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

El objetivo principal del proyecto es crear un producto mantenible, expandible y modificable a largo plazo. Esto garantiza la productividad del equipo de desarrollo y minimiza los problemas a lo largo del tiempo.

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

El software se dividió en secciones lógicas utilizando namespaces y clases para separar las responsabilidades. Esto promueve un comportamiento claro, coherente y de menor complejidad en el sistema. Además los errores se limitan a una parte más pequeña del sistema, lo que facilita su identificación.

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) mejora la mantenibilidad. Los tests frecuentes y con buena cobertura identifican rápidamente funciones o características que dejan de funcionar en el código, lo cual se va volviendo muy beneficioso a medida que crece en complejidad. TDD ayuda a reducir problemas y a lograr un sistema más robusto.

Clean Code y refactoreos

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.

Uso de GRASP, SOLID y patrones de diseño:

Se utilizaron algunas de las técnicas y estrategias de diseño encontradas en GRASP, en SOLID, y en los patrones de diseño encontrados en la industria del software, para mejorar la calidad y mantenibilidad de nuestro software.

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: Al namespace que controla el acceso a datos lo llamamos “Repositorio” por su función. Tiene la función de almacenar, consultar, borrar datos y proveer datos cuando se lo piden.

[Figura 1]

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

En el diseño se evitó hacer dependencias circulares, como el caso en el que la clase Lógica depende del repositorio pero el repositorio no depende de Lógica.
Esto evita errores en cadena y proporciona una mayor claridad en el comportamiento del programa al no agregar complejidad innecesaria.

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. Esto evita que una sola lógica tenga demasiadas responsabilidades y se implementan varias para distribuir las tareas.

La capa de datos se divide en varios “repositorios”, cada uno almacena y manipula un tipo específico de datos como pueden ser figuras o clientes. 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 almacenar información dentro de ellos y validar sus propios campos.

[Figura 2]

Motor gráfico

El motor gráfico se encuentra dentro del dominio en un subpaquete, éste se encarga de guardar los ajustes del Motor Gráfico y se encarga de hacer las operaciones de RayTracing.

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

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.

[Figura 3]

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.

[Figura 4]

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.

[Figura 4.1]

Estructura de clases

Dominio

El dominio contiene clases "atómicas" que almacenan información y se relacionan con otras clases del mismo namespace. Aquí se encuentran entidades como figuras, tipos de figuras, materiales, modelos, escenas y cuentas de usuario. También se incluyen objetos internos del sistema como cámaras, vectores, rayos y registros de impacto.

[Figura 5]

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

Repositorios

Los repositorios tienen el objetivo de almacenar los objetos del dominio en una base de datos de Entity Framework. Todos ellos contienen una lista y métodos para añadir, borrar, buscar y retornar elementos almacenados.
Estas órdenes son ejecutadas por clases en el namespace de lógica.

Lógica

El namespace de lógica de negocio realiza operaciones lógicas, validaciones y acciones relevantes para el sistema. Hay varias clases de lógica que se encargan de diferentes tareas, pero comparten un objetivo y eso es ser el “cerebro” del programa. Esto incluye crear instancias de objetos, guardar en el repositorio, realizar algunas validaciones, manejar excepciones, obtener datos de los repositorios y otras acciones coherentes con lo mencionado anteriormente.

No hay muchas relaciones entre las clases de lógica, pero igualmente aparecen este diagrama.

[Figura logica]

Interacción con los repositorios

Las clases de lógica están relacionadas con los repositorios, ya que muchas de ellas necesitan objetos. Estas clases guardan un repositorio en uno de sus atributos y llaman a los métodos correspondientes para guardar, consultar o borrar datos. Antes de realizar estas operaciones, se realizan validaciones necesarias.

Las operaciones con los repositorios se hacen mediante interfaces. Los métodos de las interfaces son implementados por clases que interactúan directamente con la base de datos y realizan las operaciones solicitadas.

Un ejemplo de eso es la clase de ClientLogic en la siguiente imagen.

[Figura 5.1]

FigureLogic, ModelLogic, SceneLogic, MaterialLogic se comportan de forma análoga.

[Figura 5.2]

Adicionalmente aquí hay un par de diagramas de actividad que indican cómo se crea una figura y cómo se crea un material.

[Figura 5.3]

Sesión de cliente

Para manejar el caso en que el cliente ingresa sus datos e inicia sesión, creamos la lógica de SessionLogic, la cual interactúa también con ClientLogic.

Podemos ver un ejemplo de su comportamiento en el siguiente diagrama de actividad donde se muestra como un cliente inicia sesión.

[Figura 5.4]

Validaciones

Las clases de lógica tienen la responsabilidad de realizar algunas validaciones, como evitar agregar elementos repetidos en la base de datos. Por otro lado las clases de dominio se encargan de validar sus propios atributos ya que se alinea con la responsabilidad de conocer dichos datos.

Se guardan constantes de error dentro de las clases que tiran excepciones, de ésta forma se evita usar valores totalmente fijos o “hardcodeados” (“Magic Numbers” según Clean Code).

Futuros diferentes tipos de datos

Para poder aceptar fácilmente diferentes tipos de datos, utilizamos herencia y polimorfismo. Por ejemplo: Los diferentes tipos de figuras son clases que heredan de la clase figura.
Esto permite que cada tipo de figura pueda tener sus propios atributos particulares y redefinir métodos de su superclase para que funcionen consigo mismo.

Singletons

Las clases de lógica y las clases de repositorio sólo se instancian una vez en la ejecución. Se sigue el patrón de singleton, llamarlo por primera vez crea una instancia, después cada vez que se lo llama se utiliza la primera instancia creada.

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.
Su responsabilidad es permitir la interacción del usuario con el sistema, permitiéndole realizar las operaciones ofrecidas y avisando sobre errores.

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.

[Figura 6]

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 interfaz se crean objetos del dominio para enviárselos a la lógica.

Esto genera acoplamiento, pero nos permite cumplir con la regla de Clean Code de tener 3 parámetros o menos en un método; la alternativa es pasarle todos los atributos al constructor y romper la regla de Clean Code. Debido a la falta de tiempo tuvimos que elegir una de las dos malas opciones.

En el futuro es posible crear un DTO y pasar los atributos a la lógica de esta forma. Así se evitaría el acoplamiento entre interfaz y lógica, y además, se respetaría Clean Code.

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.

Por qué no interfaz

No probamos la interfaz con test unitarios debido a que al ser código de terceros, comprenderlo y predecirlo es difícil. Además los tests unitarios de consola pueden no detectar problemas como texto que se sale de una ventana y componentes que no se ven bien.

Cómo están organizados

Cada namespace (Dominio, Lógica, Repositorio) tiene un namespace de pruebas correspondiente (Domain Tests, LogicTests, RepositoryTests).
A su vez, cada clase de Dominio tiene una clase de test en DomainTests, lo mismo ocurre análogamente en Lógica y Repositorio.

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.

[Figura 7]

Algunas clases tienen una cobertura inferior a 90%
Estas son clases de excepciones, Scene, Model, PositionedModel, ImageSaver y los repositorios

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.
Scene tiene métodos de renderizado los cuales no se prueban directamente, pero su funcionamiento está probado en las clases del motor gráfico.
Image saver no fue testeado ya que consideramos que no era necesario testearlo ya que utiliza clases del sistema para guardar archivos.
Los repositorios tienen un try catch en caso de que falle la conexión con la base de datos, lo que hace que baje el coverage ya que no testeamos los casos en los que la base de datos no funcione correctamente.
Model y PositionedModel tienen una única función sin probar que les baja el promedio significativamente. Esta función es “getHashCode” la cual es usada por el equals.

Reconocemos que los tests de prueba faltantes son un error a la hora de seguir TDD, los casos de prueba faltante son 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 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 un 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.

-Se utiliza un mismo namespace de excepciones para todas las excepciones del sistema. Esto puede generar acoplamiento indeseado.

-El código de los anteriores repositorios InMemory sigue estando dentro del proyecto. Esto está mal, pero inadvertidamente hicimos que nuestro proyecto lo necesite para funcionar. Se puede solucionar, pero requiere más tiempo que ahora no tenemos.

-Los errores emitidos por la base de datos tienen una excepción genérica (Database Error). Lo cual no es muy descriptivo, pero es mejor que mostrar el mensaje de inner exception por defecto.
Esto es mejorable.

-Debido a la complejidad de Entity Framework y el poco tiempo disponible, no se ha podido realizar un control de calidad completo sobre el acceso a base de datos. Esto significa que hicimos las pruebas de TDD sobre todas las funciones, pero no de forma muy exhaustiva.
Pero esto se puede mejorar en un futuro.

Técnicas de Clean Code y estándares de código

Se han aplicado las reglas de Clean Code, como usar nombres claros y cortos para métodos comunes, ubicar métodos comunes al principio de una clase, usar nombres descriptivos para funciones y variables, evitar código voluminoso. entre otras varias reglas.

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.

Algunos constructores de clases con muchos atributos reciben más que 3 parámetros. Esto ocurre en los casos donde la interface necesita crear ciertos objetos, ya que tiene que pasarle dichos parámetros a la capa de lógica.
En algunos constructores solucionamos esto haciendo que la interface cree el objeto y se lo pase como parámetro a la lógica, pero no lo hemos hecho en todas, eso hay que revisarlo.
En un futuro se puede solucionar utilizando un DTO, solucionando también el problema del acoplamiento.

Sufrimos de problemas de densidad vertical, en el sentido de que a veces se hacen más saltos de línea de los necesarios y debería ser corregido. Aún no hemos corregido esto debido a falta de tiempo.

Se hace uso de Magic Numbers, esto es contraproducente para la comprensión del código y debería ser corregido en futuras versiones del proyecto. Aún no hemos corregido esto debido a falta de tiempo.

Se retorna null en algunas funciones. Esto es una mala práctica y se debería hacer uso de try-catch en su lugar. Aún no hemos corregido esto debido a falta de tiempo.

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”.

Grasp, Solid, Patrones

Se ha hecho uso de técnicas y estrategias de Grasp, Solid y Patrones.

Haremos algunas menciones de funcionalidades o instancias en que fueron aplicadas.

- Las lógicas se encargan de crear objetos y mandar a almacenar objetos, cumpliendo con la función de “Creador”.
- Las responsabilidades se reparten de tal forma de que cada clase y cada namespace es “experto” en cierto ámbito y los métodos relacionados con esa responsabilidad se les asignan a ellos.
- Las clases de dominio, conocen sus propios atributos, y son “expertos” en validar sus propios campos.
- Se ha utilizado polimorfismo para aprovechar sus ventajas y hacer buen uso de la reutilización de código.
- Se han creado clases de utilidades, como por ejemplo serializar, que no encajan con otras responsabilidades y se han hecho clases aparte para ellas. Esto en patrones se llama Fabricación Pura.
- Se ha intentado mantener una alta cohesión dividiendo las responsabilidades de forma comprensible y sin saturar a ninguna clase.
- Se ha intentado mantener un bajo acoplamiento, si bien hemos fallado en algunos aspectos (Como por ejemplo que Interfaz cree objetos, en vez de hacerlo la Lógica), se ha tenido el objetivo en cuenta.
- Se ha intentado que cada clase o método tenga una sola responsabilidad, uno de los principios de SOLID. Por ejemplo: Repositorio hace acceso a datos, Motor Gráfico hace las operaciones del motor gráfico, pero ninguna de las dos tiene una responsabilidad aparte de ésa.
- Implementamos algunas interfaces para cumplir el principio de inversión de dependencias, aunque no se implementaron todas las que se podían implementar.
- Se implementaron singletons, las clases de Lógica son ejemplos de ello.

Datos de prueba

Se ha creado un bak con los datos de prueba solicitados.
La versión de Entity Framework instalada es 6.
Se debe instalar SQL Server Express y el manager para utilizarlo.

Los strings de conexión son:

```
<connectionStrings>
      <add name="RayTracingContext" providerName="System.Data.SqlClient"

connectionString="Server=.\SQLExpress;Database=Obligatorio1DbDA1;Trusted_Conn
ection=True; MultipleActiveResultSets=True;"/>
   </connectionStrings>
```

Los username y contraseña de los usuarios son:
Username: Antonio
Contraseña: Antonio123

Username: Jose
Contraseña: Jose123

Username: Miguel
Contraseña: Miguel123

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.

[Figura 8]

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

Figura 1 (Ir a donde es citada):

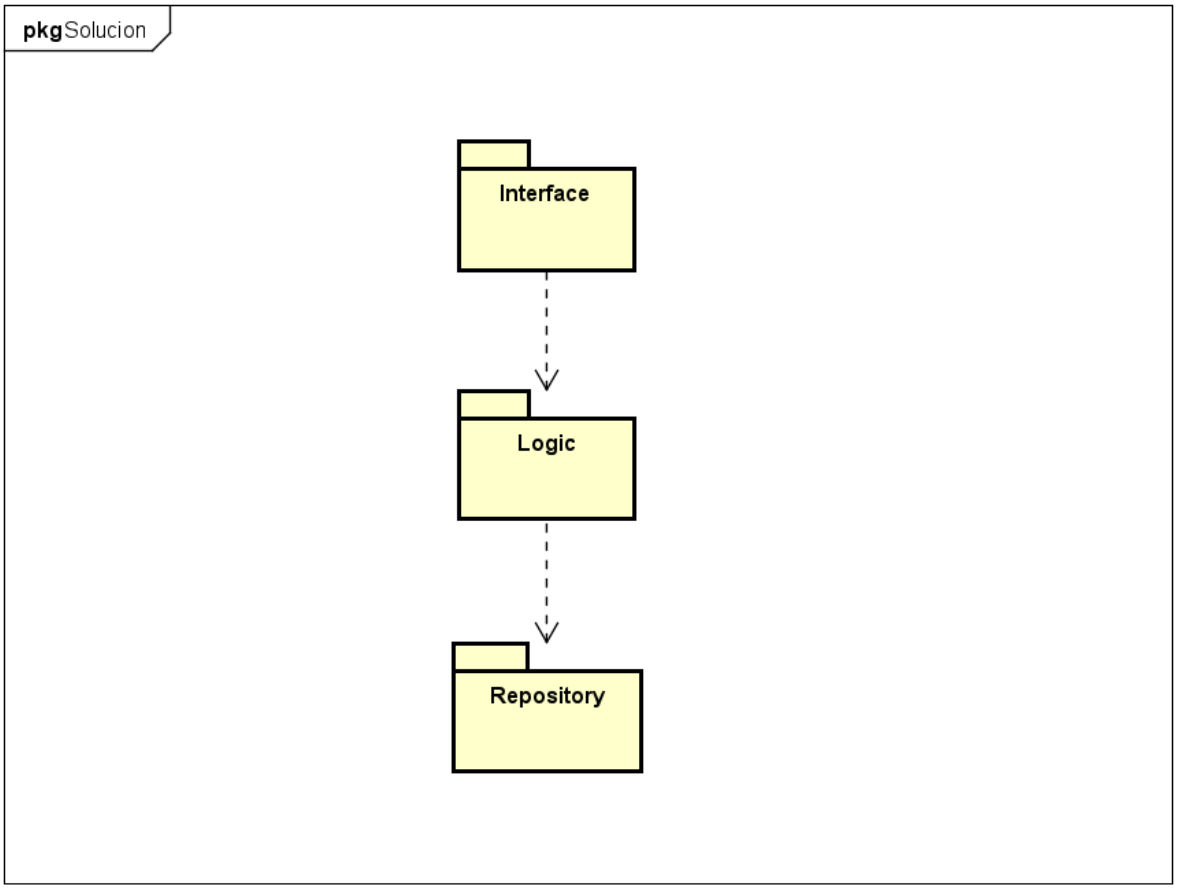


Figura 2 (Ir a donde es citada):

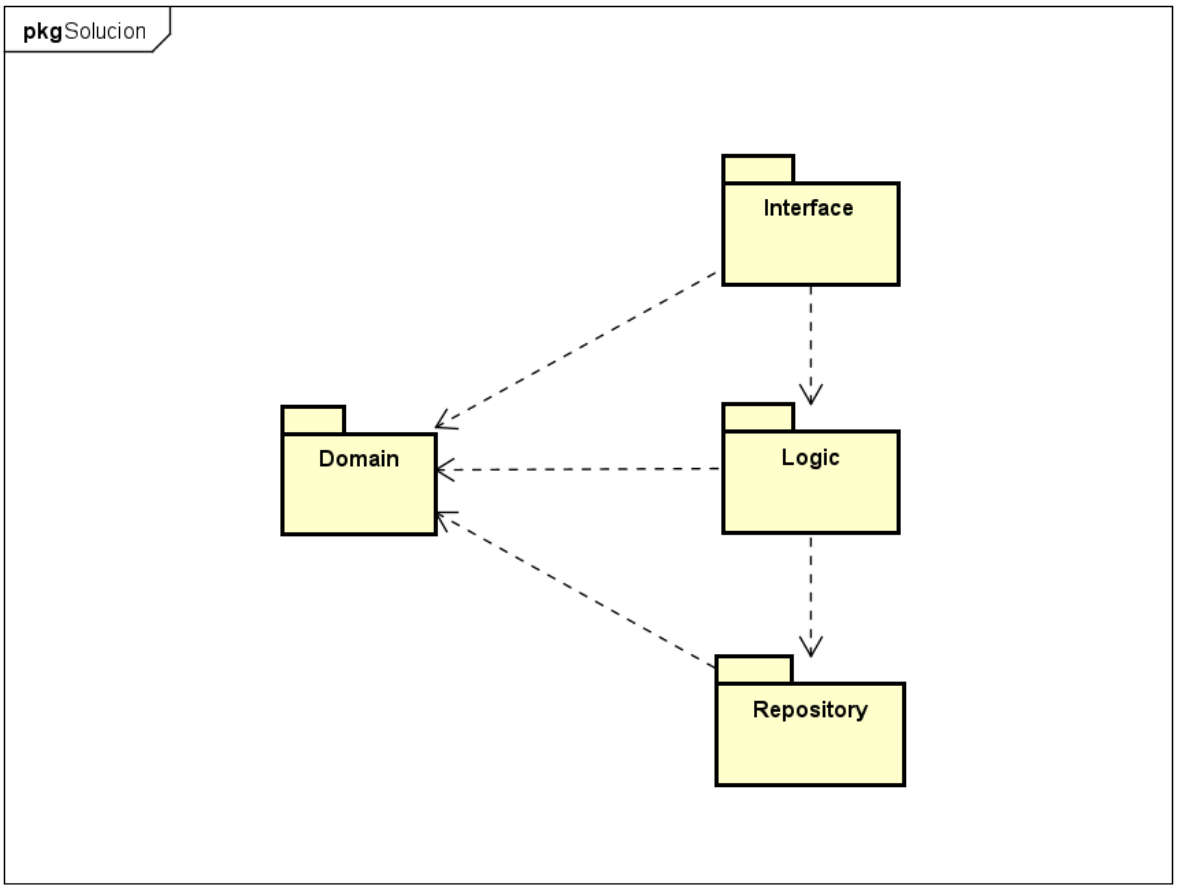


Figura 3 (Ir a donde es citada):

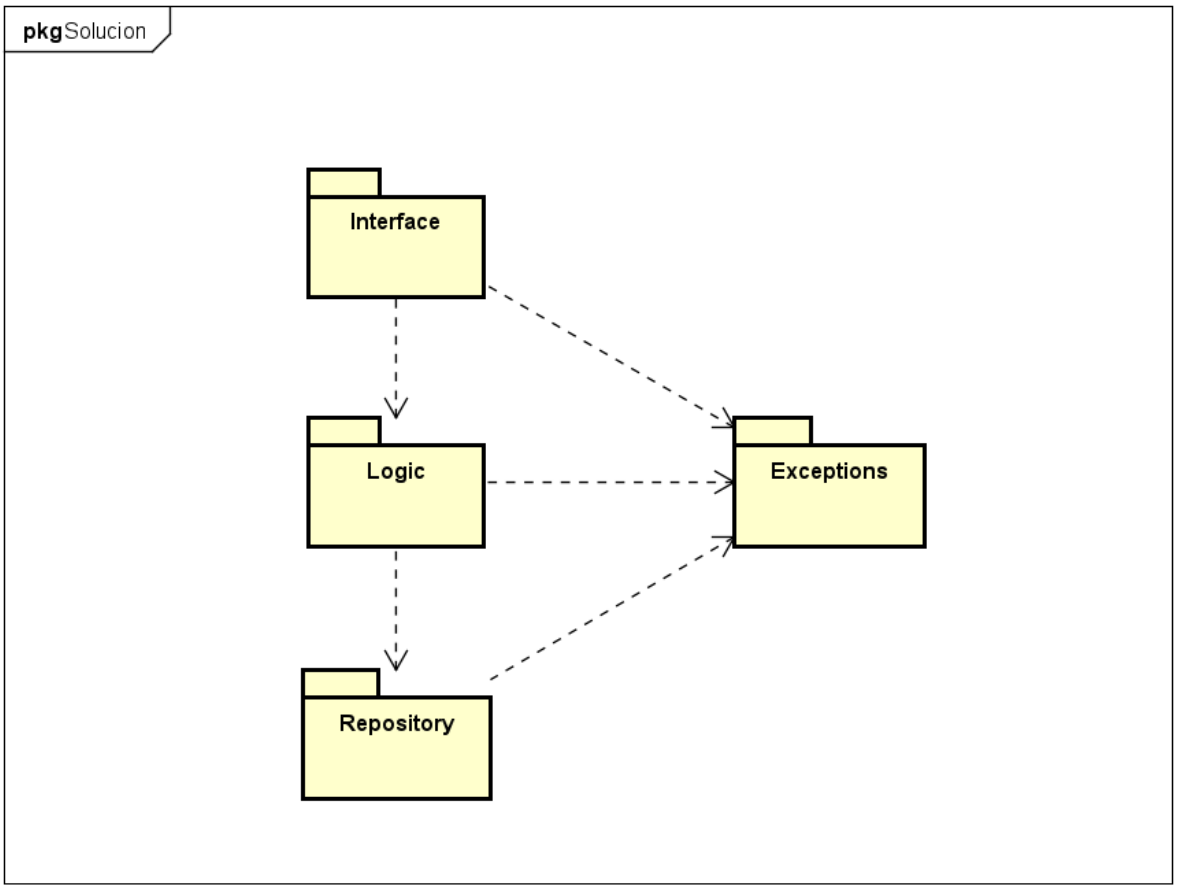


Figura 4 (Ir a donde es citada):

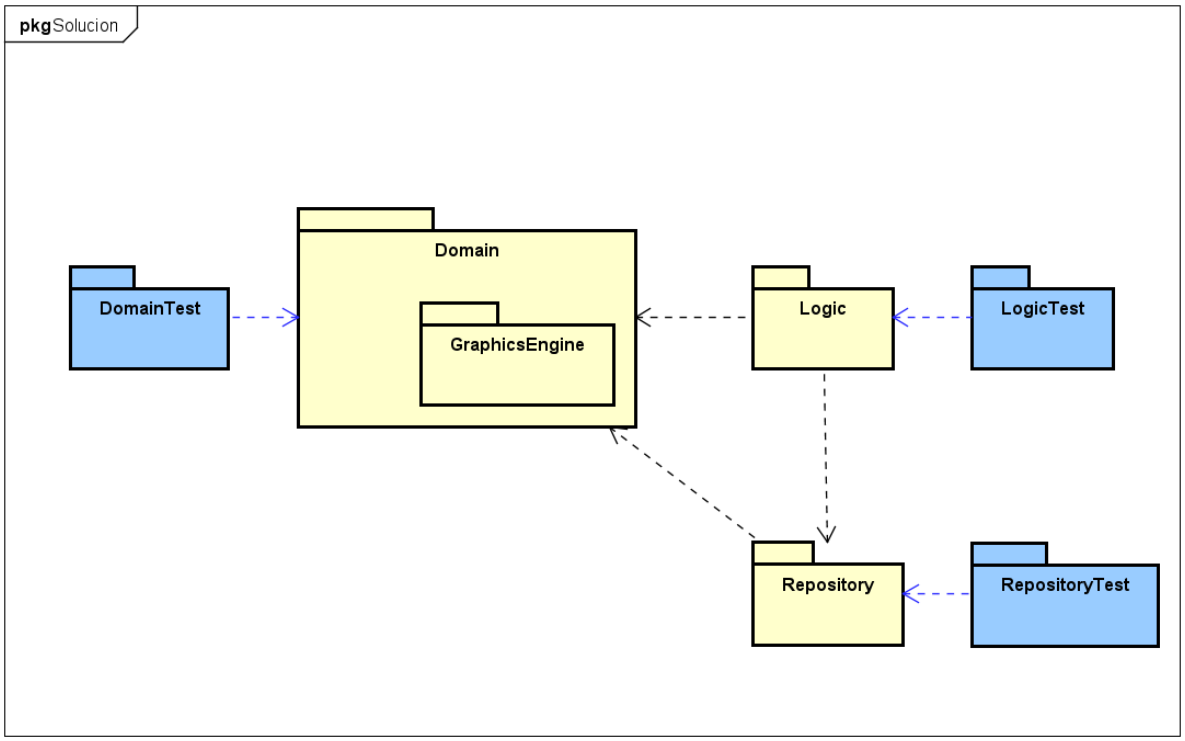


Figura 4.1 (Ir a donde es citada):

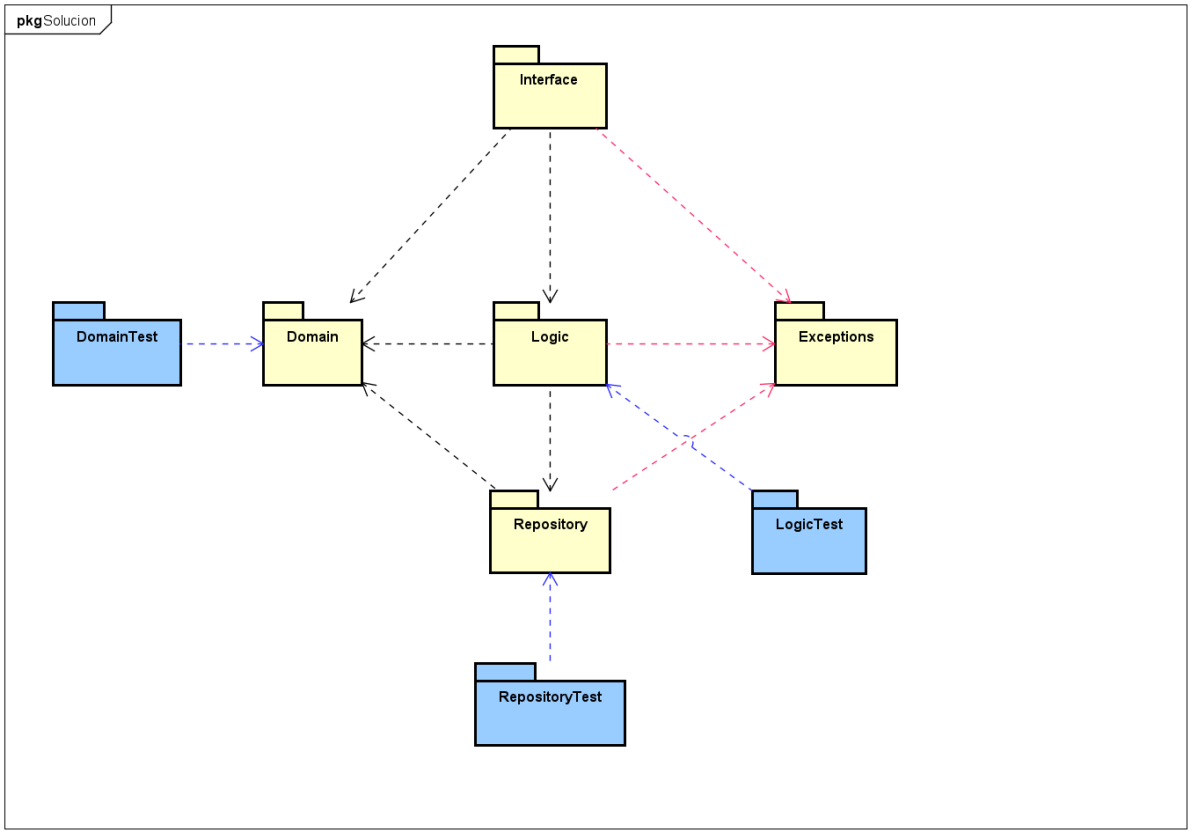
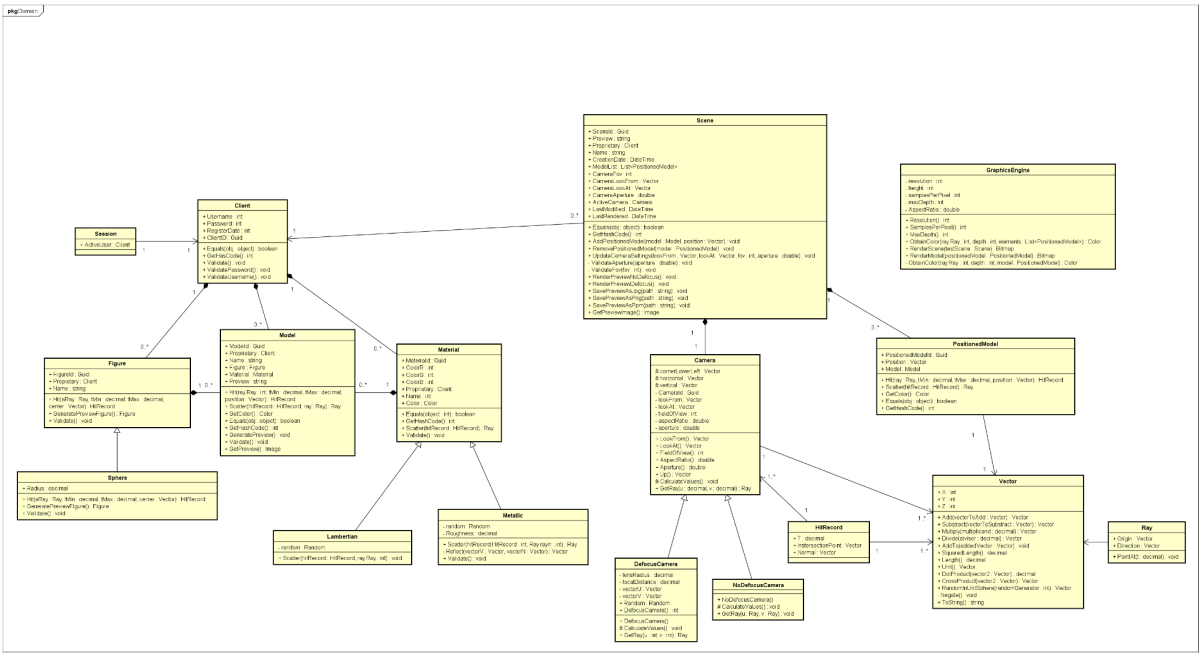
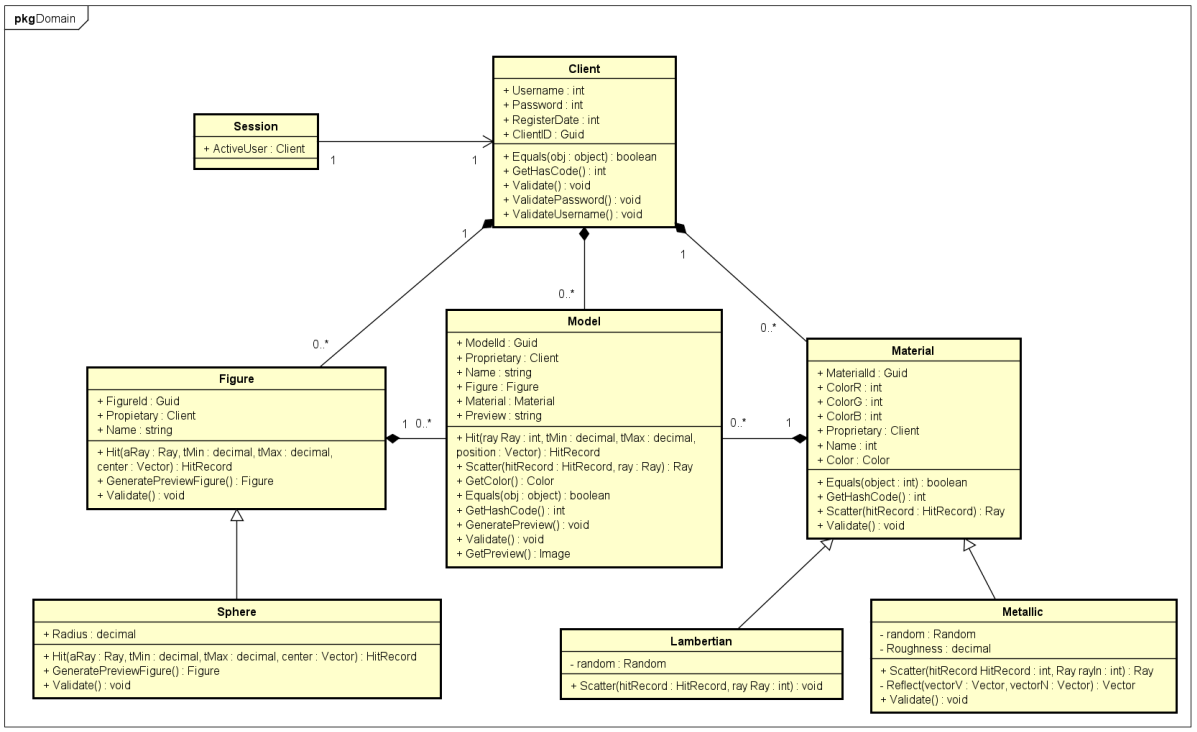
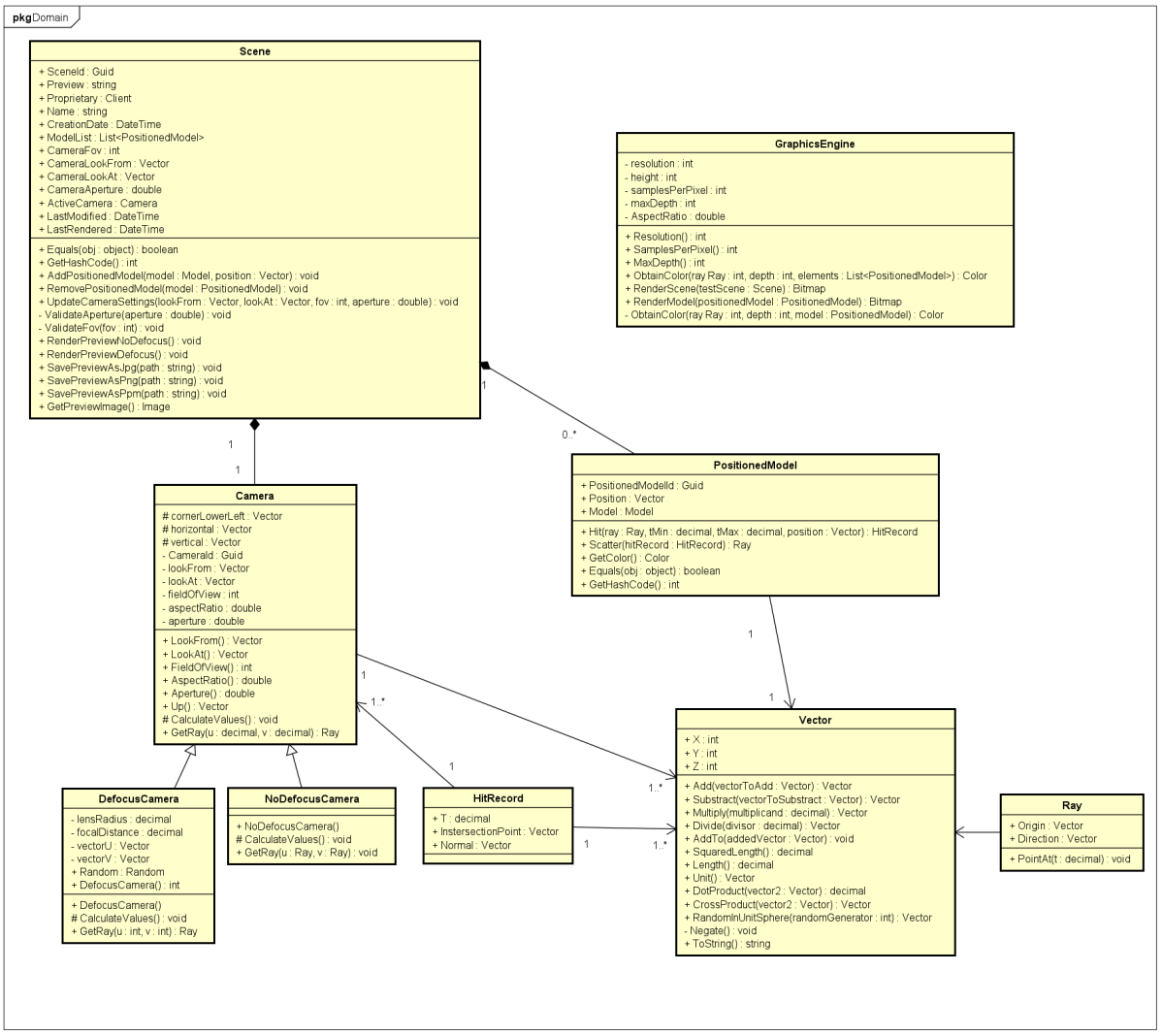


Figura 5 (Ir a donde es citada):






pkgDomain
Scene
+ SceneId : Guid
+ Preview : string
+ Proprietary : Client
+ Name : string
+ CreationDate : DateTime
+ ModelList : List<PositionedModel>
+ CameraFov : int
+ CameraLookFrom : Vector
+ CameraLookAt : Vector
+ CameraAperture : double
+ ActiveCamera : Camera
+ LastModified : DateTime
+ LastRendered : DateTime
+ Equals(obj : object) : boolean
+ GetHashCode() : int
+ AddPositionedModel(model : Model, position : Vector) : void
+ RemovePositionedModel(model : PositionedModel) : void
+ UpdateCameraSettings(lookFrom : Vector, lookAt : Vector, fov : int, aperture : double) : void
- ValidateAperture(aperture : double) : void
- ValidateFov(fov : int) : void
+ RenderPreviewNoDefocus() : void
+ RenderPreviewDefocus() : void
+ SavePreviewAsJpg(path : string) : void
+ SavePreviewAsPng(path : string) : void
+ SavePreviewAsPpm(path : string) : void
+ GetPreviewImage() : Image
GraphicsEngine
- resolution : int
- height : int
- samplesPerPixel : int
- maxDepth : int
- AspectRatio : double
+ Resolution() : int
+ SamplesPerPixel() : int
+ MaxDepth() : int
+ ObtainColor(ray Ray : int, depth : int, elements : List<PositionedModel>) : Color
+ RenderScene(testScene : Scene) : Bitmap
+ RenderModel(positionedModel : PositionedModel) : Bitmap
- ObtainColor(ray Ray : int, depth : int, model : PositionedModel) : Color
1
0..*
1
1
PositionedModel
+ PositionedModelId : Guid
+ Position : Vector
+ Model : Model
+ Hit(ray : Ray, tMin : decimal, tMax : decimal, position : Vector) : HitRecord
+ Scatter(hitRecord : HitRecord) : Ray
+ GetColor() : Color
+ Equals(obj : object) : boolean
+ GetHashCode() : int
Camera
cornerLowerLeft : Vector
horizontal : Vector
vertical : Vector
- CameraId : Guid
- lookFrom : Vector
- lookAt : Vector
- fieldOfView : int
- aspectRatio : double
- aperture : double
+ LookFrom() : Vector
+ LookAt() : Vector
+ FieldOfView() : int
+ AspectRatio() : double
+ Aperture() : double
+ Up() : Vector
# CalculateValues() : void
+ GetRay(u : decimal, v : decimal) : Ray
1
1
1
1..*
1
1
Vector
+ X : int
+ Y : int
+ Z : int
+ Add(vectorToAdd : Vector) : Vector
+ Substract(vectorToSubstract : Vector) : Vector
+ Multiply(multiplicand : decimal) : Vector
+ Divide(divisor : decimal) : Vector
+ AddTo(addedVector : Vector) : void
+ SquaredLength() : decimal
+ Length() : decimal
+ Unit() : Vector
+ DotProduct(vector2 : Vector) : decimal
+ CrossProduct(vector2 : Vector) : Vector
+ RandomInUnitSphere(randomGenerator : int) : Vector
- Negate() : void
+ ToString() : string
1..*
1
1..*
DefocusCamera
- lensRadius : decimal
- focalDistance : decimal
- vectorU : Vector
- vectorV : Vector
+ Random : Random
+ DefocusCamera() : int
+ DefocusCamera()
# CalculateValues() : void
+ GetRay(u : int, v : int) : Ray
NoDefocusCamera
+ NoDefocusCamera()
# CalculateValues() : void
+ GetRay(u : Ray, v : Ray) : void
HitRecord
+ T : decimal
+ IntersectionPoint : Vector
+ Normal : Vector
Ray
+ Origin : Vector
+ Direction : Vector
+ PointAt(t : decimal) : void

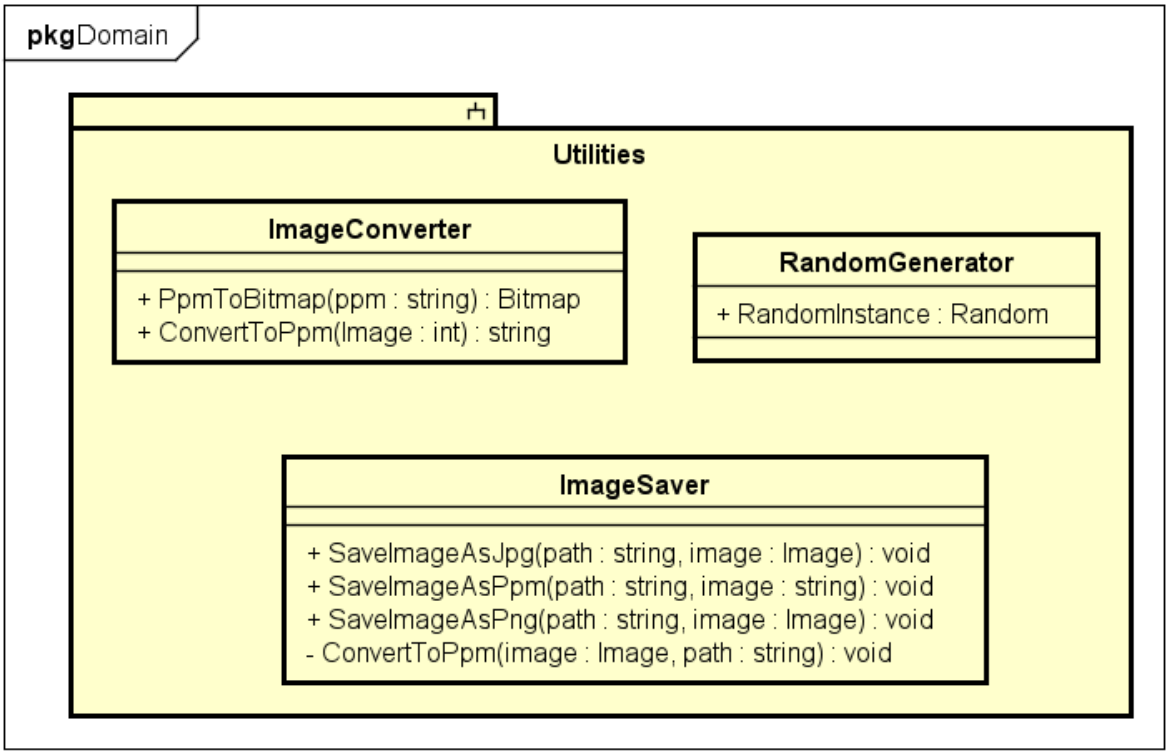


Figura logica (Ir a donde es citada):

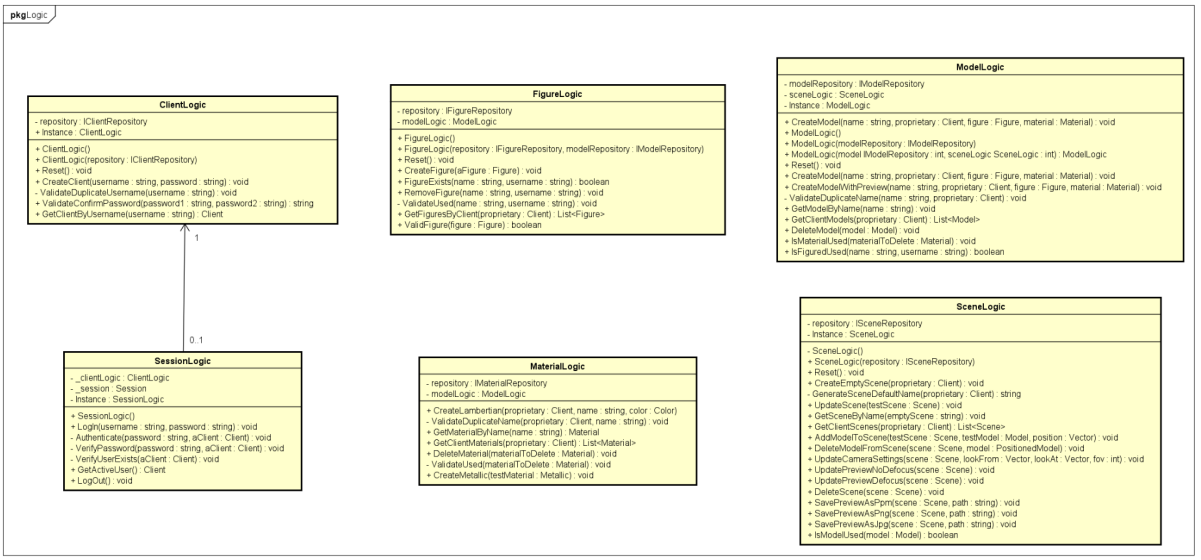


Figura 5.1 (Ir a donde es citada):

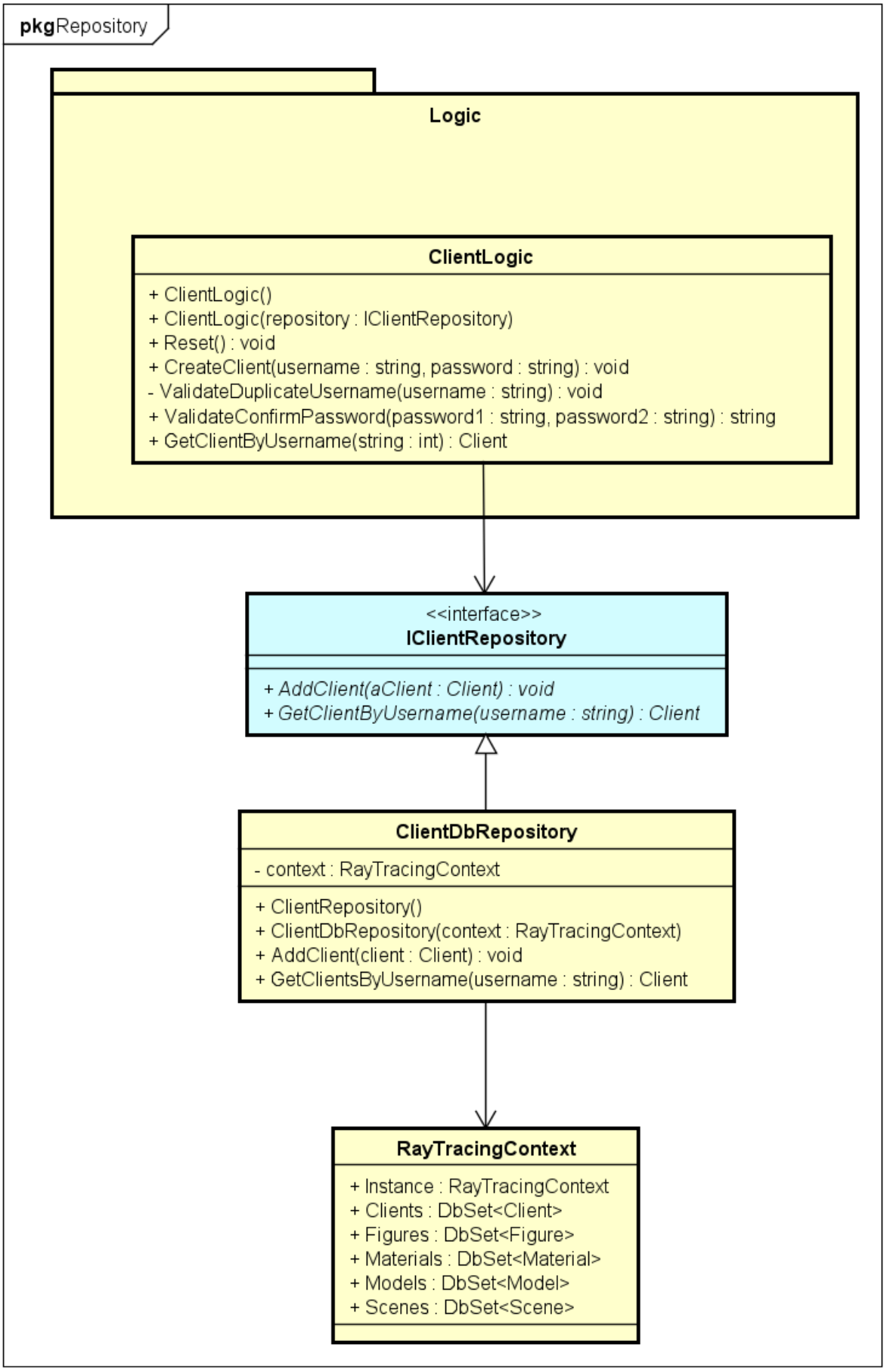

pkgRepository
Logic
ClientLogic
+ ClientLogic()
+ ClientLogic(repository : IClientRepository)
+ Reset() : void
+ CreateClient(username : string, password : string) : void
- ValidateDuplicateUsername(username : string) : void
+ ValidateConfirmPassword(password1 : string, password2 : string) : string
+ GetClientByUsername(string : int) : Client
<<interface>>
IClientRepository
+ AddClient(aClient : Client) : void
+ GetClientByUsername(username : string) : Client
ClientDbRepository
- context : RayTracingContext
+ ClientRepository()
+ ClientDbRepository(context : RayTracingContext)
+ AddClient(client : Client) : void
+ GetClientsByUsername(username : string) : Client
RayTracingContext
+ Instance : RayTracingContext
+ Clients : DbSet<Client>
+ Figures : DbSet<Figure>
+ Materials : DbSet<Material>
+ Models : DbSet<Model>
+ Scenes : DbSet<Scene>

Figura 5.2 (Ir a donde es citada):

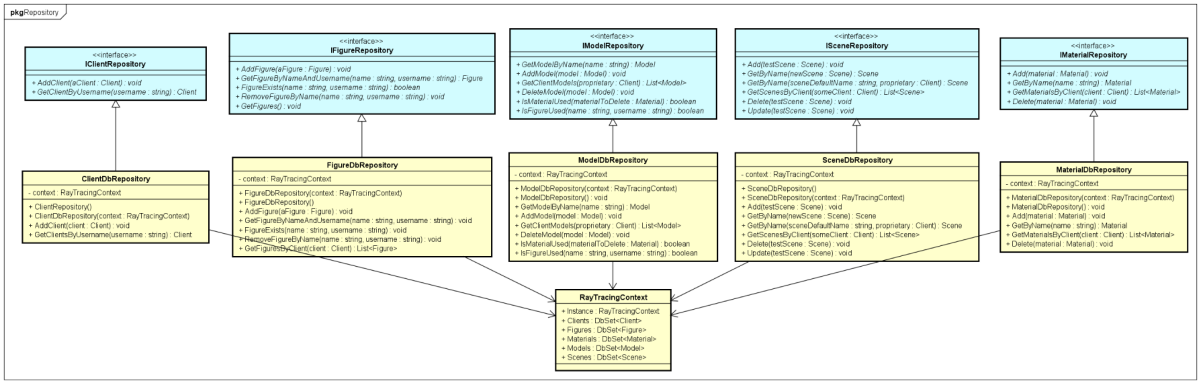
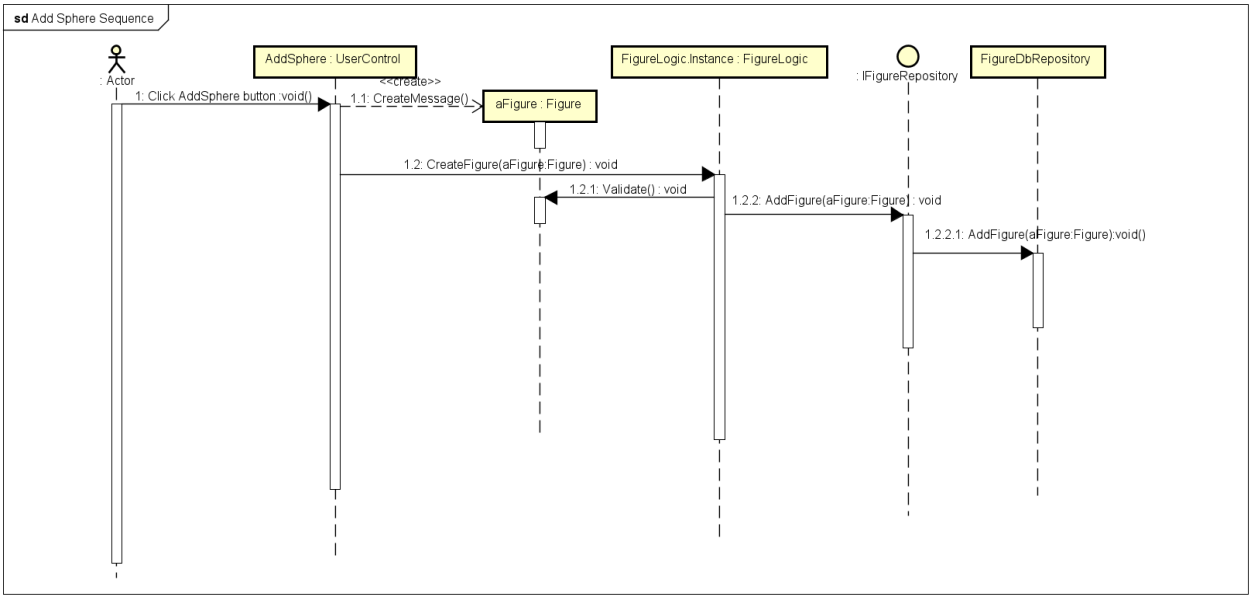


Figura 5.3 (Ir a donde es citada):



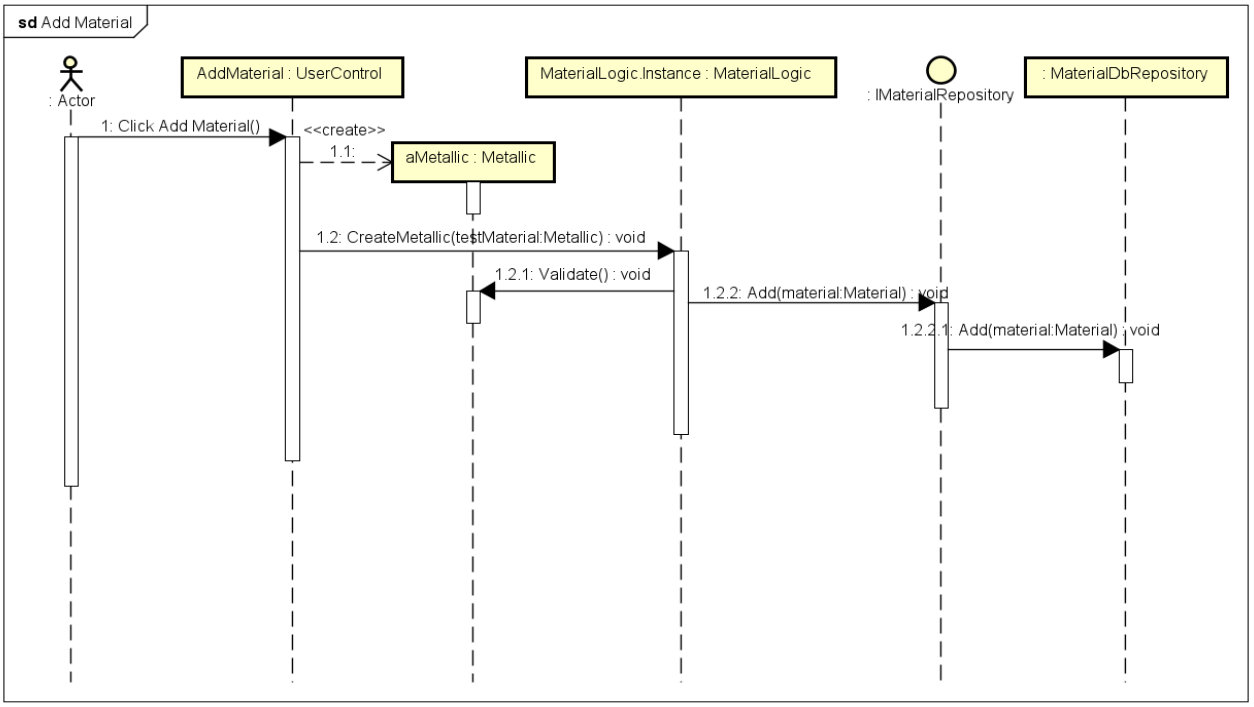


Figura 5.4 (Ir a donde es citada):

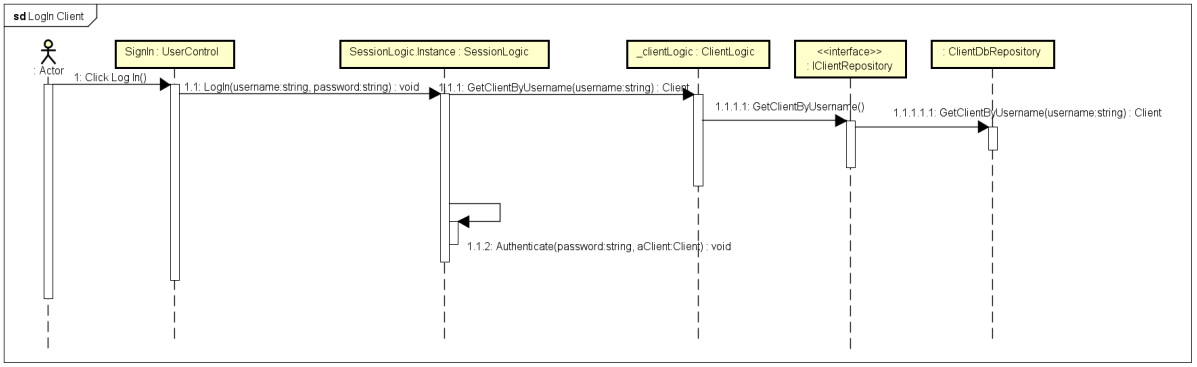


Figura 6 (Ir a donde es citada):

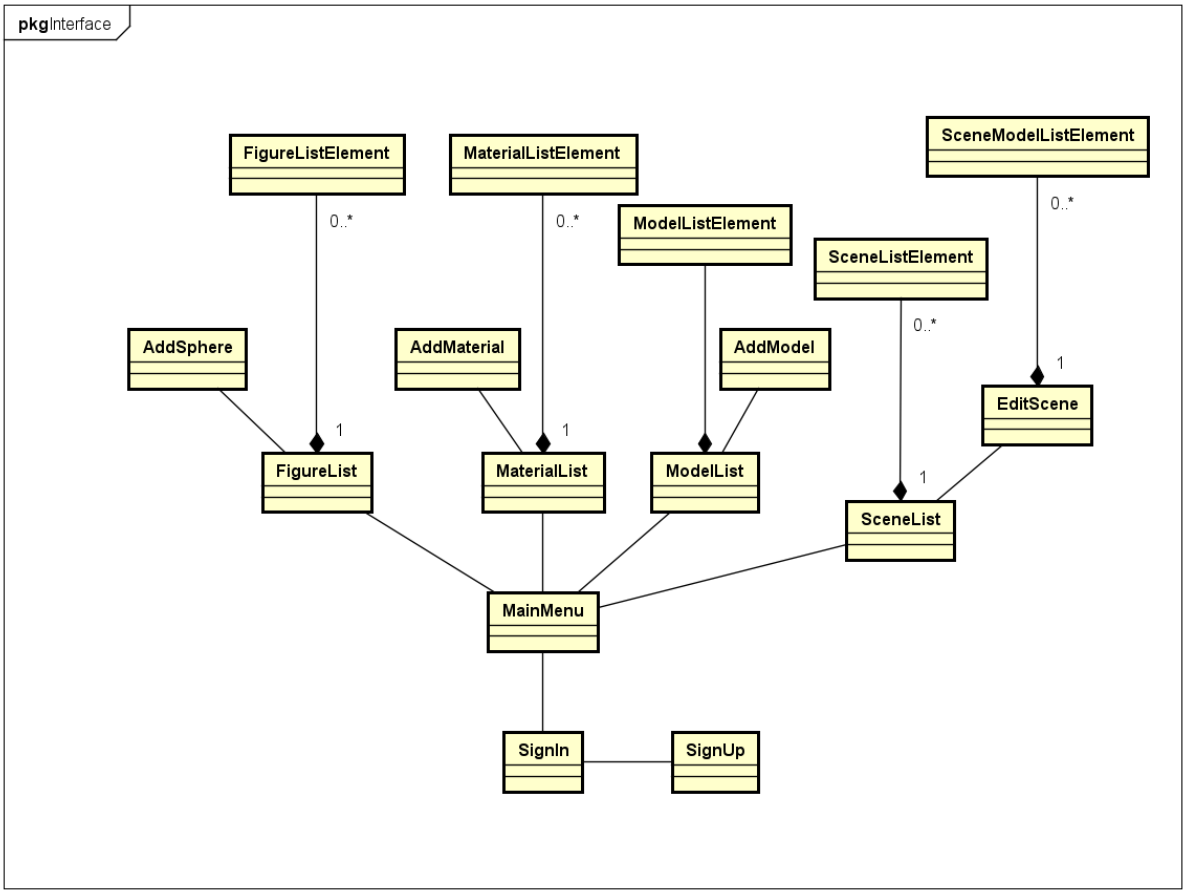
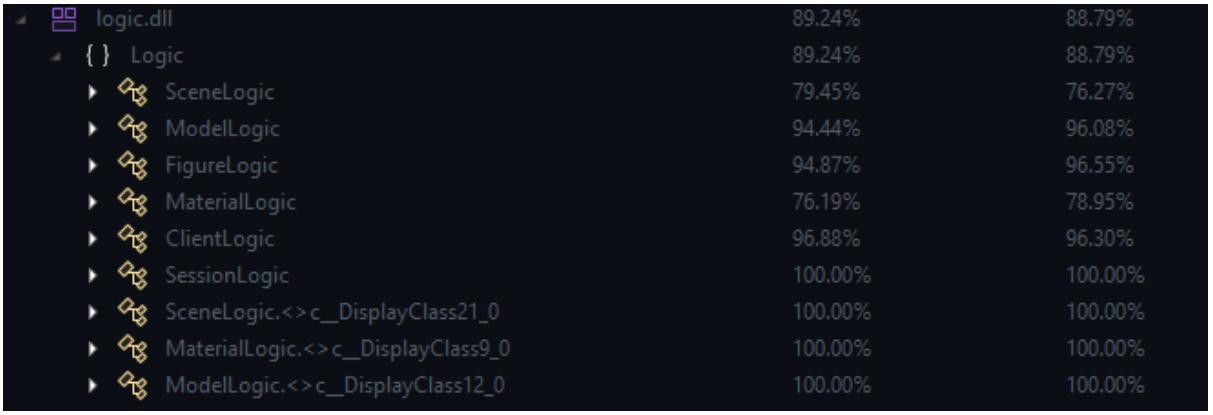


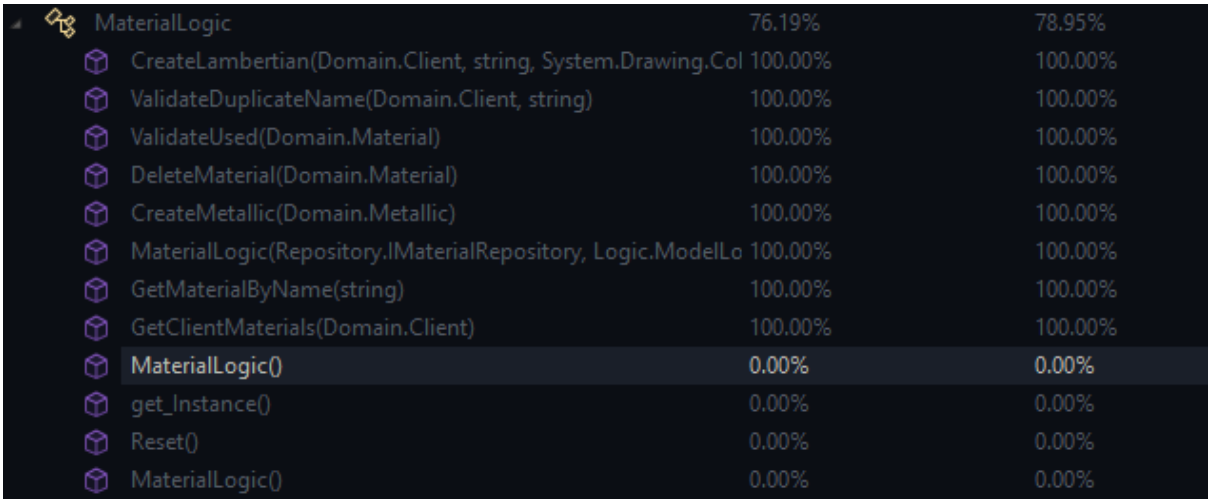
Figura 7 (Ir a donde es citada):

Hierarchy	Covered (%Blocks)	Covered (%Lines)
domain.dll	90.35%	88.01%
{ } Domain.GraphicsEngine	96.56%	94.26%
GraphicsEngine	97.75%	96.67%
Vector	94.15%	93.85%
DefocusCamera	100.00%	100.00%
NoDefocusCamera	100.00%	100.00%
PositionedModel	86.96%	61.11%
Camera	94.29%	93.10%
Ray	100.00%	100.00%
{ } Domain	86.07%	85.62%
Model	90.63%	89.39%
Client	98.78%	97.96%
Scene	58.82%	62.96%
Material	96.61%	93.75%
Sphere	100.00%	100.00%
Metallic	93.48%	91.30%
Figure	96.30%	94.44%
Lambertian	100.00%	100.00%
Session	100.00%	100.00%
{ } Domain.Utilities	61.18%	63.83%
ImageConverter	100.00%	100.00%
RandomGenerator	75.00%	66.67%
ImageSaver	0.00%	0.00%

repository.dll	86.20%	79.36%
{ } Repository	100.00%	100.00%
{ } Repository.Migrations	91.95%	87.23%
{ } Repository.InMemoryRepository	82.88%	81.16%
{ } Repository.DBRepository	83.00%	67.88%
ClientDbRepository	85.71%	75.00%
MaterialDbRepository	81.67%	63.33%
ModelDbRepository	90.55%	71.43%
SceneDbRepository	84.96%	63.64%
FigureDBRepository	73.11%	69.70%
RemoveFigureByName(string, string)	93.55%	75.00%
FigureExists(string, string)	100.00%	100.00%
GetFiguresByClient(Domain.Client)	90.91%	50.00%
AddFigure(Domain.Figure)	80.00%	71.43%
FigureDBRepository()	100.00%	100.00%
FigureDBRepository(Repository.RayTracingContext)	100.00%	100.00%
GetFigureByNameAndUsername(string, string)	0.00%	0.00%



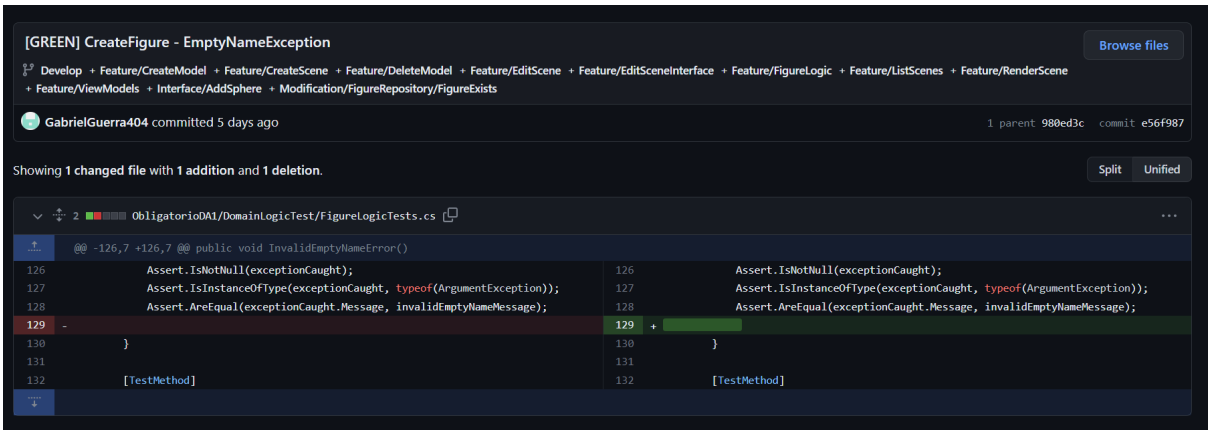
logic.dll	89.24%	88.79%
{ } Logic	89.24%	88.79%
SceneLogic	79.45%	76.27%
ModelLogic	94.44%	96.08%
FigureLogic	94.87%	96.55%
MaterialLogic	76.19%	78.95%
ClientLogic	96.88%	96.30%
SessionLogic	100.00%	100.00%
SceneLogic.<>c__DisplayClass21_0	100.00%	100.00%
MaterialLogic.<>c__DisplayClass9_0	100.00%	100.00%
ModelLogic.<>c__DisplayClass12_0	100.00%	100.00%



MaterialLogic	76.19%	78.95%
CreateLambertian(Domain.Client, string, System.Drawing.Col	100.00%	100.00%
ValidateDuplicateName(Domain.Client, string)	100.00%	100.00%
ValidateUsed(Domain.Material)	100.00%	100.00%
DeleteMaterial(Domain.Material)	100.00%	100.00%
CreateMetallic(Domain.Metallic)	100.00%	100.00%
MaterialLogic(Repository.IMaterialRepository, Logic.ModelLo	100.00%	100.00%
GetMaterialByName(string)	100.00%	100.00%
GetClientMaterials(Domain.Client)	100.00%	100.00%
MaterialLogic()	0.00%	0.00%
get_Instance()	0.00%	0.00%
Reset()	0.00%	0.00%
MaterialLogic()	0.00%	0.00%

SceneLogic	79.45%	76.27%
SceneLogic()	100.00%	100.00%
SceneLogic(Repository.ISceneRepository)	100.00%	100.00%
get_Instance()	100.00%	100.00%
Reset()	0.00%	0.00%
CreateEmptyScene(Domain.Client)	100.00%	100.00%
GenerateSceneDefaultName(Domain.Client)	100.00%	100.00%
UpdateScene(Domain.Scene)	0.00%	0.00%
GetSceneByName(string)	100.00%	100.00%
GetClientScenes(Domain.Client)	100.00%	100.00%
AddModelToScene(Domain.Scene, Domain.Model, Domain.C	100.00%	100.00%
DeleteModelFromScene(Domain.Scene, Domain.GraphicsEng	100.00%	100.00%
UpdateCameraSettings(Domain.Scene, Domain.GraphicsEngi	100.00%	100.00%
UpdatePreviewNoDefocus(Domain.Scene)	0.00%	0.00%
UpdatePreviewDefocus(Domain.Scene)	0.00%	0.00%
DeleteScene(Domain.Scene)	100.00%	100.00%
SavePreviewAsPpm(Domain.Scene, string)	0.00%	0.00%
SavePreviewAsPng(Domain.Scene, string)	0.00%	0.00%
SavePreviewAsJpg(Domain.Scene, string)	0.00%	0.00%
IsModelUsed(Domain.Model)	100.00%	100.00%
SceneLogic()	100.00%	100.00%

Figura 8 (Ir a donde es citada):




Commits on May 6, 2023
[RED] RemoveFigureUsedByModels Exception
GabrielGuerra404 committed 5 days ago
[GREEN] RemoveFigure
GabrielGuerra404 committed 5 days ago
[GREEN] CreateFigure - Name with spaces exception
GabrielGuerra404 committed 5 days ago
[GREEN] CreateFigure - EmptyNameException
GabrielGuerra404 committed 5 days ago
[GREEN] CreateFigure - test CreateFigureAndCheckIfItExists
GabrielGuerra404 committed 5 days ago
[GREEN] CreateSphere - test InvalidRadiusException
GabrielGuerra404 committed 5 days ago
[GREEN] FigureExists - test CheckIfFigureDoesNotExist
GabrielGuerra404 committed 5 days ago