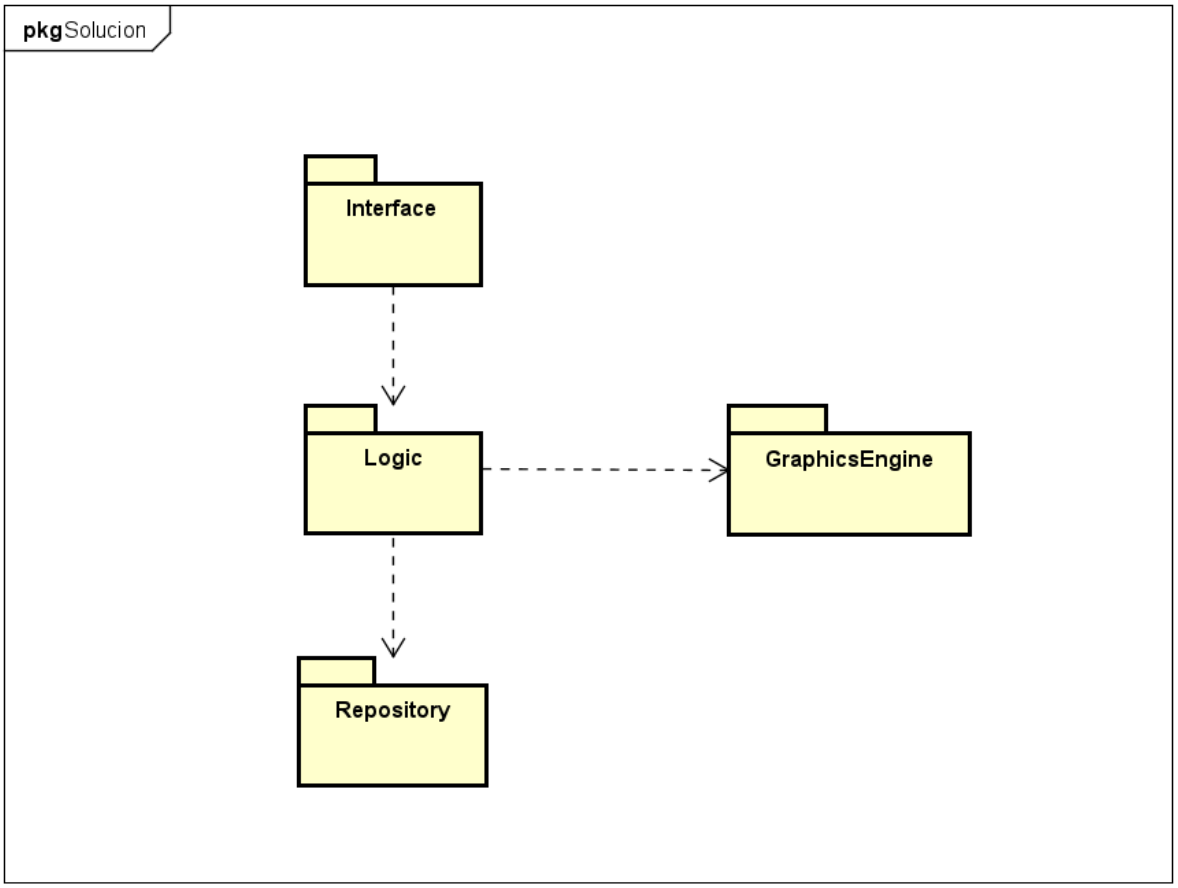
Namespace de Dominio

El dominio es usado por las tres capas de Interfaz, Lógica y Datos. Ahí es donde se definen las clases de los objetos “atómicos” como figuras, clientes, materiales, modelos, etc.
El propósito de estos objetos es existir y tener información dentro de ellos.



Motor gráfico

Las funcionalidades del motor gráfico están contenidas dentro de su propia lógica, encapsulada y aislada de las otras clases. En este namespace está todo lo referente a la lógica usada por el Ray Tracing, así como todos los cálculos e insumos que utiliza.

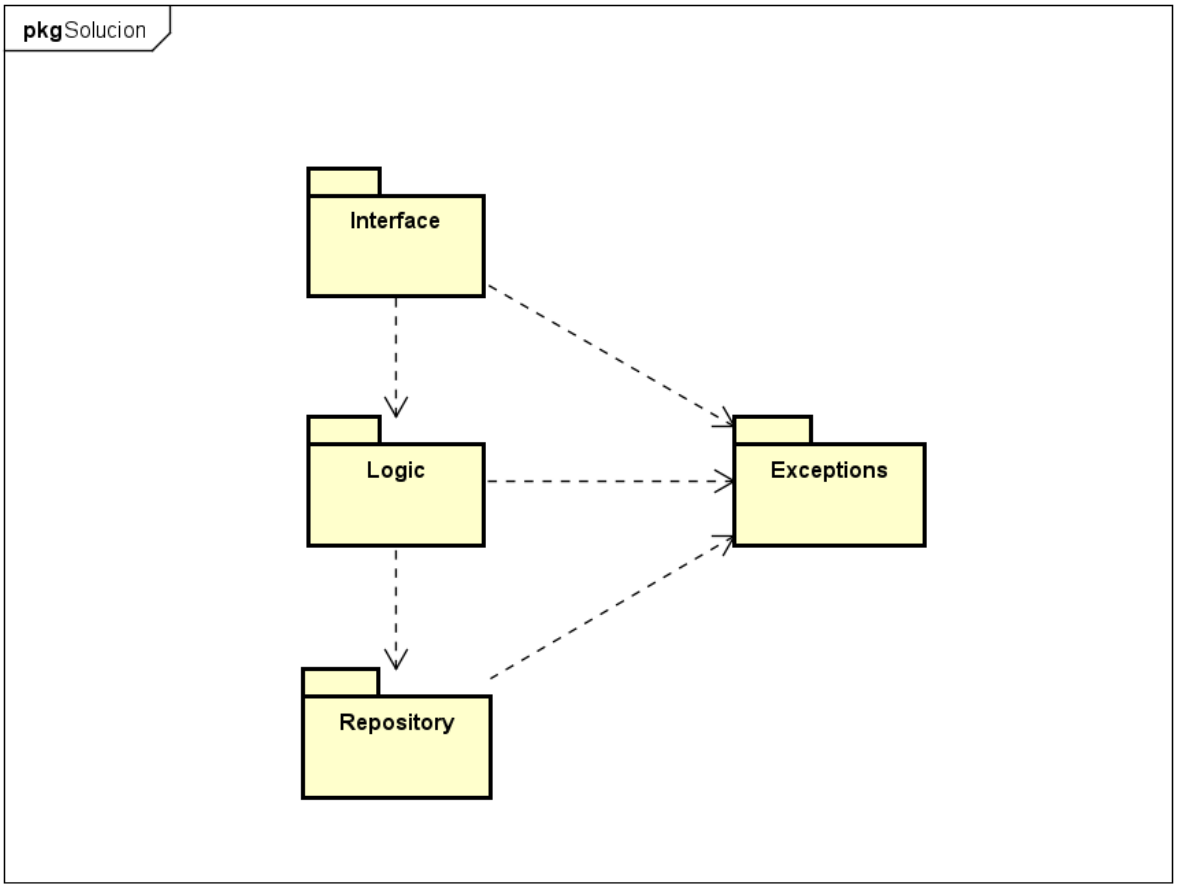


La capa lógica pide al motor gráfico que haga renderizados y operaciones por ella.

En la versión actual las partes del motor gráfico están distribuidas organizadamente entre los namespaces de dominio y lógica, pero cuando tengamos más tiempo pretendemos acomodarlo en un namespace propio.

Namespace de excepciones

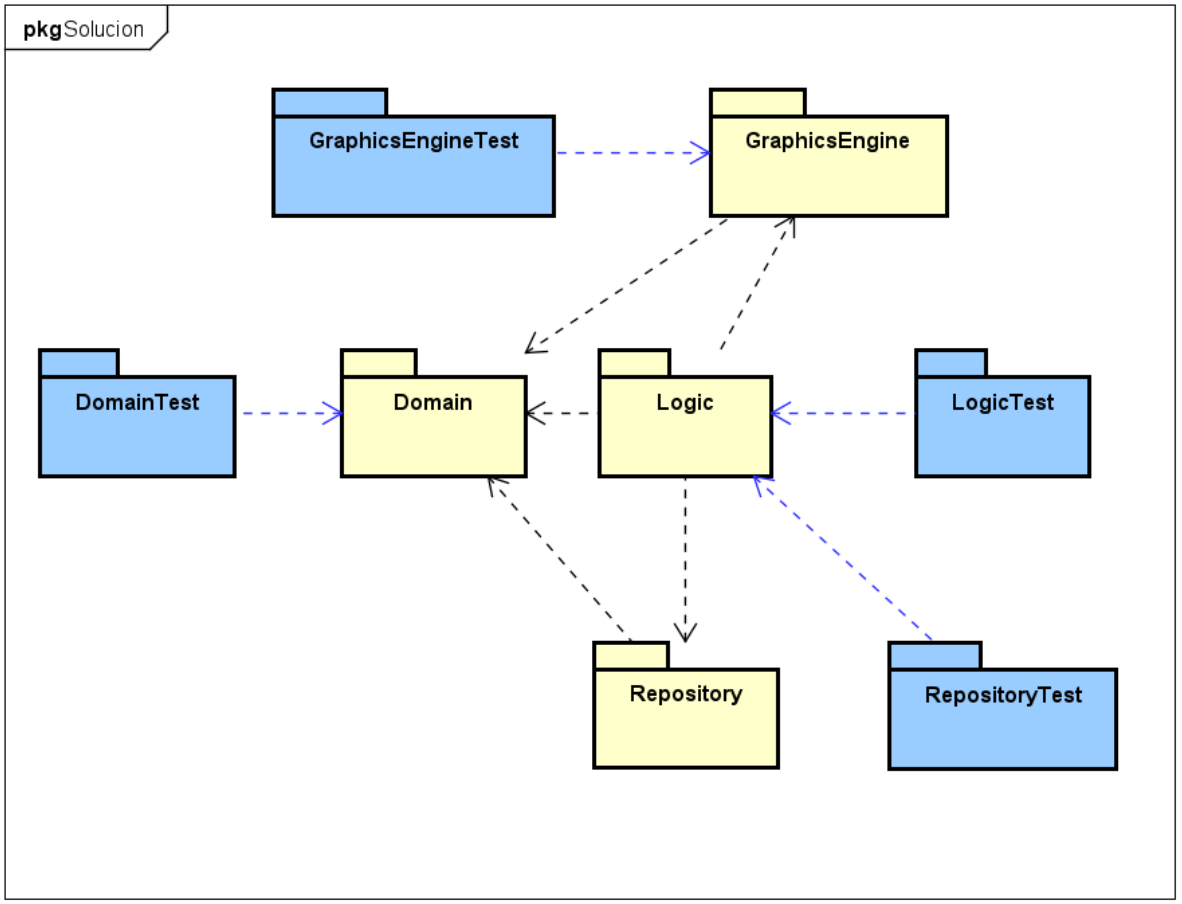
Se ha creado un namespace en el que se declaran las excepciones personalizadas para el sistema, esto permite organizadamente crear nuevas excepciones para que el sistema use y así identificar mejor algunos tipos de error.
Por ahora esa es su única función, declarar algunos tipos de excepciones.



Namespaces de pruebas

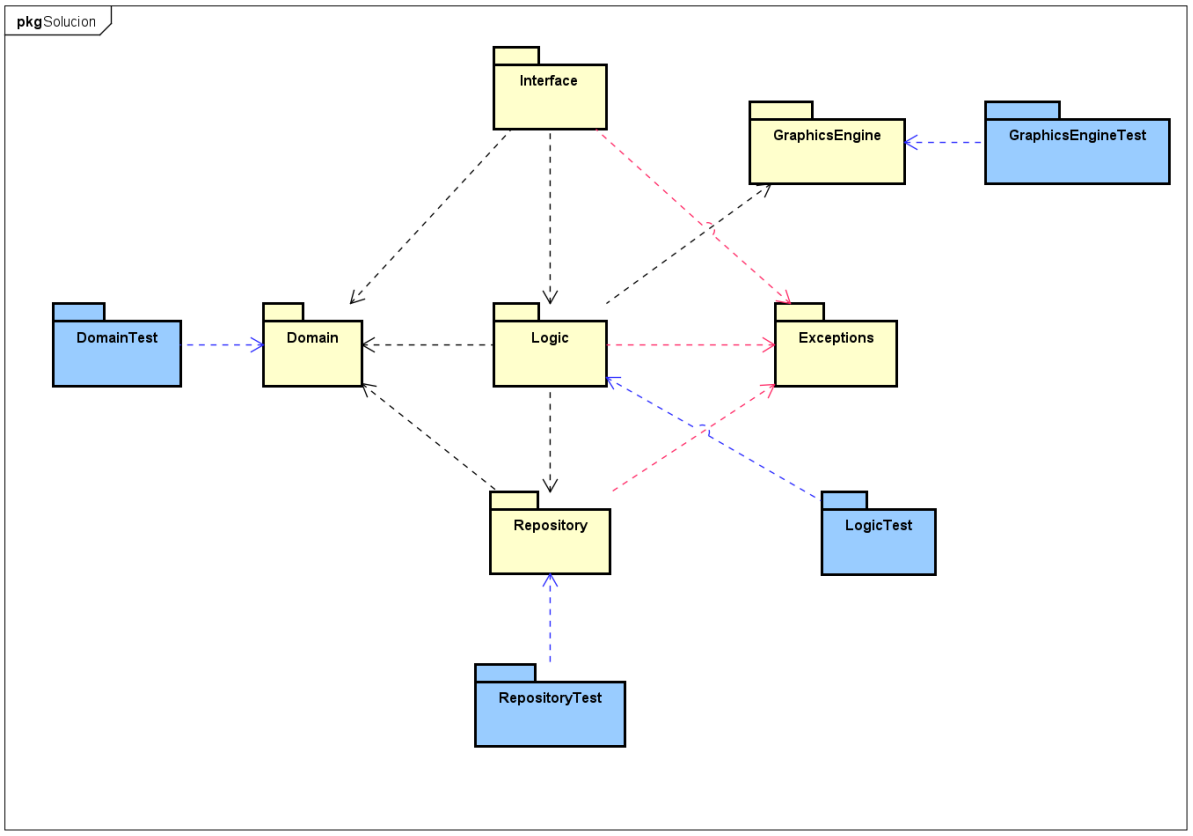
Hay varios namespaces creados para realizar las pruebas unitarias de TDD. Para cada clase a probar se ha creado a una clase de test, estas clases de tests están agrupadas en namespaces: Uno para los repositorios de datos,

otra para lógica y otra para el dominio donde están declarados objetos como figuras, lógica, etc.



Panorama general de namespaces

Como resultado de la organización y estructura realizada con los namespaces mencionados, obtenemos una estructura lógica dividida en partes y cada una de partes es testeada por un namespace de pruebas separado.
Además al tomar este enfoque, si se desea expandir y añadir nuevas funcionalidades, es posible crear namespaces dedicados a esas nuevas cosas, evitando agregar complejidad al namespace de lógica por ejemplo.



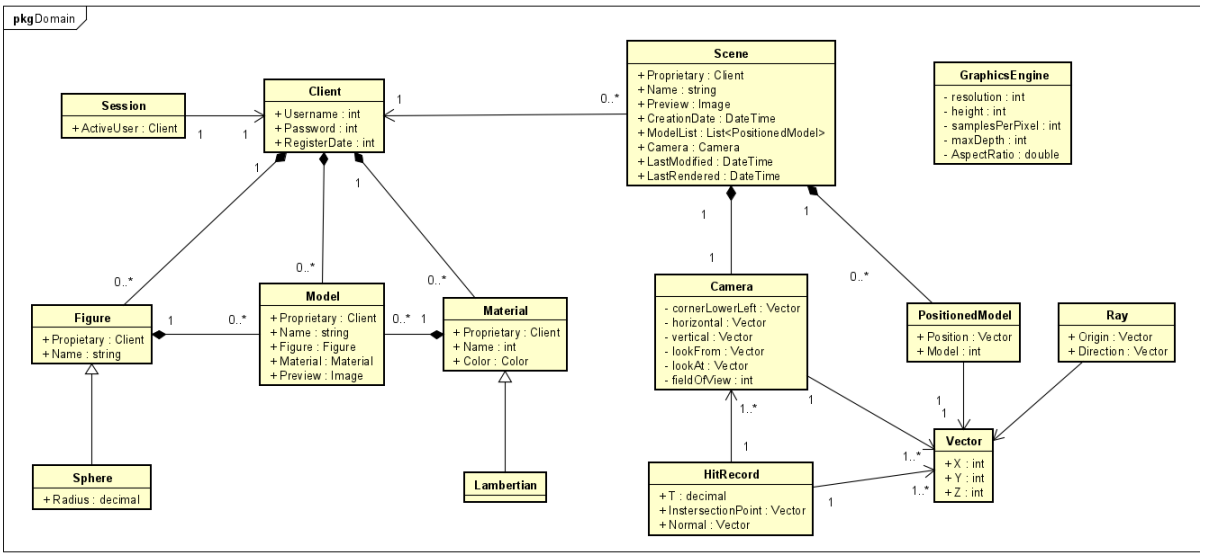
Este diagrama no es del todo representativo porque GraphicsEngine no está en un namespace aparte, sino dentro de dominio, y la clase que lo prueba está adentro de DomainTest.

Estructura de clases

Dominio

En dominio están todas las clases “atómicas”, aquí se declaran clases con atributos para guardar información y se suelen almacenar de una forma u otra, tienen métodos que suelen sólo interactuar consigo mismo o con otras clases del namespace dominio.
Aquí se guardan entidades que tienen sentido para el cliente como por ejemplo figuras, tipos de figuras, materiales, modelos, escenas, o cuentas.

Pero además de eso, también se encuentran objetos que solamente sirven de utilidad internamente en el sistema como por ejemplo cámaras, vectores, rayos, registros de impacto para los rayos.



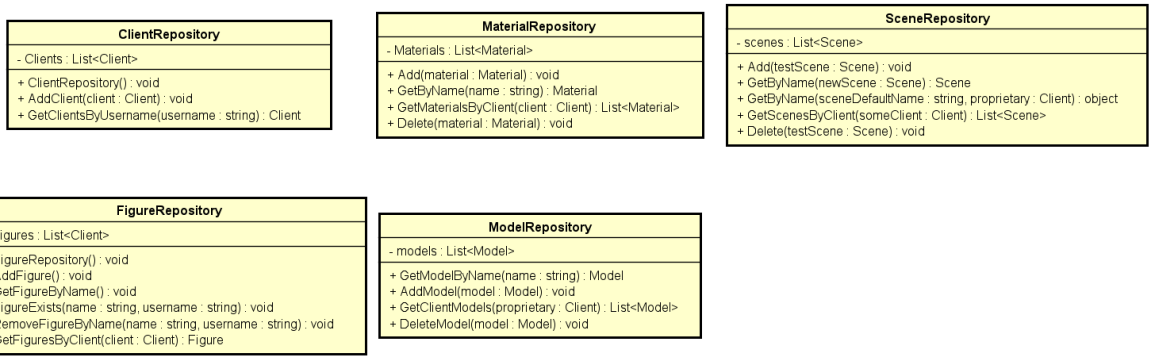
En el anexo que hay junto a la entrega se encuentra una imagen grande que incluye los métodos.

Motor gráfico en dominio

El motor gráfico temporalmente está alojado en el dominio, creemos que es un lugar inapropiado por el rol que cumple dicho namespace, cuando tengamos más tiempo veremos si podemos darle un lugar más apropiado.

Repositorios

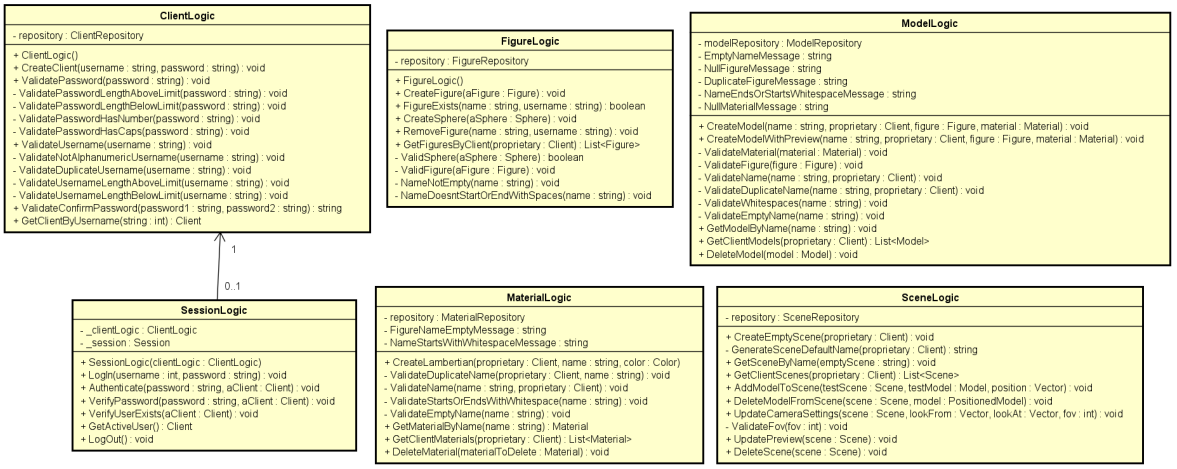
Los repositorios tienen el objetivo de almacenar los objetos del dominio en la memoria local. Todos ellos contienen una lista y métodos para añadir, borrar, buscar y retornar elementos de la lista que tienen dentro. Estas órdenes son ejecutadas por clases en el namespace de lógica.



Lógica

La lógica o “lógica de negocio” es la que se encarga de hacer la mayoría de operaciones lógicas, validaciones y acciones relevantes para el sistema. Hay varias clases de lógica, cada una se encarga de un conjunto de tareas diferentes, pero todas tienen un objetivo similar que les da el nombre de “Lógica”.

Por ejemplo se encarga de crear instancias de objetos, mandar al repositorio a guardarlas, hacer validaciones, llamar a excepciones existentes, obtener datos mediante la llamada a repositorios y otras acciones que pueden variar de clase en clase, pero que son coherentes con lo mencionado anteriormente.



Interacción con los repositorios

La mayoría de las clases de lógica necesitan guardar colecciones de objetos (aunque no todas), es por esto que éstas clases están íntimamente relacionadas con los repositorios.

Cada una de ellas que necesita guardar objetos guarda en un repositorio en uno de sus atributos. Cuando necesitan guardar, consultar datos, o borrar datos llaman a los métodos de los repositorios, aunque no sin antes hacer las validaciones necesarias para cada una de esas operaciones.

Validaciones

Como se mencionó, las clases de lógica tienen la responsabilidad de realizar validaciones. Éstas validaciones a veces se dividen en varios métodos que a su vez llaman a otros métodos dentro de la misma clase, esto con el propósito de modularizar cada uno de los diferentes tipos de validaciones, además de reducir la cantidad de responsabilidades por función.

Al realizar está descomposición de funciones en otras funciones, los métodos en los que se descomponen los convertimos en privados para dejar en claro que no están pensados para ser usados fuera de la clase.

También se guardan algunas constantes con mensajes de error dentro, por ahora no son demasiadas, pero en un futuro puede que haga falta encontrar un mecanismo para guardar mensajes de error sin saturar la cantidad de atributos de la clase.

Futuros diferentes tipos de datos

Para poder aceptar fácilmente diferentes tipos de datos en el futuro, planeamos utilizar herencia y polimorfismo. De momento tenemos la

herencia, tenemos un método para crear un objeto de un tipo específico y un constructor general.
Por ejemplo en figuras, donde tenemos una clase de figura genérica y una esfera que hereda de dicha clase.

Sin embargo, lo que tenemos para afrontar ese problema no está listo todavía, falta implementar un plan para generar figuras de diferentes tipos (por dar un ejemplo) de forma que sea fácil de comprender y genérico para cualquier tipo de objeto que aparezca.

Interfaz

La interfaz utiliza Windows Forms, se encarga de crear o manejar elementos gráficos (no confundir con el motor gráfico) dentro de las ventanas y está aislada del resto de clases, pero no realiza operaciones de lógica de negocio.

“Instance”

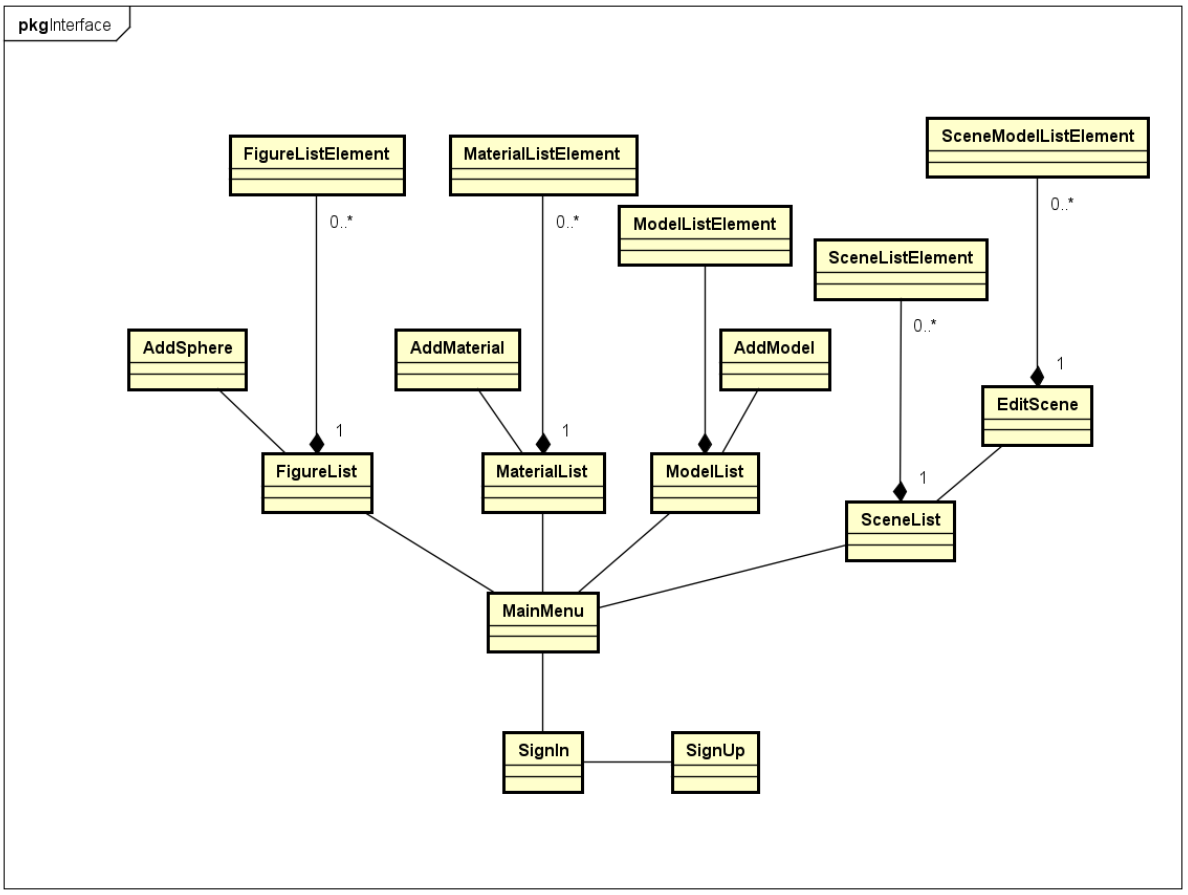
Dentro de interfaz hay una clase llamada “Instance” donde se declara una instancia estática de las lógicas de cliente, sesión, figura, material, escena y modelo para que puedan ser utilizadas por la interfaz.

Navegación

Al iniciar el programa, el punto inicial es la pantalla de login.
Hay un botón para loguearse y otro para registrarse, en la parte de registrarse uno es capaz de ingresar usuario y contraseña para registrarse, la interfaz llama a la lógica de negocio para validar los datos al escribir y al intentar crear un cliente.

Luego de registrar un usuario, es posible loguearse y acceder al menú principal.
Hay un menú lateral con las opciones de Figuras, Materiales, Modelos y Escenas. Hacer click en cada una abrirá una lista del respectivo tema y habrá una opción para agregar elementos, usarla abrirá una ventana que permitirá agregar un elemento de ese tipo a la lista.

Además, en la lista se pueden borrar elementos usando un botón.



Crear objetos dentro de la interfaz

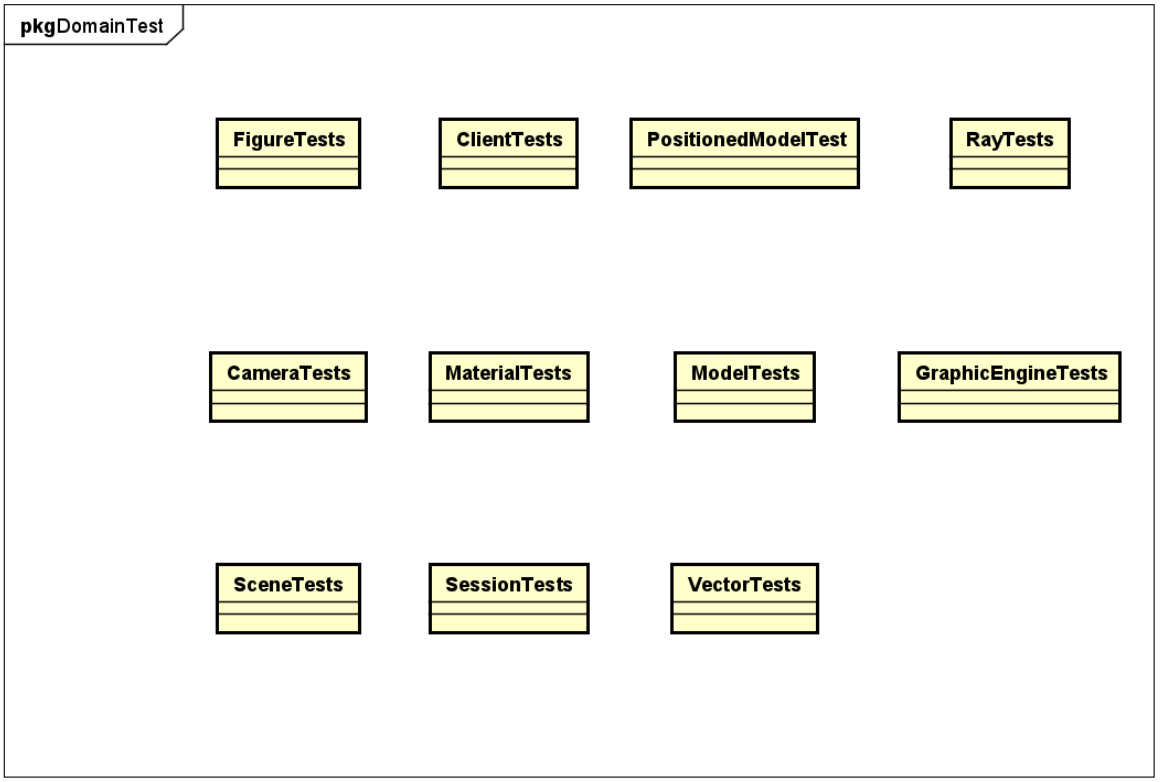
Si bien contradice a lo que dijimos de que la interfaz no hace lógica de negocio, dentro de la lógica se crean objetos del dominio para enviarselos a la lógica.

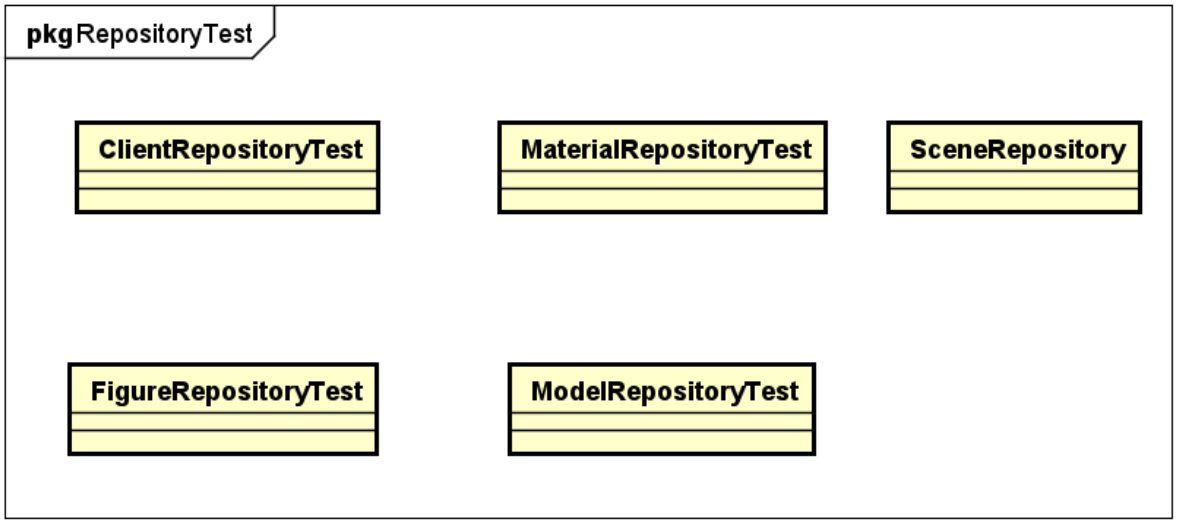
El problema con esto es que no se logra el aislamiento absoluto o casi absoluto de la interfaz con la lógica, lo cual sería ideal si queremos un sistema con menos dependencias. No hemos tenido tiempo para idear un plan para afrontar esto, está en nuestra de lista de cosas que tenemos en mente.

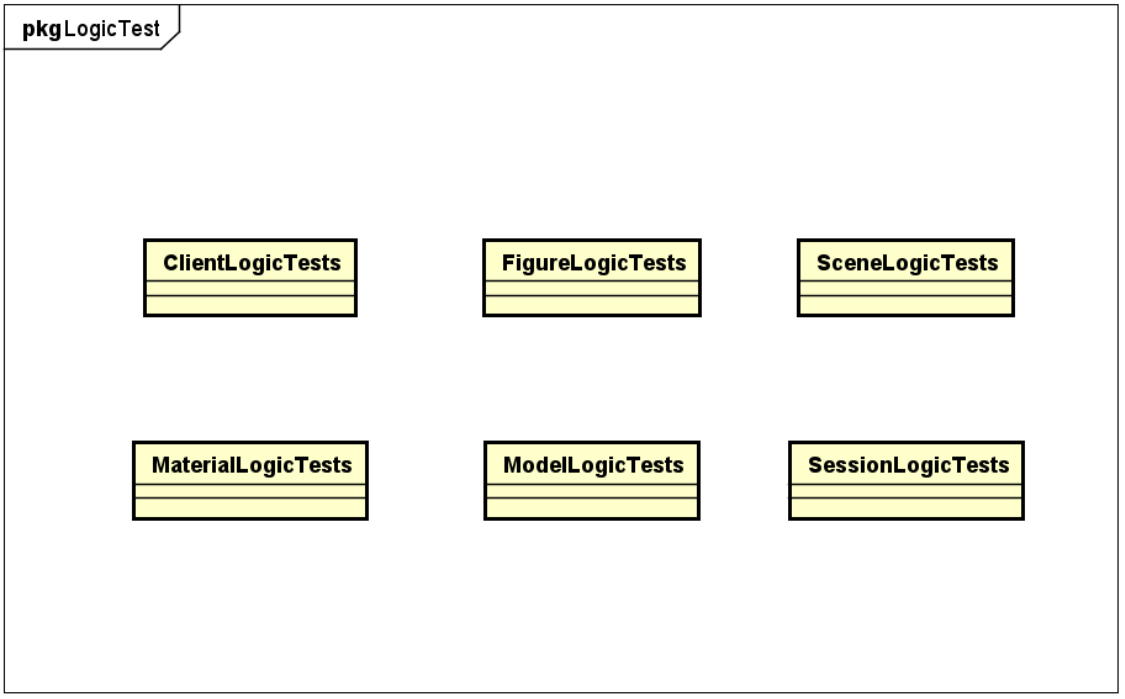
Clases de test

Para asegurar la robustez del sistema a largo plazo, toda clase existente en el sistema, excepto interfaz, es probada con una clase de test unitario dedicado específicamente a esa clase.

En los siguientes diagramas de clases de test no se especificará atributos o métodos ya que el propósito de estas clases es probar clases y no forman parte de la funcionalidad. Para ver los casos de prueba se recomienda verlos en el código de la solución.







Por qué no interfaz

La interfaz no se prueba con test unitarios porque sale de nuestro alcance.

Los elementos de la interfaz son de terceros lo cual dificulta nuestro entendimiento de la misma, además la interfaz es dificil de predecir y hay muchos casos en los que los tests unitarios a nivel de consola no son capaces de detectar que algo no funciona de manera esperada.

Como por ejemplo que un texto se salga de la ventana y que el usuario no lo vea, no sabemos siquiera si es posible detectar algo como eso.

Cómo están organizados

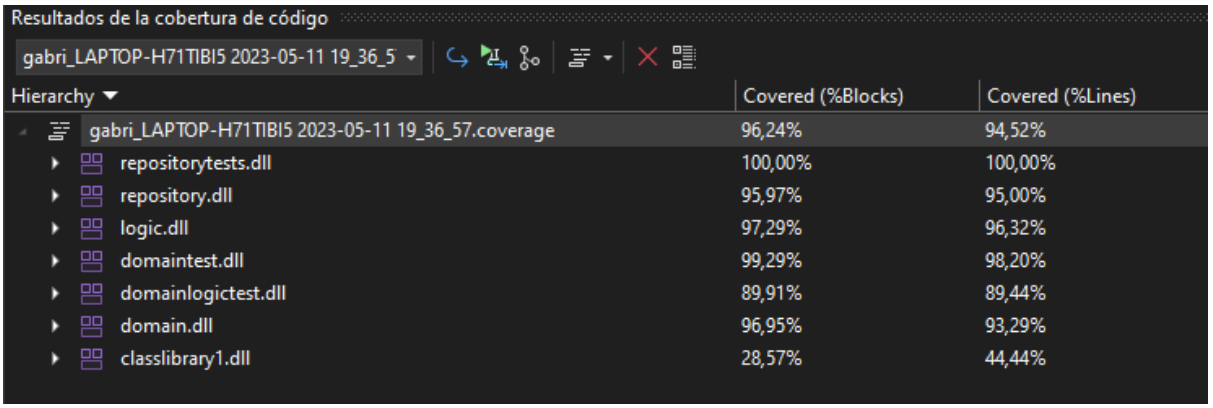
Cada namespace de pruebas de los ya mostrados anteriormente en la sección de namespaces, cuenta con varias clases de pruebas unitarias en su interior.

Naturalmente, ninguna de las clases del sistema depende de estas clases de prueba para funcionar.

Testing y coverage de las pruebas

Coverage

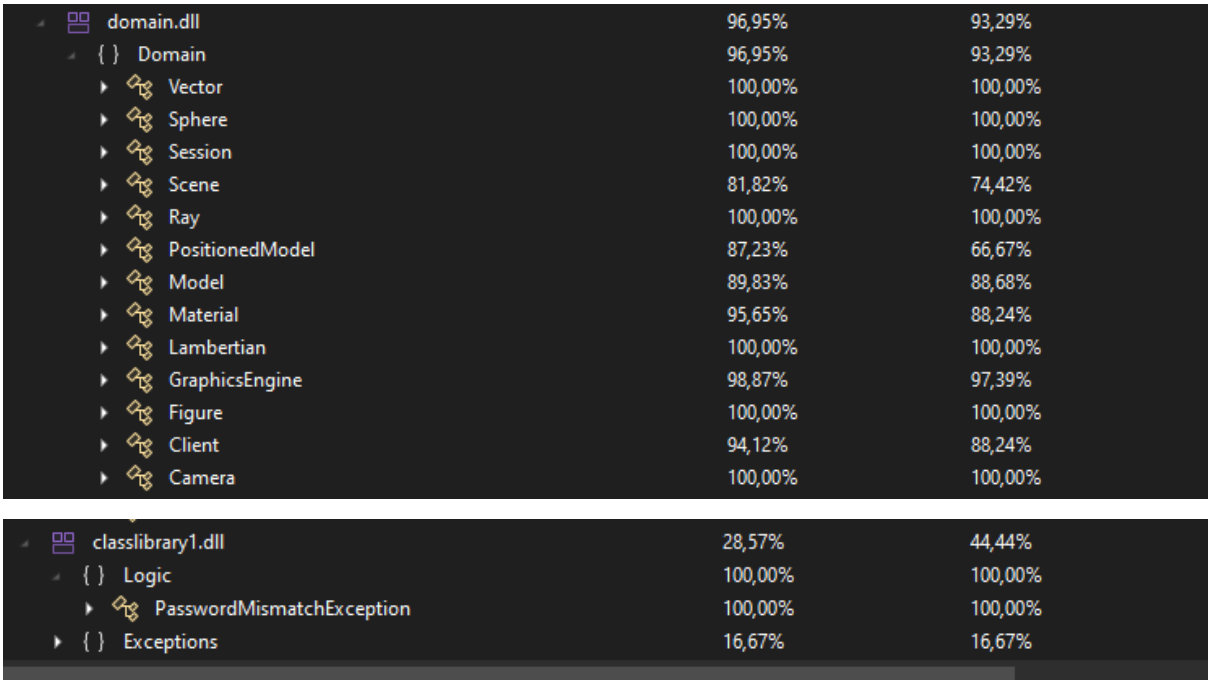
Al aplicar el análisis de cobertura, el resultado de los porcentajes de cobertura es el siguiente.
Recordar que sólo se debe analizar la cobertura de las clases de implementación, no los dll de las clases de prueba.



Hierarchy	Covered (%Blocks)	Covered (%Lines)
gabri_LAPTOP-H71TIBI5 2023-05-11 19_36_57.coverage	96,24%	94,52%
repositorytests.dll	100,00%	100,00%
repository.dll	95,97%	95,00%
{ } Repository	95,97%	95,00%
SceneRepository.<>c__DisplayClass4_0	100,00%	100,00%
SceneRepository.<>c__DisplayClass3_0	87,50%	0,00%
SceneRepository.<>c__DisplayClass2_0	100,00%	100,00%
SceneRepository	100,00%	100,00%
ModelRepository.<>c__DisplayClass3_0	100,00%	100,00%
ModelRepository.<>c__DisplayClass1_0	100,00%	100,00%
ModelRepository	100,00%	100,00%
MaterialRepository.<>c__DisplayClass5_0	100,00%	100,00%
MaterialRepository.<>c__DisplayClass4_0	100,00%	100,00%
MaterialRepository	100,00%	100,00%
FigureRepository.<>c__DisplayClass9_0	0,00%	0,00%
FigureRepository.<>c__DisplayClass6_0	100,00%	100,00%
FigureRepository	95,65%	93,75%
ClientRepository.<>c__DisplayClass6_0	100,00%	100,00%
ClientRepository	100,00%	100,00%
logic.dll	97,29%	96,32%
domaintest.dll	99,29%	98,20%
domainlogictest.dll	89,91%	89,44%
domain.dll	96,95%	93,29%

Hierarchy	Covered (%Blocks)	Covered (%Lines)	Covered (Blocks)	Not Covered (Bl
gabri_LAPTOP-H71TIBI5 2023-05-11 19_13_47.coverage	96,24%	94,44%	2943	115
repositorytests.dll	100,00%	100,00%	186	0
repository.dll	96,30%	93,65%	130	5
logic.dll	97,73%	96,93%	302	7
{ } Logic	97,73%	96,93%	302	7
SessionLogic	100,00%	100,00%	24	0
SceneLogic	96,43%	95,24%	54	2
ModelLogic.<>c__DisplayClass11_0	100,00%	100,00%	3	0
ModelLogic	100,00%	100,00%	61	0
MaterialLogic.<>c__DisplayClass4_0	100,00%	100,00%	3	0
MaterialLogic	100,00%	100,00%	39	0
FigureLogic	91,49%	87,10%	43	4
ClientLogic	98,68%	98,11%	75	1
domaintest.dll	99,39%	98,19%	972	6
domainlogictest.dll	89,15%	87,78%	501	61
domain.dll	97,03%	94,21%	848	26
classlibrary1.dll	28,57%	37,50%	4	10

logic.dll	97,29%	96,32%
{ } Logic	97,29%	96,32%
SessionLogic	100,00%	100,00%
SceneLogic	96,72%	94,92%
ModelLogic.<>c__DisplayClass11_0	100,00%	100,00%
ModelLogic	100,00%	100,00%
MaterialLogic.<>c__DisplayClass4_0	100,00%	100,00%
MaterialLogic	100,00%	100,00%
FigureLogic	89,09%	85,19%
ClientLogic	98,70%	98,70%



domain.dll	96,95%	93,29%
Domain	96,95%	93,29%
Vector	100,00%	100,00%
Sphere	100,00%	100,00%
Session	100,00%	100,00%
Scene	81,82%	74,42%
Ray	100,00%	100,00%
PositionedModel	87,23%	66,67%
Model	89,83%	88,68%
Material	95,65%	88,24%
Lambertian	100,00%	100,00%
GraphicsEngine	98,87%	97,39%
Figure	100,00%	100,00%
Client	94,12%	88,24%
Camera	100,00%	100,00%

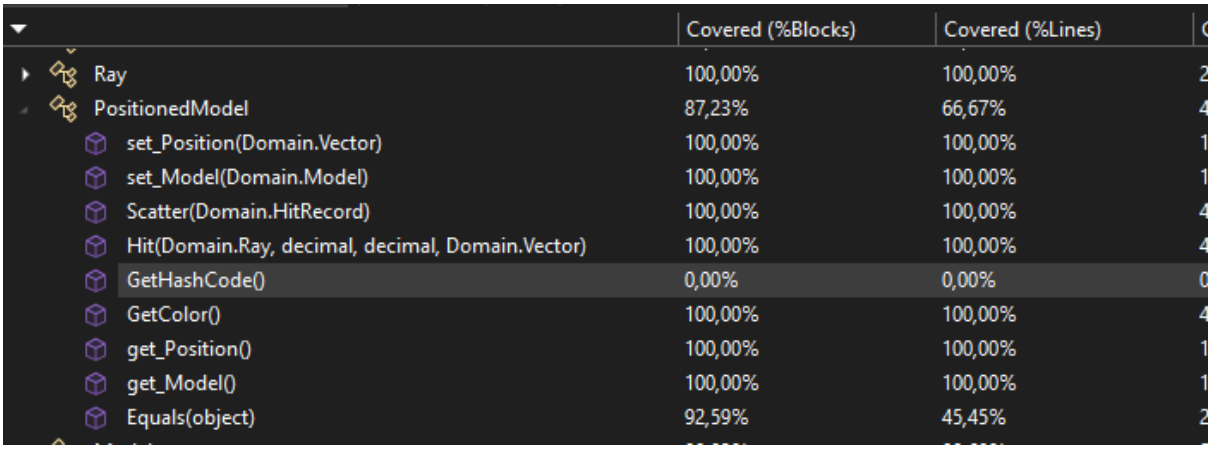
classlibrary1.dll	28,57%	44,44%
Logic	100,00%	100,00%
PasswordMismatchException	100,00%	100,00%
Exceptions	16,67%	16,67%

Algunas clases tienen una cobertura inferior a 90%
Estas son clases de excepciones, escenas, motor gráfico, model y positionedModel.

Justificaciones

Las clases de excepciones sólo contienen la declaración de las mismas, a su vez las excepciones han sido testeadas al ser utilizadas en los tests de las otras clases para las que se crearon. En los tests unitarios actuales se evidencia su funcionamiento correcto.

Scenes, motor gráfico, model y positionedModel tienen una única función sin probar que les baja el promedio significativamente. Esta función es “getHashCode”.



	Covered (%Blocks)	Covered (%Lines)
Ray	100,00%	100,00%
PositionedModel	87,23%	66,67%
set_Position(Domain.Vector)	100,00%	100,00%
set_Model(Domain.Model)	100,00%	100,00%
Scatter(Domain.HitRecord)	100,00%	100,00%
Hit(Domain.Ray, decimal, decimal, Domain.Vector)	100,00%	100,00%
GetHashCode()	0,00%	0,00%
GetColor()	100,00%	100,00%
get_Position()	100,00%	100,00%
get_Model()	100,00%	100,00%
Equals(object)	92,59%	45,45%

Reconocemos que es un error a la hora de seguir TDD, esto ocurrió debido a la falta de tiempo y el apuro que eso conlleva.

Pero más allá de eso, la cobertura promedia por encima del 90%, por lo que cumple con la meta establecida

Problemas y bugs conocidos

-El menú de figuras te deja borrar figuras aunque sean usadas por modelos, de hecho hay una prueba en RED para esa excepción no implementada.

-El texto de los objetos figura, material, etc se sale de la ventana o del user control cuando éste es demasiado largo.

-Hay inconsistencias estéticas en la interfaz, algunas listas tienen los elementos con fondo blanco, otras no.

-Salir de editar escena y volver hace que los txtLabel con la información de LookAt, LookFrom, etc deje de mostrarse por haber recargado la ventana. No es error, pero es molesto que la información deje de verse.

-Al pedir coordenadas, éstas se piden una a una, esto es molesto y es un problema de usabilidad.

Técnicas de Clean Code y estándares de código

Se han seguido y aplicado a medida de lo posible las reglas de Clean Code. Se ha hecho esfuerzo por mantener nombres de funciones claros, nombres cortos para métodos muy usados, métodos más comunes arriba del todo de una clase, se han usado nombres que describan la función de los métodos y el propósito de las variables, se ha intentado evitar código que ocupa mucho volumen, entre otras técnicas.

Cosas que no hicimos tan bien

Hay cierta inconsistencia con firmas de funciones similares. Por ejemplo, “RemoveFigure()” del repositorio de figuras y “DeleteMaterial()” del repositorio de materiales hacen esencialmente lo mismo, pero el nombre diferente puede hacer más difícil recordar el nombre del método.

Los constructores de clases con muchos atributos reciben más que 3 parámetros, incluso haciendo uso de objetos, no hemos encontrado solución a esto todavía.

Es posible que algunos métodos hagan más de una cosa (Lo cual prohíbe Clean Code).
Estamos trabajando en aprender cómo modularizar el código más de lo acostumbrado.

Git

Commits de TDD

Seguir la estrategia de TDD implica respetar el ciclo de “Red, Green, Refactor”.
Esto se ve reflejado en los commits del repositorio, donde se pone [RED], [GREEN] o [REFACTOR] según la etapa de TDD en la que estamos.

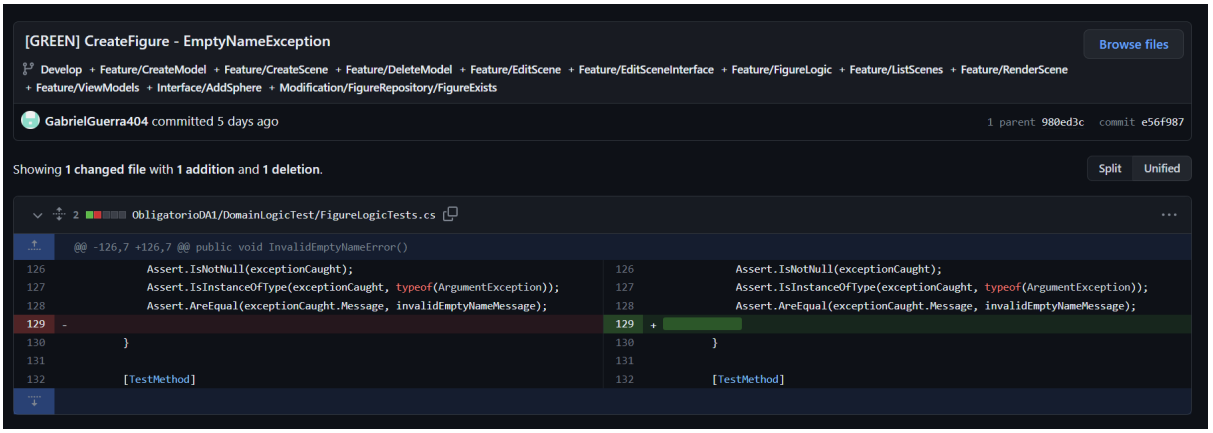
Estrategia de uso de ramas

Para cada feature a crear se han creado ramas usando el formato feature/<nombre de la feature>
Es posible encontrar variaciones de este formato en nuestro repositorio como “Interface/AddSphere” o “Modification/FigureRepository/FigureExists”.

Aclaraciones de algunos commits

Aclaración 1

En cierta ocasión se han creado commits en blanco para reflejar los commits de [GREEN] de algunos casos. Esto es porque se trabajaron en varias funciones antes de hacer commits.
Esto fue causado por la falta de práctica con esta estrategia y tratará de ser evitado en el futuro.



Los commits en cuestión son los de esta zona.

Aclaración 2

También se han realizado hotfixes y refactoreos directamente en develop, lo cual reconocemos que no es buena práctica, recientemente hemos descubierto una manera de evitarlo e intentaremos aplicarla.

Anexo

Link al repositorio: https://github.com/ORT-DA1-2023/275723_276712