**Universidad ORT Uruguay**
**Facultad de Ingeniería**

# Diseño de Aplicaciones 1
# Obligatorio 1

Entregado como requisito para el Obligatorio 1 de Diseño de Aplicaciones 1

Diego Acuña - 222675
Felipe Briosso - 269851
Nicole Uhalde - 270303

Tutores:
Gastón Mousqués
Facundo Arancet
Franco Galeano

2023

Repositorio: https://github.com/ORT-DA1-2023/222675-269851-270303.git
Enlace video de la aplicación: https://youtu.be/FQ17BtEu1PU

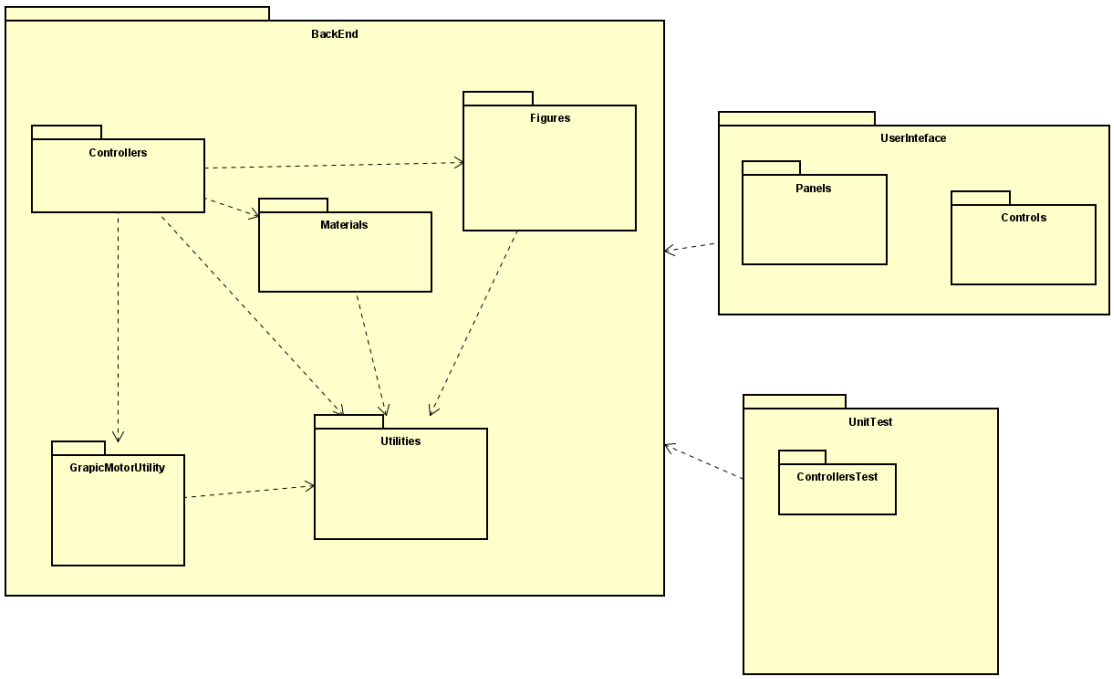
# Índice

## Estructura de la solución

En primer lugar, decidimos organizar la solución en tres proyectos distintos con propósitos muy diferentes y bien definidos. Estos son el Backend, encargado de toda la lógica de la aplicación; los tests del backend y la interfaz de usuario.

Con respecto a la distribución de namespaces, al comienzo se creó un namespace por cada proyecto, de forma tal de mejorar la modularización y el mantenimiento del código. Sin embargo estos namespaces eran demasiado grandes y realizaban múltiples funciones muy diferentes entre sí. Debido a esto, para seguir el principio de responsabilidad única (SRP) y para mejorar la cohesión optamos por hacer subdivisiones en estos namespaces de forma de agrupar las clases que tienen un rol similar.

A continuación, se muestra el diagrama de paquetes de la solución:




Si bien todavía no se mencionó cómo están conformados cada namespace, ya el nombre da una idea del propósito de cada uno de ellos.

## Utilidades

A lo largo de la entrega detectamos que determinadas funciones se estaban repitiendo en muchas clases, por lo que siguiendo el principio de *Don´t repeat yourself* (DRY), optamos por refactorizarlo y extraerlo.

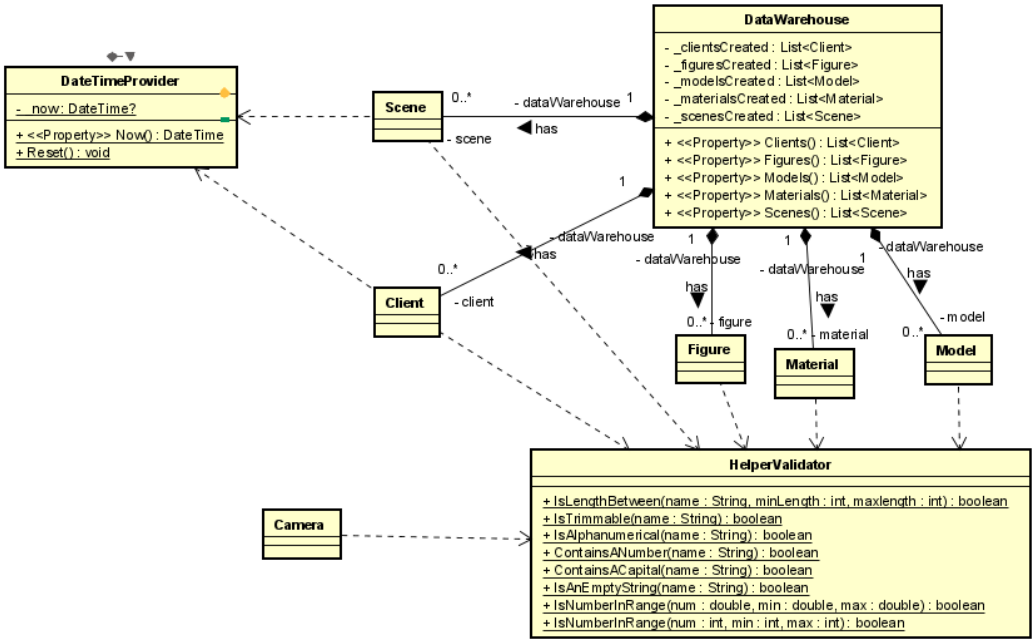
La primera en agregar fue el HelperValidator, que nació debido a que repetíamos mucho código al validar determinados campos. En esta clase se incluyen métodos que ayudan a validar lo necesario para la entrega.

A su vez, como la funcionalidad de estos métodos es de ayuda y no están relacionadas directamente con el estado de los objetos, optamos que todos los métodos de esta clase sean estáticos.

Luego se creó el DateTimeProvider, no tanto con el fin de ahorrar código, sino con la finalidad de poder testear las fechas correctamente. Entendemos que poder fijar la fecha era la forma más correcta de hacer esto.

Por último, se encuentra la DataWarehouse, esta clase se encarga de llevar los registros de todo lo creado.

A continuación, se muestran estas tres clases y cómo se relacionan con otras clases del backend. Con el fin de mejorar la claridad del diagrama y para hacer énfasis en este módulo, ignoraremos los atributos y relaciones de las clases que no pertenecen a este namespace.




En esta imagen se ve claramente que HelperValidator permite evitar duplicaciones significativas de código y que el rol de DataWarehouse será central en el almacenamiento de datos.

## Figure

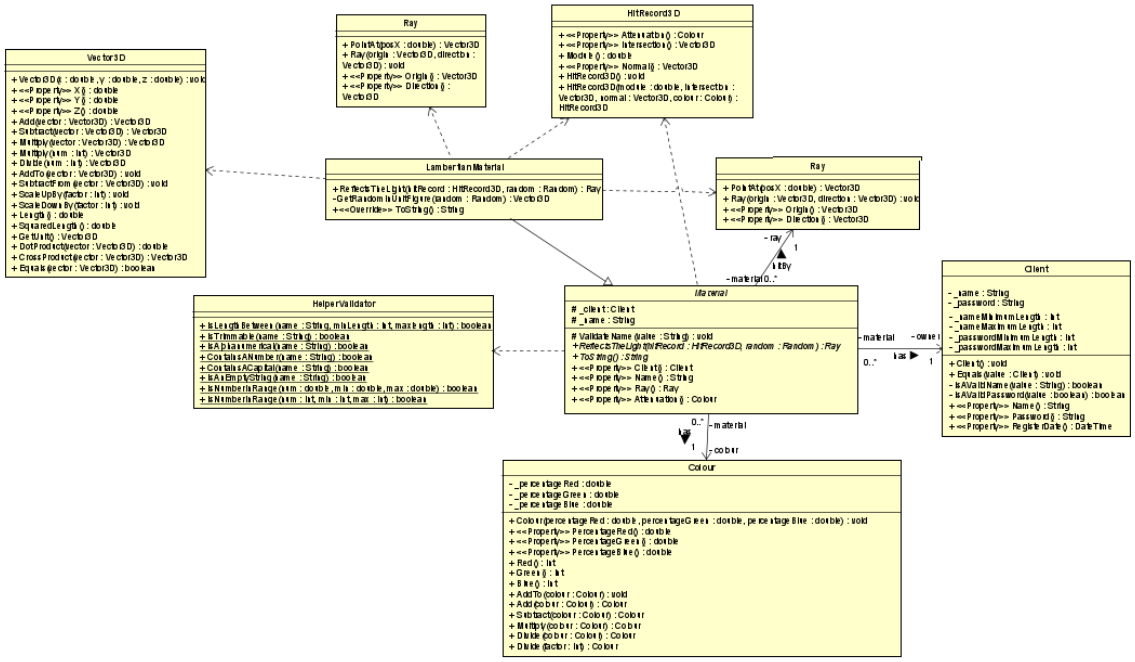
En este namespace se agrupa todo lo relacionado con las figuras. Debido a que esta es una prueba de concepto y en el futuro se podrían agregar nuevas figuras además de la esfera, optamos por utilizar polimorfismo para minimizar el impacto ante cambio. Para ello creamos la clase abstracta

*Figure* que contiene los atributos generales que necesita cualquier figura. A su vez, la clase *Sphere* hereda de *Figure* e incluye algunos atributos propios de ella, siendo el radio un ejemplo de esto.

La mayoría de las clases que integran este namespace son aportadas por el script del matemático aunque optamos por hacer pequeños cambios para mejorar la mantenibilidad del código.

Un ejemplo de esto fue la función que el matemático llama “isSphereHit”, decidimos que vaya en Sphere en vez de Ray debido a que la esfera tiene todos los datos para determinar si un rayo pasa por ella. Perfectamente este método podría haber ido en Ray, pero si en el futuro hay múltiples figuras, la clase Ray quedaría extremadamente larga. A su vez, aprovechamos la herencia que ya habíamos implementado en Figure, de manera tal que la función que se pueda llamar a “isFigureHit” sin importarnos qué tipo de figura es.

A continuación se muestra un diagrama de las clases que pertenecen a este namespace, las relaciones entre ellas y con clases por fuera del namespace (aunque se encuentra simplificado para estas):




Otra decisión que tomamos fue la de representar los vectores y colores en clases distintas. Si bien representan entidades distintas, ambas tienen varios atributos y métodos en común.

Al representar ambas entidades en una clase evitábamos repetir el código en común, pero en ese caso la clase estaba teniendo más de una responsabilidad y poca cohesión. Es por ello por lo que entendimos que la mejor opción era extraer a una nueva clase.

Si bien utilizamos clases distintas, reconocemos que lo óptimo hubiera sido implementar una interfaz, aunque por motivos de tiempo esto no pudo ser posible.

Decidimos que Figure tenga al cliente y no al revés. En la letra del obligatorio se menciona que las figuras conocen a su dueño, y con el fin de evitar dependencias circulares resolvimos que tanto los Modelos, Figuras, Escenas y Materiales conozcan su dueño.

Por el otro lado, el modelo es quien utiliza la figuran dado que es necesaria para poder crearlo. Se tomó la decisión de que el modelo tenga el objeto entero en vez de una referencia como el nombre, tanto como para mantener la lógica de que es un modelo, como para facilitar las búsquedas. A su vez, quisimos seguir el principio de Preserve the Whole Object.

La figura, al igual que los otros objetos están guardados en la DataWarehouse, la cual únicamente posee la lista, es esta junto con el controlador de figura que el usuario accede a cualquier figura y sus atributos, así como guardar, borrar o cambiar.

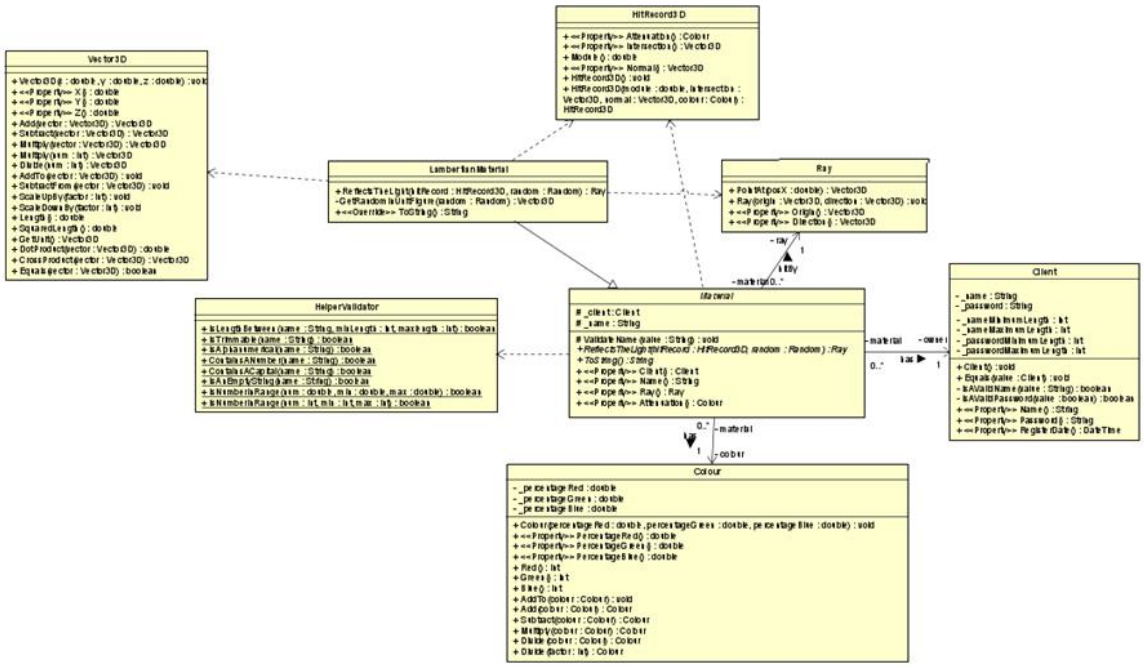
## Materials

Este namespace contiene a las clases Colour, Material y LambertianMaterial.

La primera decisión tomada fue utilizar herencia entre Material y LambertianMaterial con el fin de minimizar el impacto al agregar nuevos materiales.

Otra decisión fue a partir de ver los scripts, distintos materiales reflejan la luz de distinta manera. En el script se menciona el material difuso y en la carátula del obligatorio ya se observa que las propiedades ópticas de distintos materiales varían dependiendo del tipo de material.

Decidimos crear el método abstracto ReflectsTheLight, que indica cómo es el comportamiento de este material al ser atravesado por un rayo. Esto se realizó con la finalidad de que, si en el futuro se agregan nuevos materiales, sería necesario únicamente establecer su comportamiento reflexivo (cómo refleja la luz), minimizando el impacto del cambio.




## GraphicMotorUtility

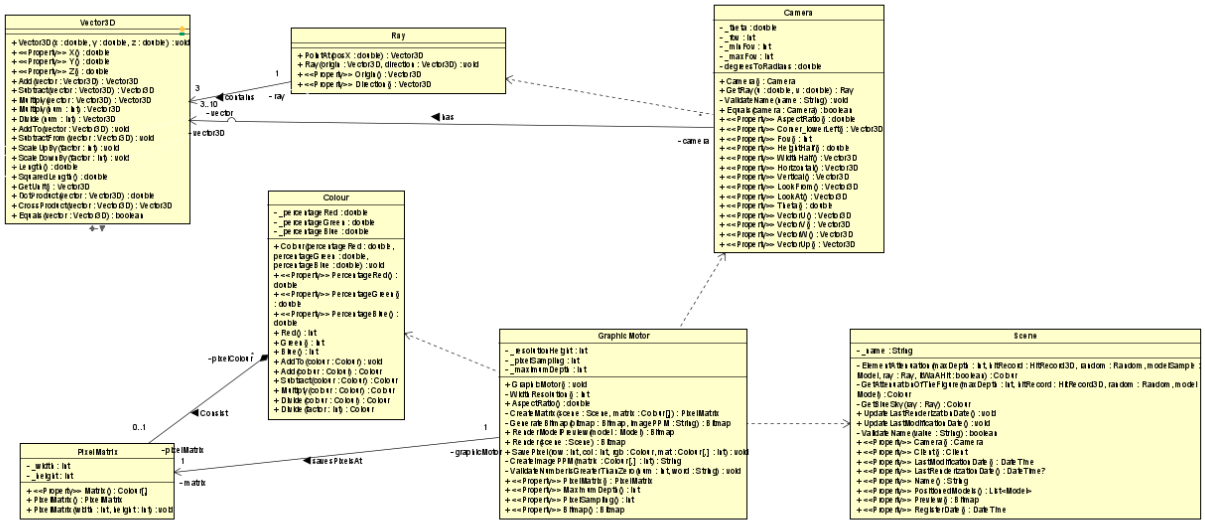
Contiene todas las clases que contribuyen principalmente en realizar el renderizado. La mayoría de estas clases ya provenían de los scripts por lo que no se tomaron mayores decisiones de diseño.

Dentro de la clase GraphicMotor tenemos la función principal que renderiza llamada “Render”, en donde a partir de una escena creada por el usuario llama a otras funciones para lograr el cometido final que es devolver un Bitmap que servirá para mostrar la imagen en la IU. Resumidamente el render transforma la escena en una matriz que se guarda en un objeto PixelMatrix, esto se hace gracias a las funciones y a la matemática brindada por los scripts los cuales pudimos adaptar a C#, manteniendo responsabilidades de clases. Luego de tener la matriz proseguimos a pasar la misma a un string que contenga el contenido del .ppm. Como parte final del renderizado devolvemos el Bitmap mencionado.

En los requisitos del sistema, se puede ver que una escena tiene varios atributos como LookFrom, LookAt o FoV, que los terminamos introduciendo como parte de los atributos de la clase Cámara, esto delega responsabilidad de scene.

Como se mencionó brevemente, separamos las funciones identificadas en el script de .js en las clases donde correspondían, dándole importancia a la responsabilidades de las mismas. Por

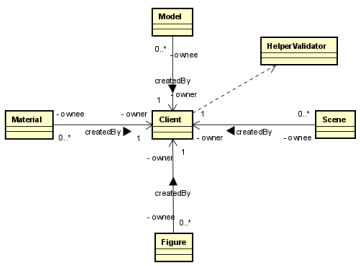
ejemplo, la función ShootRay decidimos que la clase Scene tiene la responsabilidad de tenerla, o por ejemplo que un tipo de material (en este caso por ahora solo tenemos del tipo LambertianMaterial) sepa cuál es el resultado de que se reflecte la luz en él, esto gracias a la función ReflectsTheLight(). Para todas estas funciones mantuvimos que cada una de ellas tenía que hacer una sola acción, y tratamos de reducir las funciones lo más posible. A su vez el render fue programado teniendo en cuenta que en un futuro se pueden agregar o quitar elementos, como materiales, o figuras y este estará preparado.




## Entidades transversales

Entendimos que hay tres clases que se relacionan con la amplia mayoría de clases, estas son Client, Model y Scene.

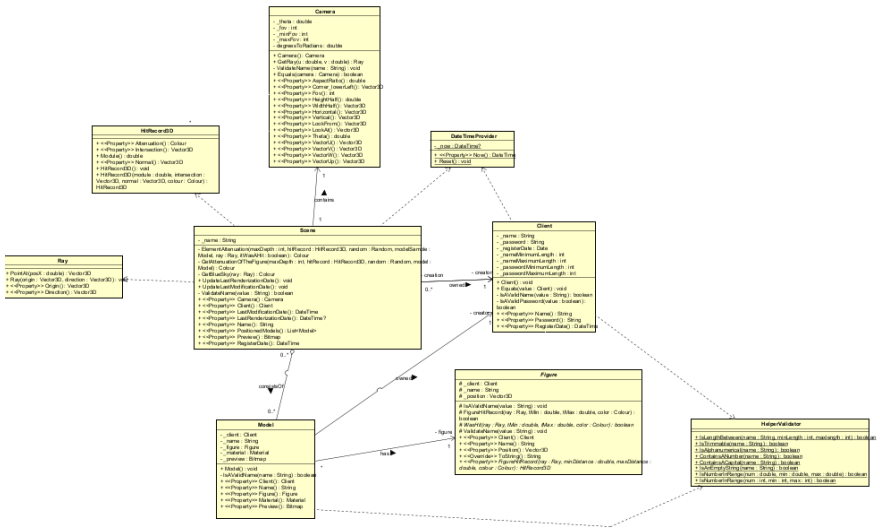
Dentro del backend cliente se relaciona con:



*Aclaración: Escribimos únicamente el nombre de las clases con el fin de no sobrecargar el diagrama.*

Sin embargo, Client se relaciona con clases controladoras, como se mencionará en el siguiente punto.

La clase Scene y Model sus relaciones se presenta a continuación:




Una decisión que tomamos en Scene que no sigue los principios de Clean Code fue haber pasado un booleano en ElementAttenuation. Sabemos que no es lo ideal recibir un booleano porque la función termina teniendo múltiples responsabilidades.

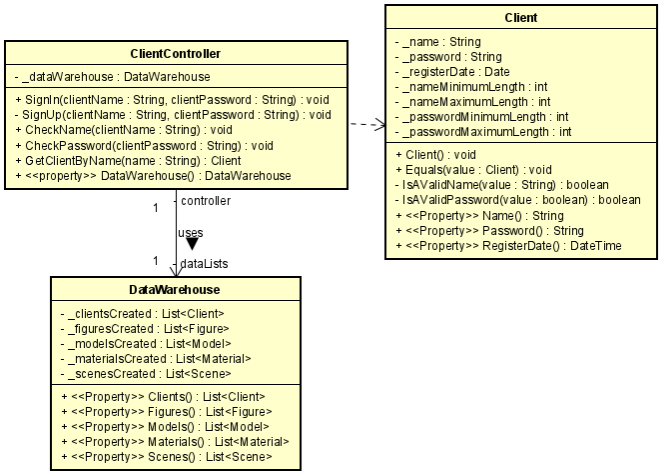
Por otro lado, no es lo ideal en términos de performance que se tenga que calcular de nuevo si hubo un impacto o no. Es por lo mencionado anteriormente que el equipo decidió descuidar este aspecto con el fin de mejorar el proceso de renderizado, que ya de por sí es lento.

## Controladores

En un principio se pensó en una clase intermedia, la cual evitaría la sobrecarga de métodos y responsabilidades. Con eso fue que originalmente se tenía un único controlador el cual había quedado muy sobrecargado. Decidimos partirlo en cinco (uno por cada objecto del backend) y que estos compartieran una única memoria que sería la DataWarehouse.

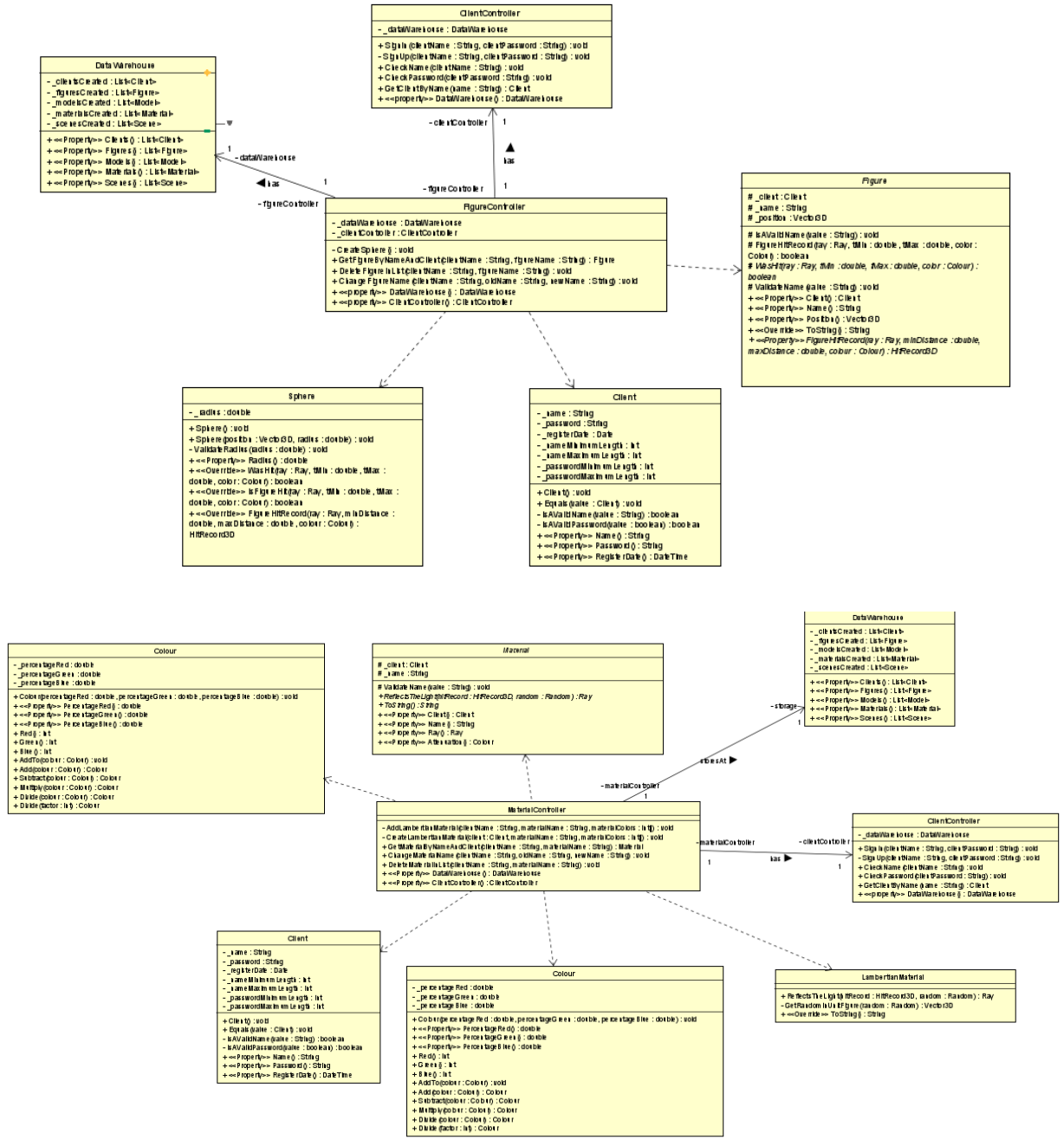
También se había pensado en Data Transfer Objects, pero esto implicaba que alguien en el backEnd se encargara del manejo de objetos y listas, nos parecía más adecuado que la DataWarehouse no tuviera otra función por fuera de tener las listas.

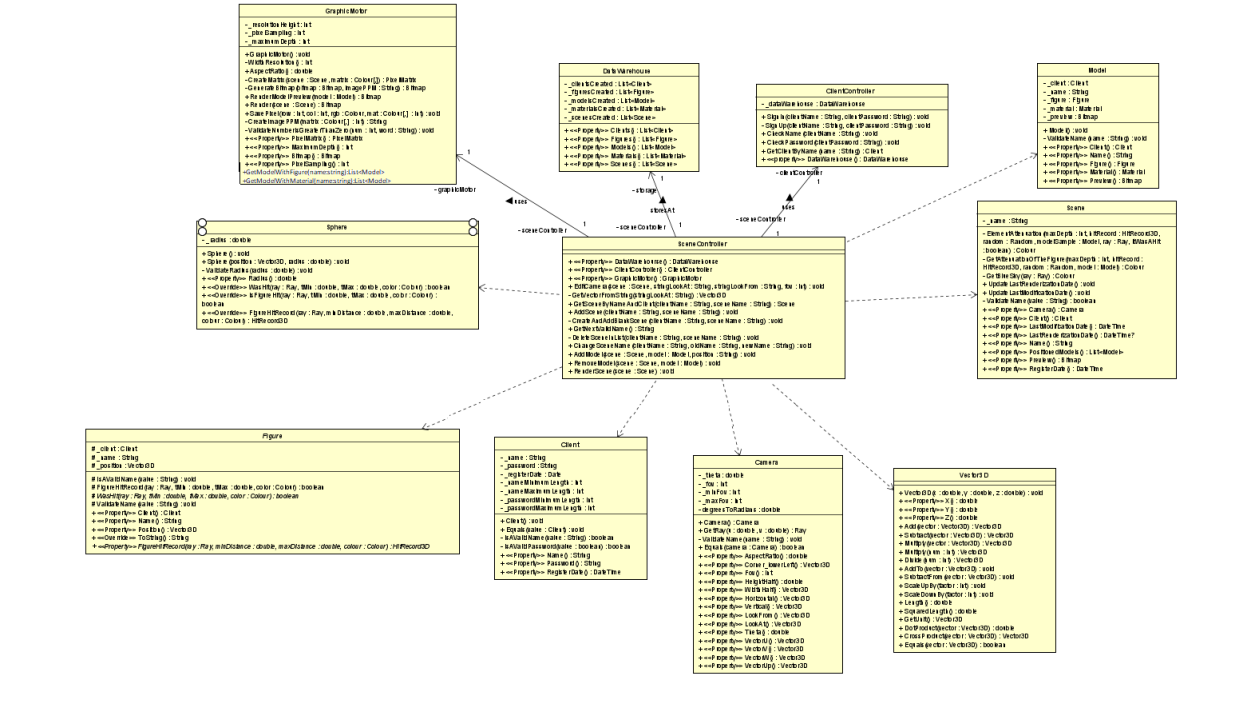
El controlador de Cliente se encarga del manejo de clientes a través de la lista de clientes en la DataWarehouse, a su vez es el que se encarga de tomar y comunicar a la interfaz sobre excepciones o sobre las listas o bien sobre la creación del cliente.

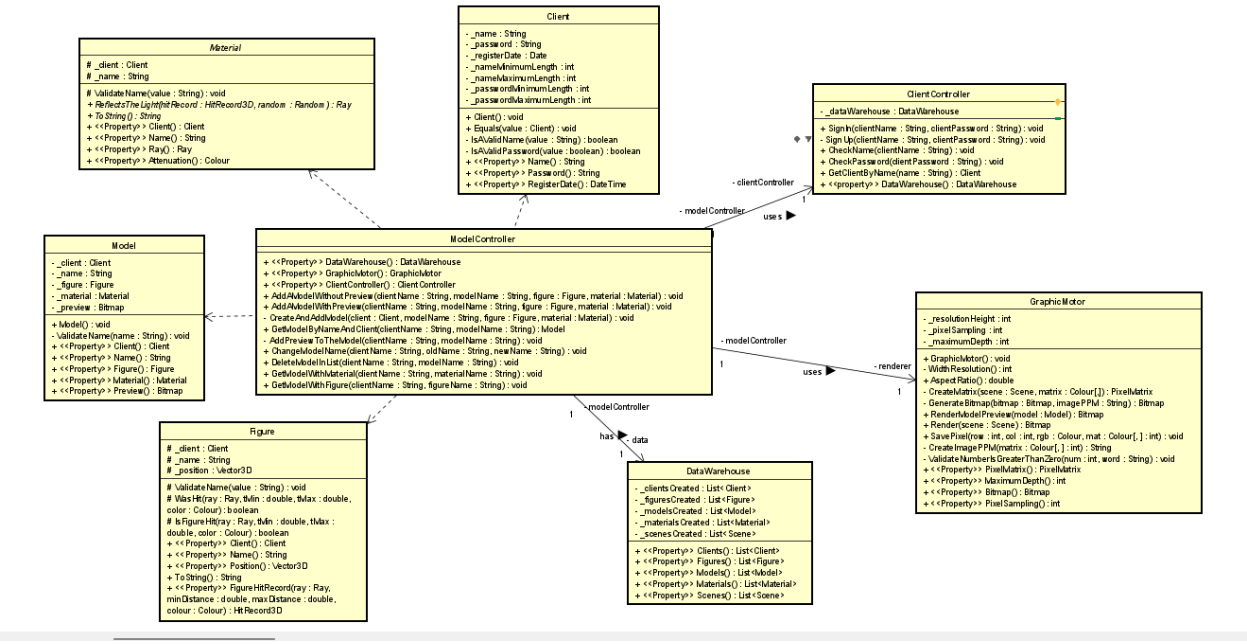



El resto de los controladores funcionan de forma análoga al de clientes.

Nos pareció adecuado marcar las mismas relaciones entre los objetos en los controladores (por ejemplo, la figura, material y modelo conocen a su cliente). De esta forma conseguimos por un lado una responsabilidad más compartida y a su vez que la creación y manejo de objectos no quedara ni en la IU ni en el BackEnd.


ClientController
- _dataWarehouse : DataWarehouse
+ SignIn (clientName : String, clientPassword : String) : void
- SignUp(clientName : String, clientPassword : String) : void
+ CheckName(clientName : String) : void
+ CheckPassword(clientPassword : String) : void
+ GetClientByName(name : String) : Client
+ <<property>> DataWarehouse() : DataWarehouse
DataWarehouse
- _clientsCreated : List<Client>
- _figuresCreated : List<Figure>
- _modelsCreated : List<Model>
- _materialsCreated : List<Material>
- _scenesCreated : List<Scene>
+ <<Property>> Clients() : List<Client>
+ <<Property>> Figures() : List<Figure>
+ <<Property>> Models() : List<Model>
+ <<Property>> Materials() : List<Material>
+ <<Property>> Scenes() : List<Scene>
- clientController
1
has
- dataWarehouse
1
has
- figureController
1
- figureController
1
FigureController
- _dataWarehouse : DataWarehouse
- _clientController : ClientController
- CreateSphere() : void
+ GetFigureByNameAndClient(clientName : String, figureName : String) : Figure
+ DeleteFigure(List(clientName : String, figureName : String) : void
+ ChangeFigureName(clientName : String, oldName : String, newName : String) : void
+ <<property>> DataWarehouse() : DataWarehouse
+ <<property>> ClientController() : ClientController
Figure
# _client : Client
# _name : String
# _position : Vector3D
# IsValidName(value : String) : void
# FigureHitRecord(ray : Ray, tMin : double, tMax : double, color : Colour) : boolean
# WasHit(ray : Ray, tMin : double, tMax : double, color : Colour) : boolean
# ValidateName(value : String) : void
+ <<Property>> Client() : Client
+ <<Property>> Name() : String
+ <<Property>> Position() : Vector3D
+ <<Override>> ToString() : String
+ <<Property>> FigureHitRecord(ray : Ray, minDistance : double, maxDistance : double, colour : Colour) : HitRecord3D
Sphere
- _radius : double
+ Sphere() : void
+ Sphere(position : Vector3D, radius : double) : void
- ValidateRadius(radius : double) : void
+ <<Property>> Radius() : double
+ <<Override>> WasHit(ray : Ray, tMin : double, tMax : double, color : Colour) : boolean
+ <<Override>> IsFigureHit(ray : Ray, tMin : double, tMax : double, color : Colour) : boolean
+ <<Override>> FigureHitRecord(ray : Ray, minDistance : double, maxDistance : double, colour : Colour) : HitRecord3D
Client
- _name : String
- _password : String
- _registerDate : Date
- _nameMinimumLength : int
- _nameMaximumLength : int
- _passwordMinimumLength : int
- _passwordMaximumLength : int
+ Client() : void
+ Equals(value : Client) : void
- IsAValidName(value : String) : boolean
- IsAValidPassword(value : boolean) : boolean
+ <<Property>> Name() : String
+ <<Property>> Password() : String
+ <<Property>> RegisterDate() : DateTime
Colour
- _percentageRed : double
- _percentageGreen : double
- _percentageBlue : double
+ Colour(percentageRed : double, percentageGreen : double, percentageBlue : double) : void
+ <<Property>> PercentageRed() : double
+ <<Property>> PercentageGreen() : double
+ <<Property>> PercentageBlue() : double
+ Red() : int
+ Green() : int
+ Blue() : int
+ AddTo(colour : Colour() : void
+ Add(colour : Colour() : Colour
+ Subtract(colour : Colour() : Colour
+ Multiply(colour : Colour() : Colour
+ Divide(colour : Colour() : Colour
+ Divide(factor : Int) : Colour
Material
# _client : Client
# _name : String
# ValidateName(value : String) : void
+ ReflectsTheLight(hitRecord : HitRecord3D, random : Random) : Ray
+ ToString() : String
+ <<Property>> Client() : Client
+ <<Property>> Name() : String
+ <<Property>> Ray() : Ray
+ <<Property>> Attenuation() : Colour
DataWarehouse
- _clientsCreated : List<Client>
- _figuresCreated : List<Figure>
- _modelsCreated : List<Model>
- _materialsCreated : List<Material>
- _scenesCreated : List<Scene>
+ <<Property>> Clients() : List<Client>
+ <<Property>> Figures() : List<Figure>
+ <<Property>> Models() : List<Model>
+ <<Property>> Materials() : List<Material>
+ <<Property>> Scenes() : List<Scene>
- storage
1
orbiesAt
- materialController
1
MaterialController
- AddLambertianMaterial(clientName : String, materialName : String, materialColors : Int) : void
- CreateLambertianMaterial(client : Client, materialName : String, materialColors : Int) : void
+ GetMaterialByNameAndClient(clientName : String, materialName : String) : Material
+ ChangeMaterialName(clientName : String, oldName : String, newName : String) : void
+ DeleteMaterial(List(clientName : String, materialName : String) : void
+ <<Property>> DataWarehouse() : DataWarehouse
+ <<Property>> ClientController() : ClientController
- materialController
1
has
- clientController
1
ClientController
- _dataWarehouse : DataWarehouse
+ SignIn(clientName : String, clientPassword : String) : void
- SignUp(clientName : String, clientPassword : String) : void
+ CheckName(clientName : String) : void
+ CheckPassword(clientPassword : String) : void
+ GetClientByName(name : String) : Client
+ <<property>> DataWarehouse() : DataWarehouse
Client
- _name : String
- _password : String
- _registerDate : Date
- _nameMinimumLength : Int
- _nameMaximumLength : Int
- _passwordMinimumLength : Int
- _passwordMaximumLength : Int
+ Client() : void
+ Equals(value : Client) : void
- IsAValidName(value : String) : boolean
- IsAValidPassword(value : boolean) : boolean
+ <<Property>> Name() : String
+ <<Property>> Password() : String
+ <<Property>> RegisterDate() : DateTime
Colour
- _percentageRed : double
- _percentageGreen : double
- _percentageBlue : double
+ Colour(percentageRed : double, percentageGreen : double, percentageBlue : double) : void
+ <<Property>> PercentageRed() : double
+ <<Property>> PercentageGreen() : double
+ <<Property>> PercentageBlue() : double
+ Red() : Int
+ Green() : Int
+ Blue() : Int
+ AddTo(colour : Colour() : void
+ Add(colour : Colour() : Colour
+ Subtract(colour : Colour() : Colour
+ Multiply(colour : Colour() : Colour
+ Divide(colour : Colour() : Colour
+ Divide(factor : Int) : Colour
LambertianMaterial
+ ReflectsTheLight(hitRecord : HitRecord3D, random : Random) : Ray
- GetRandomInUnitSphere(random : Random) : Vector3D
+ <<Override>> ToString() : String





## Interfaz de usuario

Para la interfaz de usuario nos basamos en los mockups del obligatorio.

Originalmente el plan era diseñar una interfaz de usuario según los principios de material design y aplicando las heurísticas de Nielsen, por motivos de fuerza mayor eso no fue posible y decidimos hacer énfasis principalmente a las siguientes heurísticas:

- Reconocimiento del estado del sistema: a través de labels que indican cuando una acción fue ejecutada
- Ayuda al usuario a reconocer, diagnosticar y recuperarse de los errores: a través de mensajes de error específicos, indicando cuál es el error.
- Darle al usuario control y libertad: con la implementación de botones para poder editar e ir para atrás cuando sea necesario.

Sin embargo, reconocemos que la usabilidad no es óptima y que podría mejorarse significativamente para futuras ediciones.

Otro defecto es que en el menú contamos con algunos forms que deberían ser controls. Esto es algo que se cambiará en futuras versiones del programa.

## Diseño de la interfaz

La IU consta de algunos defectos, los cuales por temas de tiempo ya no son posibles cambiar para esta versión. Un caso es el menú, en el que contamos con algunos forms que deberían ser controls. Esto es algo que el equipo se compromete a cambiar para futuras versiones del programa.

Por último, nos gustaría marcar el uso del parenting en vez de parámetros pasados entre la misma IU. Esta fue una decisión que tomó el equipo para evitar el uso de parámetros y que los objetos no “paseen” por las ventanas, sino que queden “quietas” y sea la misma ventana la que las va a buscar. Esto nos trajo un problema que es la concatenación de parentings. Pero aun así nos pareció una solución dentro de todo adecuada, que incluso con cambios el impacto no es grande.

## Cobertura de pruebas unitarias

Con respecto al testing, entendemos que la cobertura fue bastante buena. Si bien no logramos llegar al 100%, obtuvimos un 98.19% de cobertura. Entendemos que la cobertura fue muy buena, aunque esperamos que para futuras ediciones esto se pueda subir al 100%.

A continuación, se muestra una imagen de las líneas cubiertas:

| Hierarchy | Covered (Lines) | Not Covered (Lines) | Covered (%Lines) | Not Covered (%Lines) |
|---|---|---|---|---|
| fbrio_FELIPE-PC 2023-05-10 21_58_50.coverage | 2316 | 78 | 96.70% | 3.26% |
| render3d.unittest.dll | 1283 | 59 | 95.53% | 4.39% |
| render3d.backend.dll | 1033 | 19 | 98.19% | 1.81% |
| { } Render3D.BackEnd | 146 | 2 | 98.65% | 1.35% |
| { } Render3D.BackEnd.Figures | 63 | 0 | 100.00% | 0.00% |
| { } Render3D.BackEnd.Utilities | 46 | 0 | 100.00% | 0.00% |
| { } Render3D.BackEnd.Materials | 119 | 0 | 100.00% | 0.00% |
| { } Render3D.BackEnd.GraphicMotorUtility | 307 | 2 | 99.35% | 0.65% |
| { } Render3D.BackEnd.Controllers | 352 | 15 | 95.91% | 4.09% |

Además de destacar el porcentaje global, destacamos que todos los namespaces estén ampliamente testeados, aunque detectamos que el namespace de controladores fue levemente menos testeada.

Con respecto al desarrollo de tests, se buscó que fueran exhaustivos y que cubran tanto los happy paths como los sad paths.

## Bugs/debilidades detectadas y posibles mejoras

### Externos:

Luego de un arduo proceso de testing exploratorio, encontramos que existen casos muy particulares en el que se obtiene un resultado erróneo. Si bien pensamos que se debía a un error nuestro, luego de correr el script detectamos que era un problema en el código del matemático.

Este error ocurre cuando se ejecuta el script número 8 con la siguiente configuración:

```
var pixels = [];
var lookAt = new Vector(0, 0 , 0);
var vectorUp = new Vector(0, 1, 0)
var origin = new Vector(0, 5, 0);
var camera = new Camera(origin, lookAt, vectorUp, 40, aspectRatio);

var sphere = new Sphere(new Vector(0, 0, 0), 1, new Vector(0.1, 0.2, 0.5));
var terrain = new Sphere(new Vector(0, -2000, -1), 2000, new Vector(0.7, 0.7,
0.1));
var elements = [sphere, terrain];
```

Al ejecutarse este código, la matriz resultado es una matriz en donde todas las entradas son NaN.

En nuestra implementación el error emerge al llamar a *getUnit* con el vector nulo que termina cayéndose al querer cargar un píxel con un número que no esté entre 0 y 255.

### Propios:

Entendemos que nuestra principal deficiencia fue el diseño de la interfaz de usuario. Reconocemos que el diseño de una IU intuitiva y fácil de usar no fue nuestra prioridad. Por lo que esperamos para una futura edición poder mejorarlo.

Otro aspecto relacionado a la IU que nos gustaría mejorar es que la página no sea adaptable (responsive), que, si en algún input se pone un nombre muy largo, este puede ser tapado por botones.

Por último, queremos recalcar que cuando se intenta agregar un modelo a una escena, en la interfaz aparece el nombre del modelo seguido por (0,0,0). Esto se debe a que por defecto el modelo es posicionado en el origen y por ende se muestra eso.

Entendemos que no es información que necesite conocer al usuario, sino que podría llegar a confundirlo, por lo que lo tomaremos en cuenta para la entrega número dos.

También en algunos casos no tuvimos tiempo para refactorear, como por ejemplo en esfera, que en la función FigureHitRecord se calcula Bhaskara. Lo ideal hubiera sido extraer dicha operación a una función auxiliar, aspecto que será arreglado para la siguiente entrega.