Software de obtención y procesamiento de datos a tiempo real con CUDA para sistema de interferometría

Anexo III: Especificación de Diseño

Proyecto de Fin de Carrera
INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Septiembre de 2012

Autor
Álvaro Sánchez González

Tutor
Guillermo González Talaván

Cotutor
Francisco Valle Brozas

Contenido

Índice de Figuras

1. Introducción

Este tercer anexo recoge la especificación de diseño del software.

Tras la especificación de requisitos y el modelo de análisis en el dominio del problema, ha llegado la hora de determinar las estrategias de diseño concretas que se van a usar para resolver los problemas planteados, y además, comenzar a tener en cuenta los requisitos no funcionales, es decir, pasar a trabajar en el dominio de la solución.

En los siguientes apartados se pueden encontrar algunas de las diferentes vistas usadas para el diseño de la aplicación. Se describirá:

- Una vista arquitectónica que permite obtener una concepción general del la estructura y organización del programa.
- Una vista estática que muestra todas las clases, tanto de la parte central de la aplicación, como de la interfaz, y sus relaciones entre sí.
- Una vista de diseño de datos que explica los objetos de almacenamiento de datos de más relevancia así como el almacenamiento de datos en ficheros.
- Una vista de interacción que muestra la secuencia temporal de llamadas entre los distintos objetos que controlan las partes más delicadas del programa, en las que se pueden apreciar aspectos fundamentales del diseño.

2. Diseño arquitectónico

2.1. Modelo-Vista-Controlador (MVC)

Una de las alternativas más usadas para este tipo de programas es el patrón Modelo Vista Controlador (En adelante MVC).

En el MVC se sigue un esquema como el siguiente:

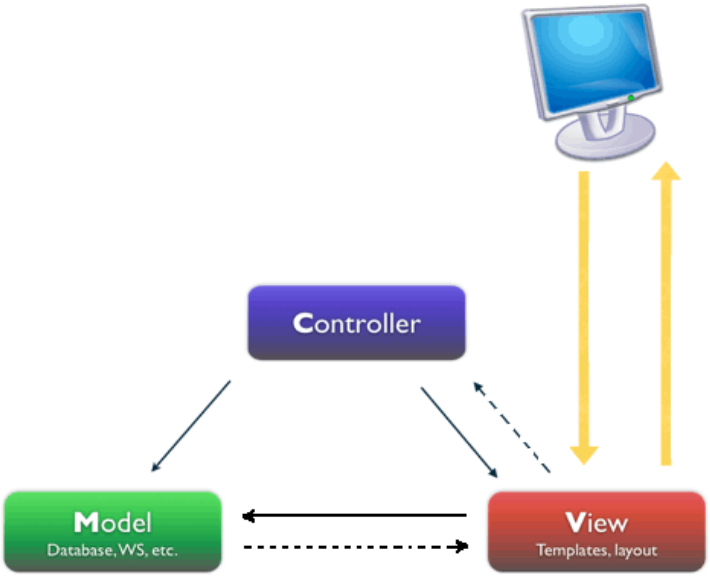

Figura 1. Modelo Vista Controlador

En este modelo, hay tres figuras importantes:

- La vista: se encarga de mostrar los elementos que correspondan. Es además la parte con la interacciona el usuario.
- El controlador: es el encargado de recibir los eventos del modelo o de la vista, y de actualizar el modelo o la vista.
- El modelo: encapsula los datos y la funcionalidad central.

Las flechas representan el trasvase de información entre los distintos elementos del patrón. Si la flecha es continua indica que hay dependencia directa, mientras la flecha discontinua indica que no hay dependencia directa, es decir, información que se recibe

al salir de un método (La clase que implementa el método no tiene porqué conocer a la clase que lo llama), o bien mediante un sistema de señales/slots de Qt que permite transferir eventos sin que el emisor de la señal tenga que conocer al receptor.

2.2. Estrategia utilizada: Modificación del MVC

En este caso, se ha utilizado una modificación del patrón MVC que permite eliminar la dependencia entre el modelo y la vista.

El esquema utilizado ha sido el siguiente:

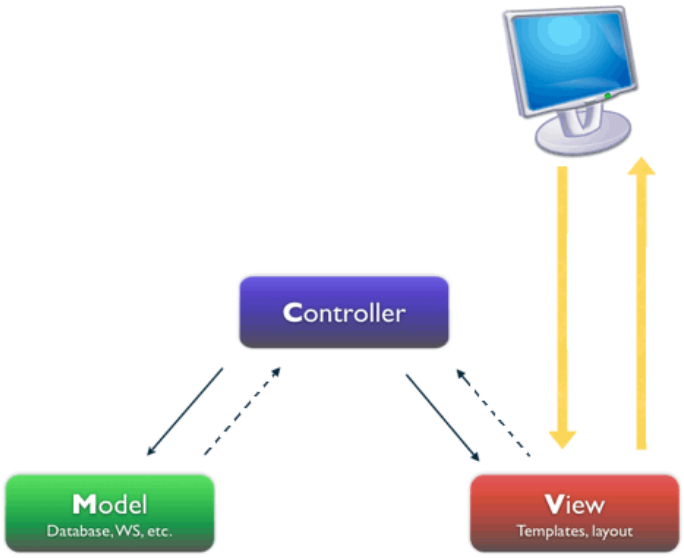

Figura 2. Modelo Vista Controlador modificado

La principal ventaja es la eliminación de la dependencia entre el modelo y la vista. Esto hace mucho más fácil cualquier cambio tanto en el modelo, como en el controlador, o como en la vista, ya que, el modelo no depende ni de la vista ni del controlador, la vista no depende ni del controlador ni del modelo, y es el controlador el único que depende, por tanto, de la vista y del modelo.

Exactamente, el cometido de cado uno de los componentes, en este patrón modificado, y para el problema concreto, es el siguiente:

- Vista: Será una clase generada automáticamente por el diseñador de Qt. Contiene todos los visualizadores y controles y se encarga de enviar señales cada vez que se produzca una acción por parte del usuario. Un ejemplo en la aplicación es Ui::MainWindow.
- Controlador: Existe uno para cada vista. En general, dada esa relación uno a uno con la vista, tiene el mismo nombre que la vista asociada (Aunque en distintos espacios de nombres). Esta clase sí está directamente codificada por el programador. Se encarga de recibir en sus slots todas las señales procedentes de la vista, y en base a ello modificar el modelo o la vista según proceda. Un ejemplo en la aplicación es MainWindow.
- Modelo: Contiene la lógica de cálculo de la aplicación, así como acceso a todos los datos. Recibe peticiones del controlador.

  En la aplicación, el modelo contiene a su vez otro controlador, el controlador central de la aplicación (MainController), a través del que se accede a todos los datos modelos o a otros controladores con acceso a ellos.

Por último, veamos un ejemplo con clases reales de la aplicación usando el patrón modificado, incluyendo la arquitectura interna dentro del modelo:

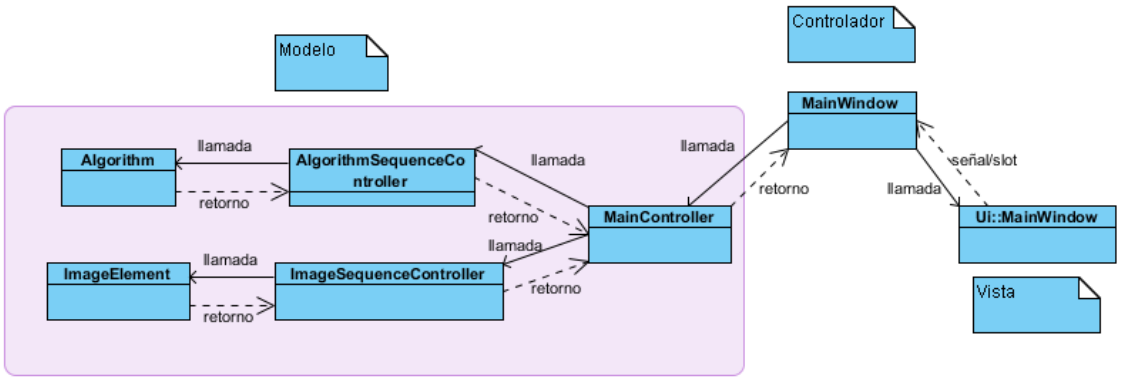

Figura 3. Ejemplo de MVC modificado

2.3. Diagrama de capas

Y como último elemento dentro de esta vista arquitectónica, se muestra un diagrama de capas del sistema con los paquetes de la aplicación sus dependencias entre sí:

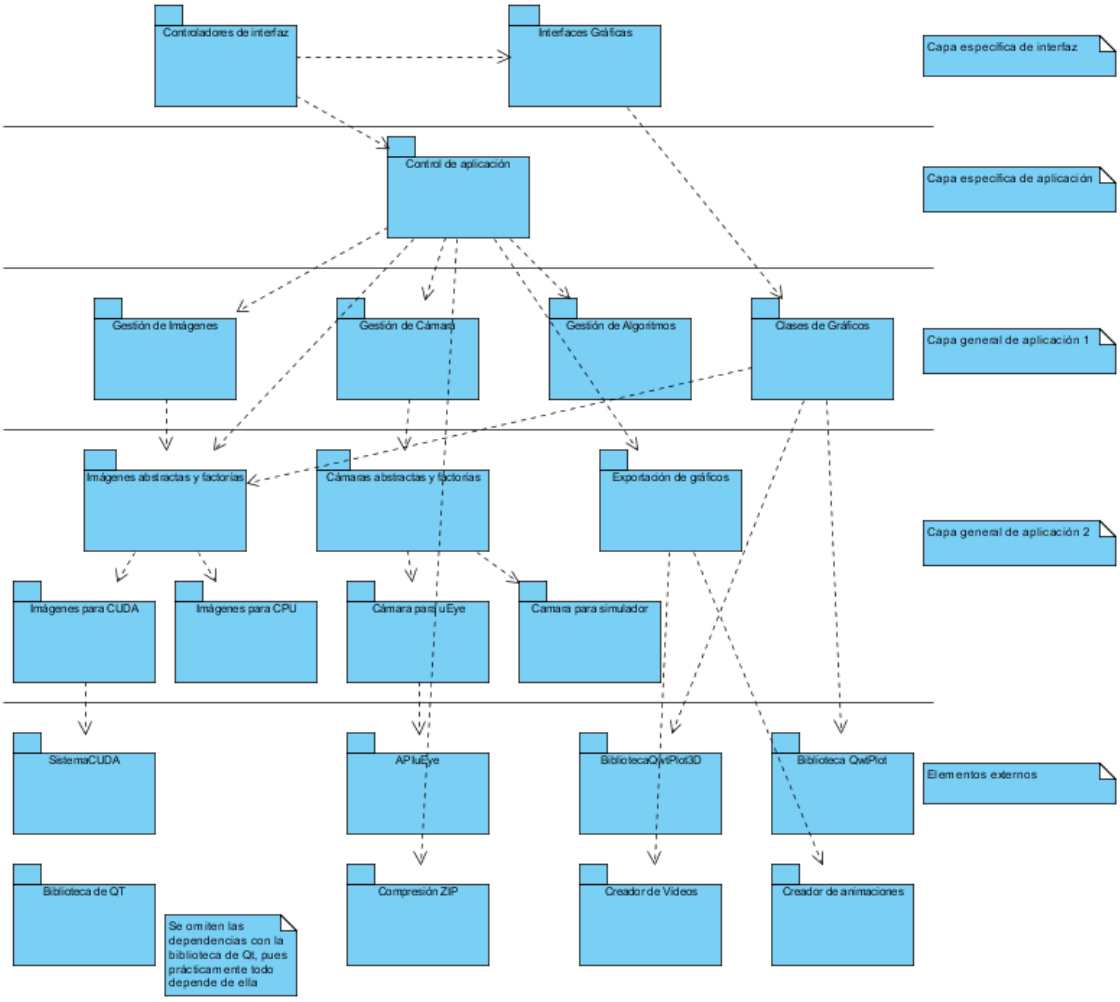

Figura 4. Diagrama de capas del sistema

- La **capa específica de la interfaz** contiene las clases relacionadas con las ventanas y diálogos que se muestran en la aplicación. Es específica de la aplicación, y por tanto no sería muy reutilizable.
- La **capa específica de aplicación** contiene esencialmente el controlador principal, que sirve de nexo, entre los distintos módulos inferiores y la interfaz. Es específica de la aplicación, y por tanto no sería muy reutilizable, aunque permitiría un cambio completo de la interfaz, y sería transparente a cambios en la lógica de los módulos inferiores.
- La **capa general de la aplicación 1** contiene los módulos independientes entre sí que han sido creados para su uso en la aplicación, pero que podrían ser fácilmente reutilizables para aplicaciones relacionadas.

- La **capa general de la aplicación 2** contiene otra serie de módulos, también fácilmente reutilizables que encapsulan el trabajo con alguna herramienta o biblioteca, o clases de cálculo puro.
- La capa de **elementos externos** contiene todas las bibliotecas y herramientas, no creadas por el desarrollador, que se utilizan para añadir funcionalidad al programa.

Dentro de cada paquete, se pueden encontrar las siguientes clases:

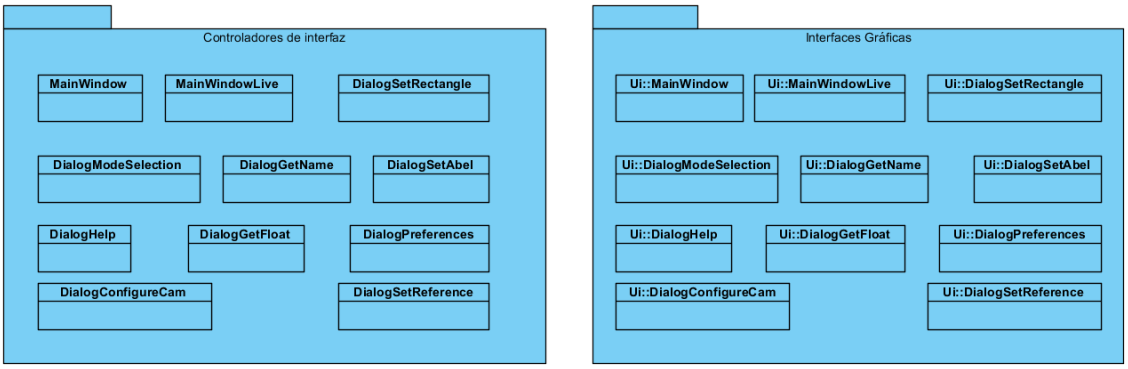

Figura 5. Paquetes: Capa específica de interfaz

Figura 6. Paquetes: Capa específica de aplicación

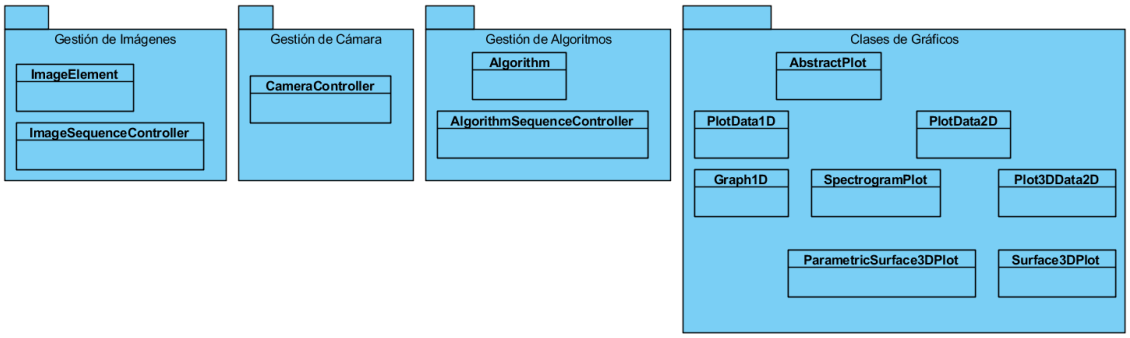

Figura 7. Paquetes: Capa general de aplicación 1

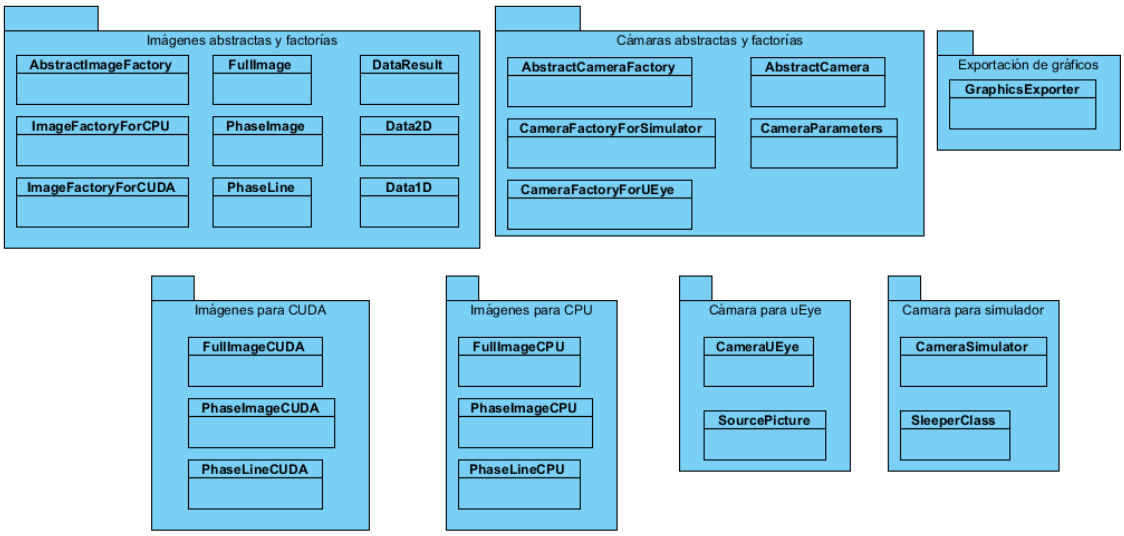

Figura 8. Paquetes: Capa general de aplicación 2

3. Vista estática

En este apartado se muestra la vista estática de la aplicación, es decir, las clases, y las conexiones que existen ellas durante la ejecución normal del programa, dejando a un lado la interacción.

Dado que ya se están representando las mismas clases que se utilizarán en la implementación, los nombres utilizados para ellas serán los mismos que los utilizados en el código.

3.1. Vista a primer nivel del núcleo de la aplicación

Se comienza con un diagrama a primer nivel (Sin mostrar todas las clases) de las clases que conforman el núcleo del programa, sin considerar la interfaz. Más adelante se desarrollará cada una de las partes del diagrama.

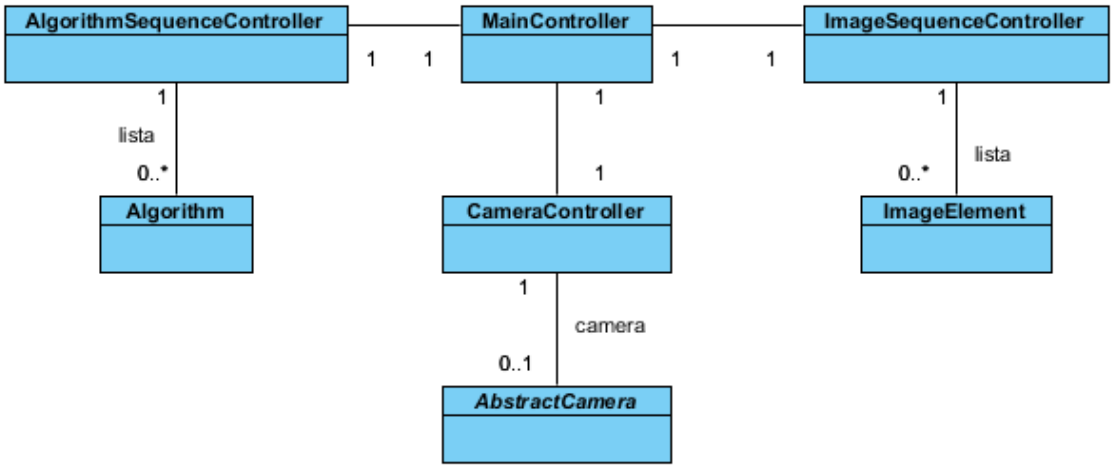

Figura 9. Vista del núcleo a primer nivel

Se puede observar a primer nivel que el programa está gobernado por un controlador principal (MainController) que tiene acceso a los diferentes elementos fundamentales del programa, como es la lista de algoritmos (Algorithm), a través de un controlador de algoritmos que gestionará esa lista (AlgorithmSequenceController); la lista de imágenes de fase (ImageElement), a través de un controlador de imágenes

(ImageSequenceController); y la cámara (AbstractCamera), a través del controlador de la cámara (CameraController).

3.2. *Vista a segundo nivel del núcleo de la aplicación*

Una vez mostrado el primer nivel de organización del núcleo se pasa a detallar, en un segundo nivel, cada uno de los elementos antes mostrados.

3.2.1. Controlador principal

Además de las clases mostradas en la Figura 9, el controlador principal está rodeado de otras clases, necesarias para la ejecución del programa, como se puede ver a continuación:

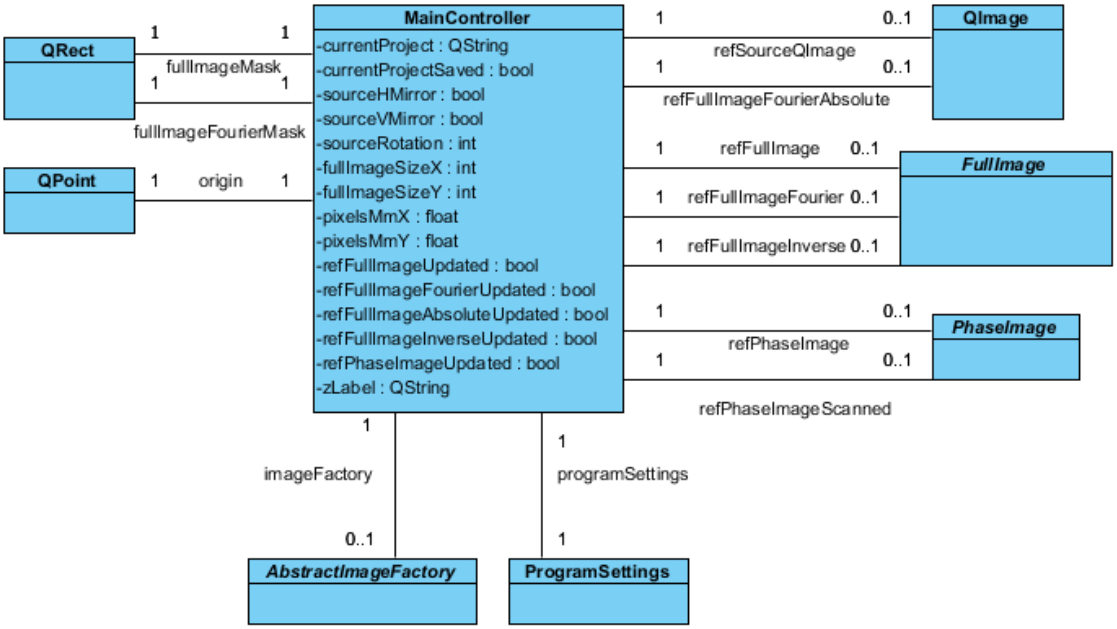

Figura 10. Vista de segundo nivel: Controlador principal

Comenzando por los atributos, los 2 primeros, contienen la ruta del proyecto abierto, y el estado de guardado del proyecto en memoria secundaria.

Los 7 siguientes, junto con los objetos de la clase QRect y QPoint forman parte de los parámetros necesarios para realizar los cálculos adecuadamente.

Los siguientes 5 atributos indican el estado de actualización de los objetos FullImage y PhaseImage relacionados. Estos objetos son productos del primer procesado de la referencia.

Por otra parte, las relaciones con la clase QImage, se corresponden a la imagen de referencia, y a una imagen con el valor absoluto de la transformada de Fourier, para su posible uso en una vista previa.

Otro detalle importante es la relación del controlador principal con una factoría de imágenes (AbstractImageFactory) cuya utilidad será manifestada en el capítulo 5.1.1.

Y para terminar, existe una relación con un objeto tipo ProgramSettings, que contiene toda la información de preferencias del programa, que será persistente de una ejecución a otra.

3.2.2. Controlador de cámara

El controlador de la cámara se encarga de todo lo relacionado con el manejo y configuración de la cámara:

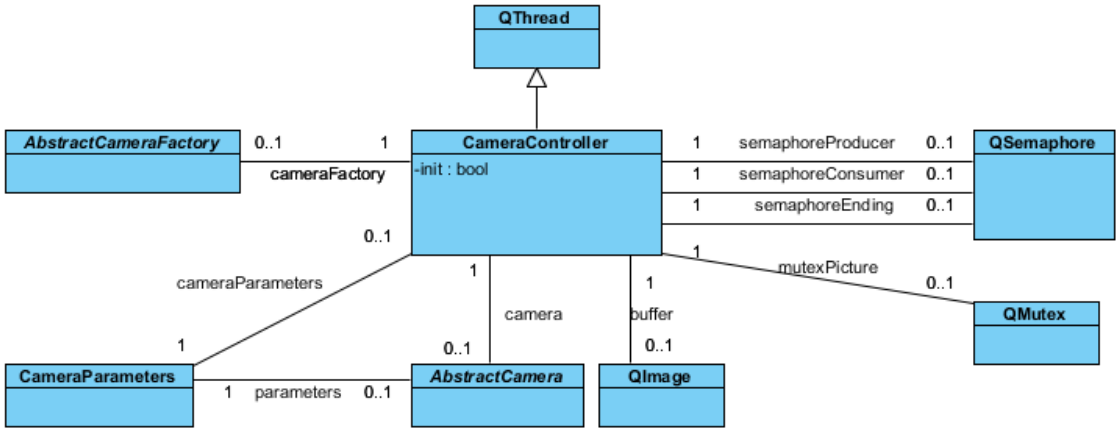

Figura 11. Vista de segundo nivel: Controlador de cámara

Como se puede observar el controlador deriva de QThread. Esto le permitirá realizar operaciones en un hilo independiente al de ejecución normal. Además para controlar la concurrencia en estas operaciones, existen tres semáforos y un mutex con los que

también se relaciona el controlador. El porqué de este trabajo en paralelo se pondrá de manifiesto en el capítulo 5.2.2.

Por otra parte, el controlador se relaciona con una factoría de cámaras, cuya finalidad también se explicará en el capítulo 5.2.1, y con una cámara creada con la factoría.

Tanto el controlador como la cámara están relacionados con un objeto de la clase de parámetros. La diferencia: el objeto relacionado con el controlador contendrá los parámetros deseados para la cámara, mientras que el relacionado con la cámara, contendrá los parámetros reales.

Y para finalizar, está relacionado con un objeto QImage que será una imagen tomada por la cámara. Una de las misiones del controlador consiste en proporcionar una imagen lo más rápidamente posible cuando alguien se lo pida. Ése buffer contendrá la imagen que se devolverá.

3.2.3. Controlador de imágenes

El controlador de imágenes se encarga de gestionar las imágenes de la fase actualmente abiertas en el programa. Deriva de una clase de lista, añadiendo métodos de gestión, y sus relaciones con otras clases se pueden ver en la Figura 12.

La clase principal relacionada con el controlador es la clase donde se almacenan las imágenes de la fase: ImageElement.

La clase ImageElement contiene varios atributos relacionados el nombre, la posición y las dimensiones de la imagen que contiene, así como un atributo que indica el estado de actualización de la imagen.

Está relacionada con un objeto tipo PhaseImage, a través de phaseImageSource que representa la imagen de la fase, justo después de preprocesar la imagen con la referencia, y antes de aplicar los algoritmos. Ese objeto será el punto de partida sobre el que se a aplicará la segunda fase del procesado: aplicar los algoritmos.

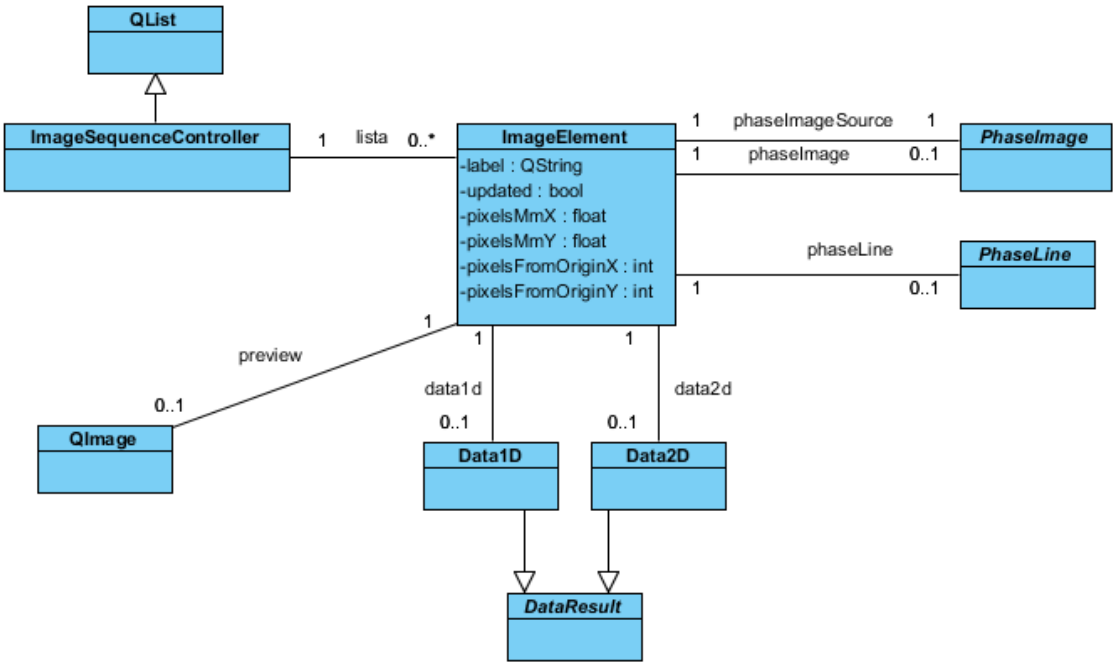

Figura 12. Vista de segundo nivel: Controlador de imágenes

A continuación existen otras dos relaciones una de nuevo con PhaseImage y otra con PhaseLine. Estos objetos representarán la imagen después de aplicar la lista de algoritmos, y su estado de actualización dependerá del parámetro "updated" de ImageElement.

Después aparecen las relaciones data1d y data2d. Estos son objetos de las clases Data1D y Data2D orientados a la representación gráfica o exportación de las estructuras de datos PhaseImage y PhaseLine respectivamente, por lo que se generan a partir de estos, de forma que cada vez que los objetos phaseImage y phaseLine cambien, se generarán objetos data1d y data2d actualizados. Al contrario que los objetos tipo PhaseImage y PhaseLine, los objetos tipo Data1D y Data2D contienen absolutamente toda la información (posición, escala, títulos de ejes,…) para su representación o su guardado, y lo que es más importante, no dependen de la implementación usada para PhaseLine o PhaseImage, detalle cuya importancia se comprenderá en el capítulo 5.1.1.

Por último existe una relación con una imagen tipo QImage que representa una vista previa de la imagen de la fase antes de ser procesada con los algoritmos.

3.2.4. Controlador de algoritmos

Como último elemento de este segundo nivel, aparece el controlador de algoritmos cuyo diagrama, tal y como se puede observar en la Figura 13, es el más sencillo de todos.

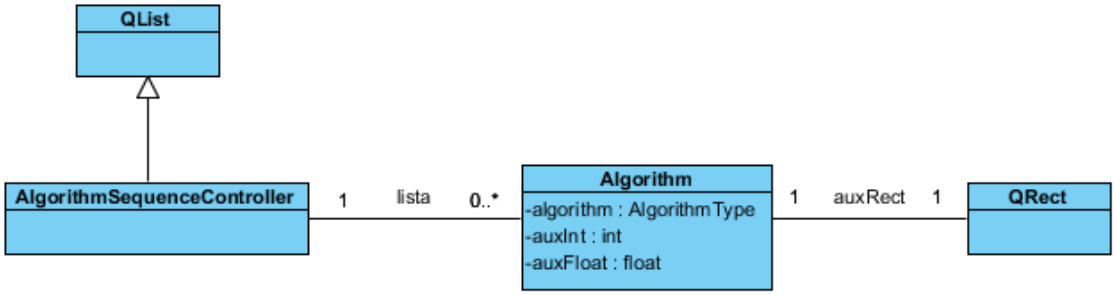

Figura 13. Vista de segundo nivel: Controlador de algoritmos

El controlador de algoritmos también es un tipo de lista, con capacidad de realizar operaciones adicionales sobre los algoritmos.

Cada algoritmo relacionado con el controlador tiene un atributo “algorithm”, del tipo de la enumeración AlgorithmType, que representa el tipo de algoritmo contenido; un entero auxiliar auxInt; un decimal auxiliar auxFloat y una relación con un rectángulo auxiliar a través de auxRect. Estos elementos auxiliares serán parámetros necesarios para algunos de los algoritmos.

3.3. Vista a tercer nivel del núcleo de la aplicación

Sumergiéndonos más en el modelo se llega a un tercer y último nivel.

Este nivel está relacionado con estructuras que han llegado a ser así por cuestiones de diseño. Son todos ejemplos de herencia y uso de factorías.

Nótese que los diagramas que se presentan hay que interpretarlos como piezas insertables en los diagramas de segundo nivel.

3.3.1. Cálculo de imágenes: GPU vs CPU

Uno de los requisitos no funcionales del proyecto, era el trabajo a la mayor velocidad posible.

Lo que más limita la velocidad es el tratamiento de imágenes y, dado que ese tratamiento es muy propenso a ser realizado en paralelo, la aplicación da la posibilidad de realizar los cálculos con la GPU (Graphics Processing Unit) utilizando la herramienta CUDA proporcionada por NVIDIA y disponible en algunas de sus tarjetas.

No obstante no se quería restringir el uso del programa a ordenadores con una tarjeta de ese tipo, por lo que todos los métodos se implementaron doblemente.

Para gestionar de forma adecuada esta situación, se aprovechó la potencia de la herencia y los patrones de factorías, llegando al siguiente modelo:

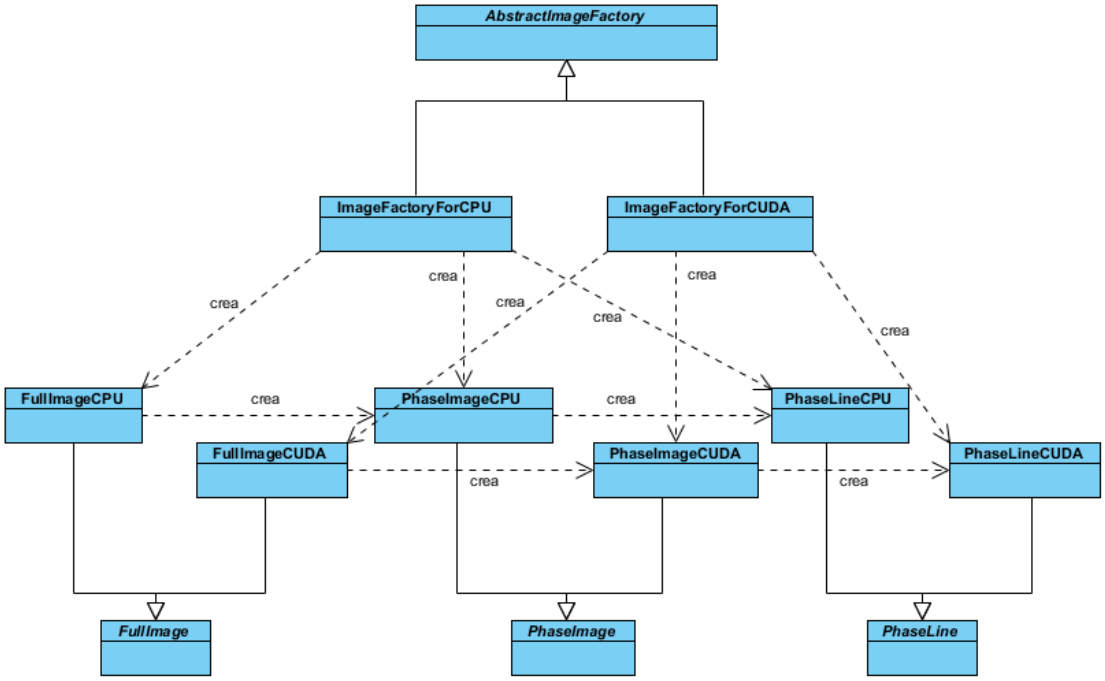

Figura 14. Vista de tercer nivel: Cálculo de imágenes

Como se puede observar, los objetos abstractos están implementados siempre de dos formas: o con CUDA, o con la CPU, y son las factorías concretas u objetos con el mismo tipo de implementación los que se encargan de su creación o de su copiado.

El uso de los patrones, así como la secuencia según la que se crean esos objetos serán explicados en la vista de interacción, en el capítulo 5.1.

3.3.2. Cámara

Para la implementación de la cámara, y pensando en el posible uso de otra cámara con otra API diferente en el futuro, se utilizó también una abstracción mediante herencia, y dado que lo que se pretende es que se use siempre la misma cámara, incluso si de enchufa o desenchufa varias veces, también se utilizó herencia, con un patrón de factorías.

De esta forma la distribución final es la siguiente:

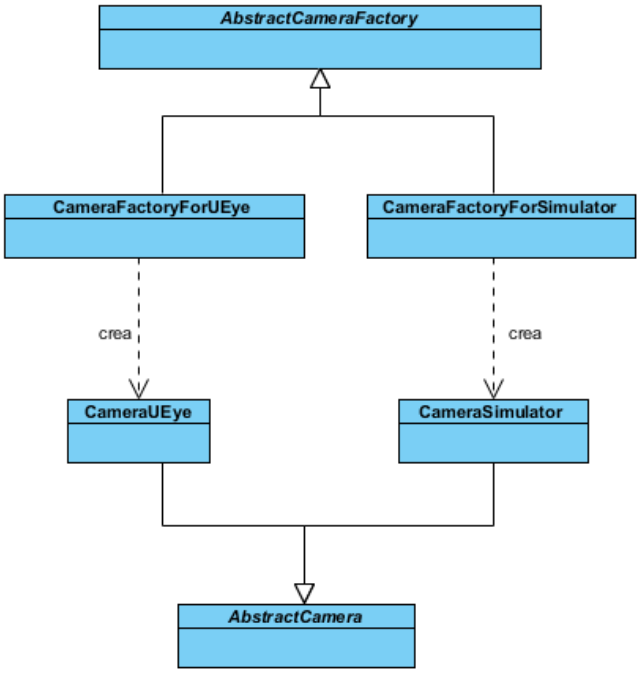

Figura 15. Vista de tercer nivel: Cámara

Esta forma de trabajar supuso una enorme ventaja, cuando adelantado el proyecto, surgió la necesidad de añadir un simulador de cámara (que leyera imágenes de un directorio), para poder mostrar la forma de funcionamiento del programa incluso cuando no hay una cámara compatible conectada.

3.4. Diagramas de la interfaz gráfica

A continuación se describirán las clases más importantes relacionadas con la interfaz gráfica. Se omiten los diálogos que sólo contienen controles de Qt por simplicidad.

3.4.1. Clases para la representación gráfica

Existirán 4 tipos distintos de representaciones gráficas en el diagrama. Estos cuatros tipos son los nodos hoja del siguiente diagrama:

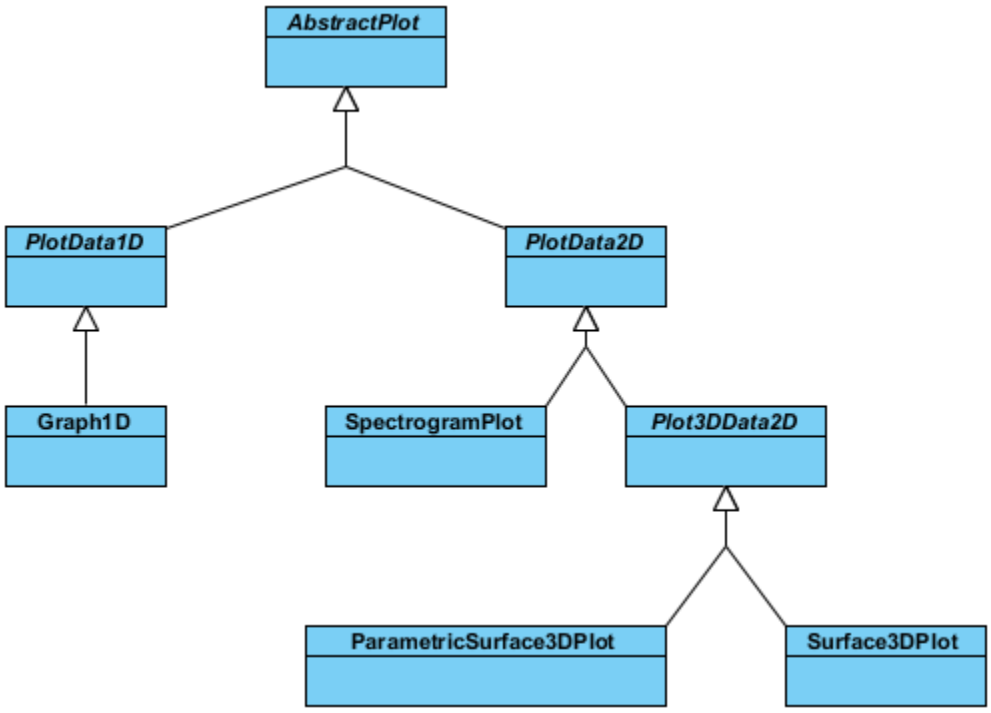

Figura 16. Diagrama de clases para los gráficos

Explicado, de padres a hijos:

- AbstractPlot: Clase abstracta orientada a representar cualquier estructura de datos que consista en una variable dependiente de cierto número de variables independientes. Contendrá varios métodos relacionados con la variable dependiente, así como un método abstracto para obtener el gráfico como un objeto QPixmap.

- PlotData1D: Clase abstracta orientada a representar estructuras con una sola variable independiente, lo que equivale en la aplicación a objetos tipo Data1D. Contendrá, entre otros, un método abstracto para establecer los datos mostrados: SetData1D(Data1D *).
- Graph1D: Heredando de PlotData1D implementará un gráfico eje X/eje Y, obteniendo un resultado como el siguiente:

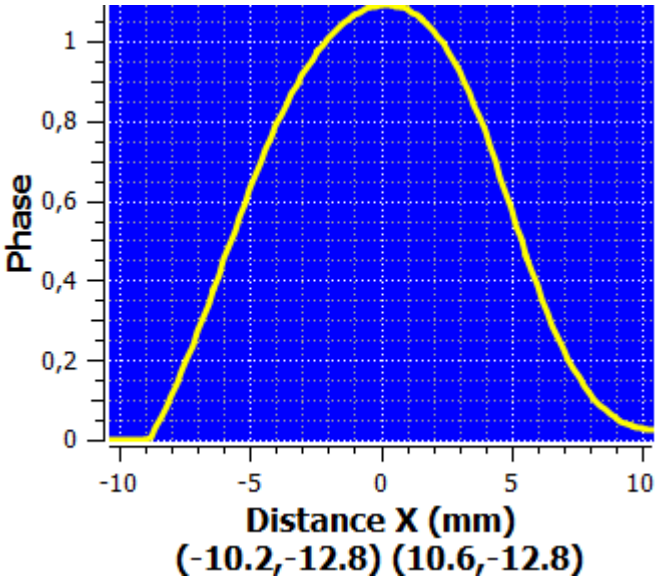

Figura 17. Graph1D

- PlotData2D: Clase abstracta orientada a representar estructuras con dos variables independientes, lo que equivale en la aplicación a objetos tipo Data2D. Contendrá, entre otros, un método abstracto para establecer los datos mostrados: SetData2D(Data2D *).
- SpectrogramPlot: Heredando de PlotData2D, implementará un espectrograma, donde los colores representarán la variable dependiente y los ejes X-Y las variables independientes, obteniendo un resultado como el siguiente:

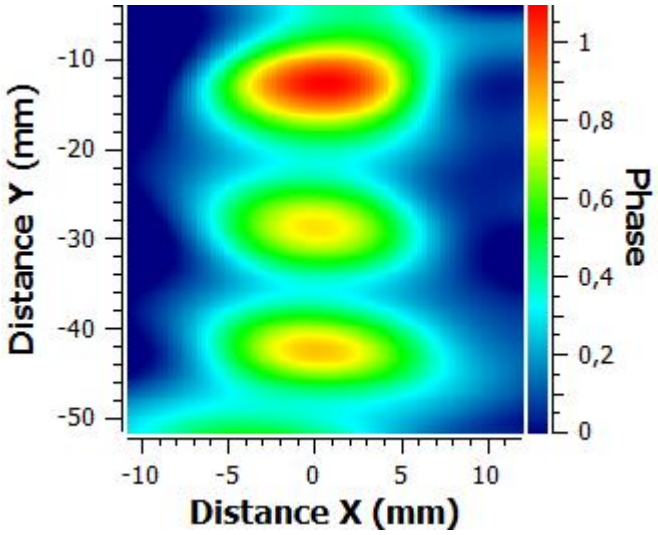

Figura 18. SpectrogramPlot

- Plot3DData2D: Heredando de PlotData2D, es una clase abstracta que se encarga de representar un gráfico 3D con un objeto tipo Data2D.
- Surface3DPlot: Representa un objeto tipo Data2D como una superficie en un espacio de 3 dimensiones, siendo el eje Z la variable dependiente y los ejes X e Y las variables independientes, obteniendo un resultado como el siguiente:

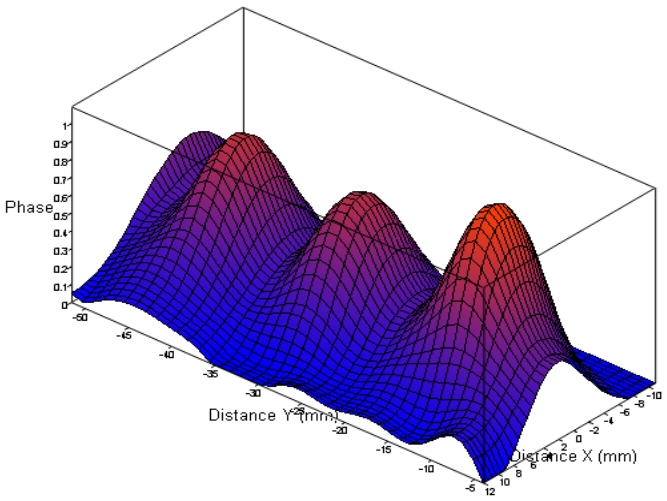

Figura 19. Surface3DPlot

- ParametricSurface3DPlot: Representa un objeto tipo Data2D como una superficie en un espacio tridimensional. La diferencia: ahora, tanto el eje X como el Y representan la primera variable independiente, y el eje Z representa la segunda variable independiente. Esta representación supone que el objeto Data2D pasado representa un objeto con simetría cilíndrica respecto del centro de la imagen. Admitido ese supuesto, la superficie representa los puntos en los que la variable dependiente alcanza cierto valor límite o *threshold*. Este *threshold* es elegido por el usuario del gráfico mediante el método SetThreshold(float). Al final se obtiene una imagen tridimensional del jet:

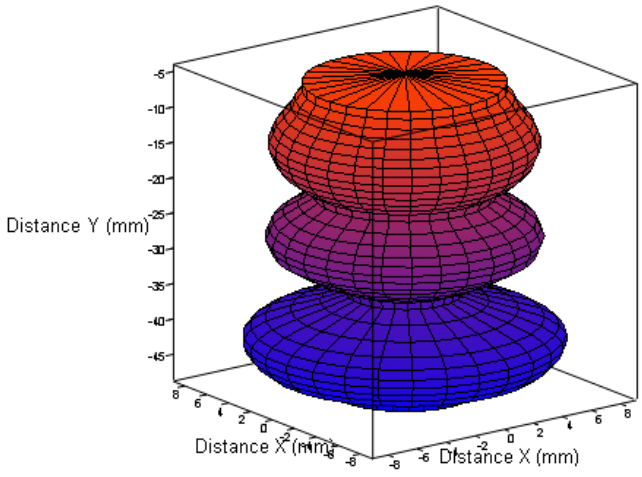

Figura 20. ParametricSurface3DPlot

3.4.2. Ventana principal

La ventana principal es el centro de trabajo de todo el programa. Desde ahí se accede al resto de funcionalidad del programa, se modifica la lista de algoritmos, se modifica la lista de imágenes,... Una vista del resultado tras la implementación de esta interfaz es la siguiente:

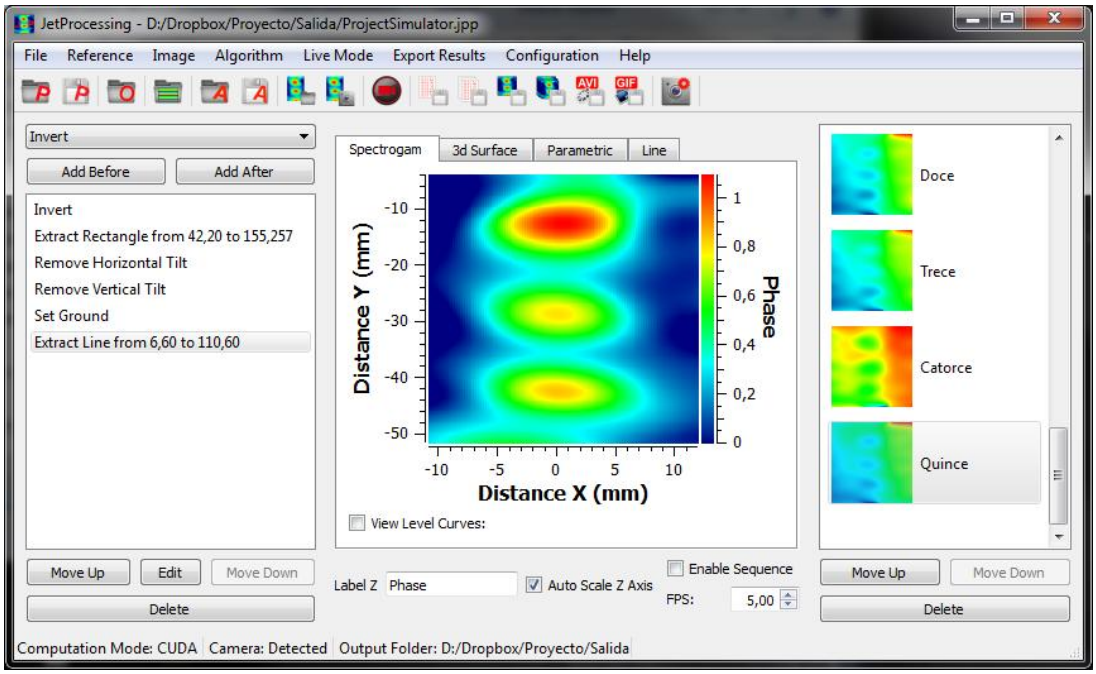

Figura 21. Imagen de la ventana principal

Y a continuación se representa un diagrama con sus objetos más relevantes, sin incluir los controles:

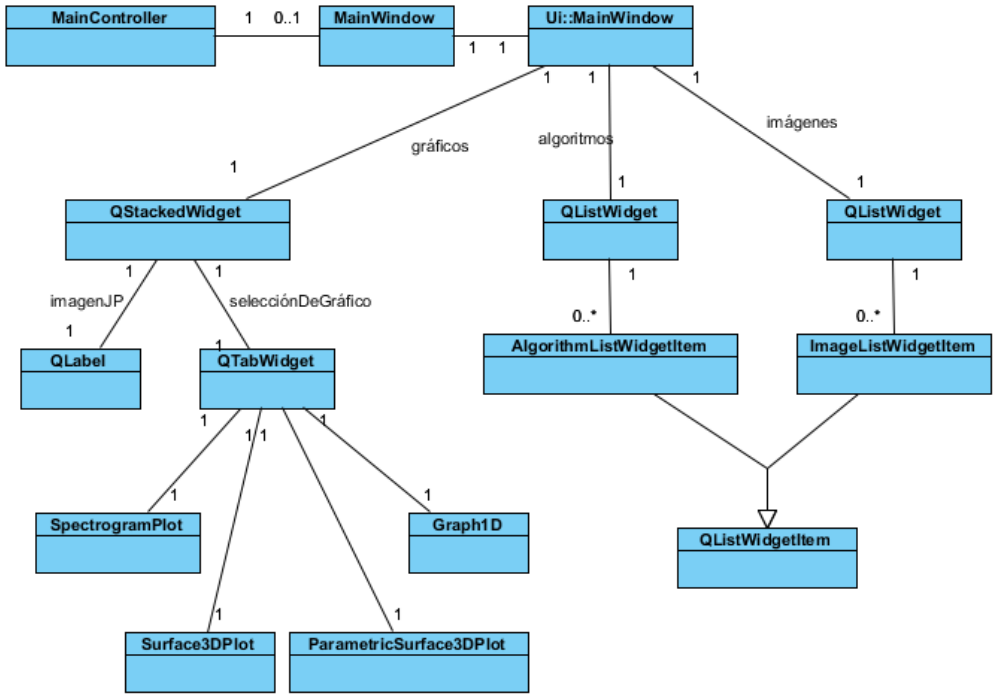

Figura 22. Diagrama de clases de la ventana principal

Como se puede observar la ventana principal consta de una lista de imágenes, una lista de algoritmos y unos gráficos, organizados primero en una pila, para poder ocultarlos o mostrarlos, y después mediante pestañas, para seleccionar el gráfico deseado.

3.4.3. Diálogo de selección de referencia

De la misma forma se puede ver una imagen del resultado del diálogo de fijación de la referencia, en este caso del paso 3, la selección de la máscara:

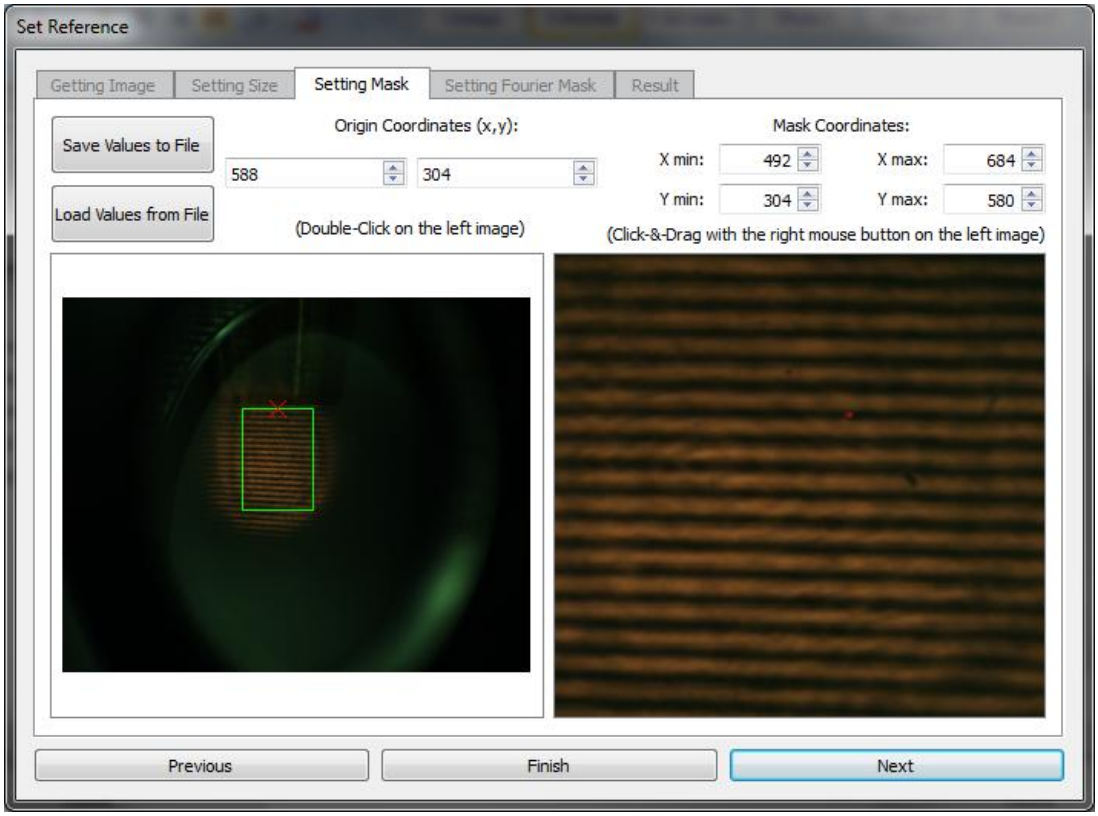

Figura 23. Imagen del diálogo de fijado de referencia, selección de la máscara

Y su diagrama asociado, en el que se han omitido los controles, que contiene una series de pestañas para los distintos pasos del proceso, y después distintas vistas previas para la muestra de los resultados parciales del proceso.

La clase MaskSelectionScene será la que permitirá la aplicación de operaciones con el ratón para arrastrar la imagen con el botón izquierdo, fijar la máscara con el derecho, fijar el origen con doble-click, o usar zoom con la rueda.

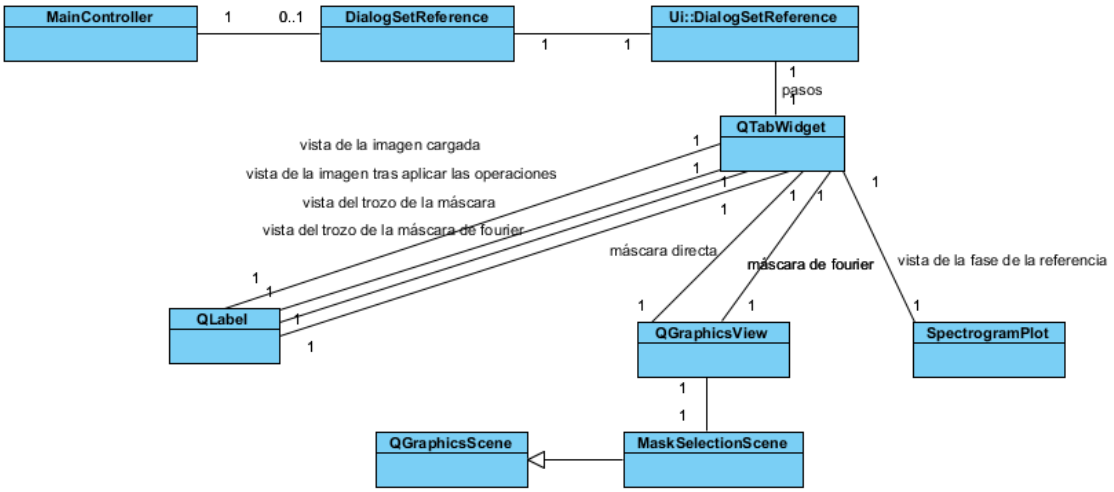

Figura 24. Diagrama de clases de la ventana de fijar referencia

3.4.4. Diálogo de selección de línea

Este diálogo permite elegir los parámetros de la operación de selección de rectángulo o de línea en una imagen de la fase. El resultado es el siguiente:

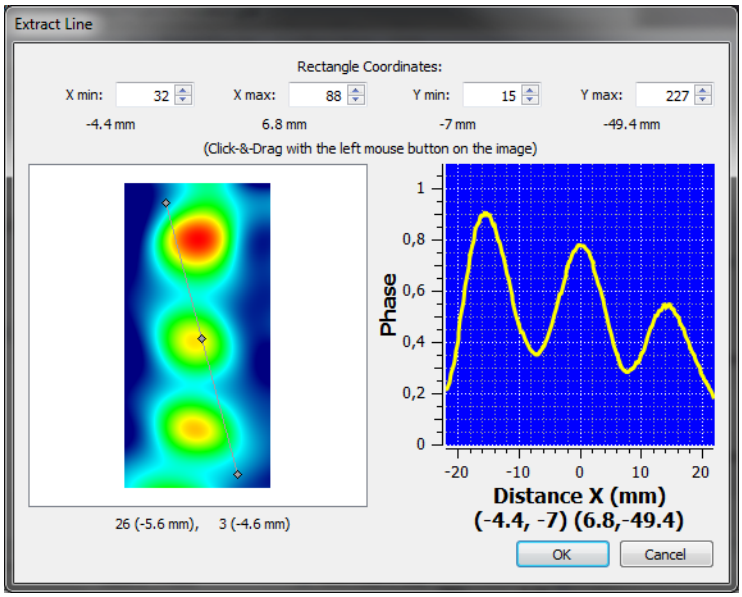

Figura 25. Imagen del diálogo de selección de línea

Y su diagrama asociado, que tras los apartados anteriores, no precisa de explicación:

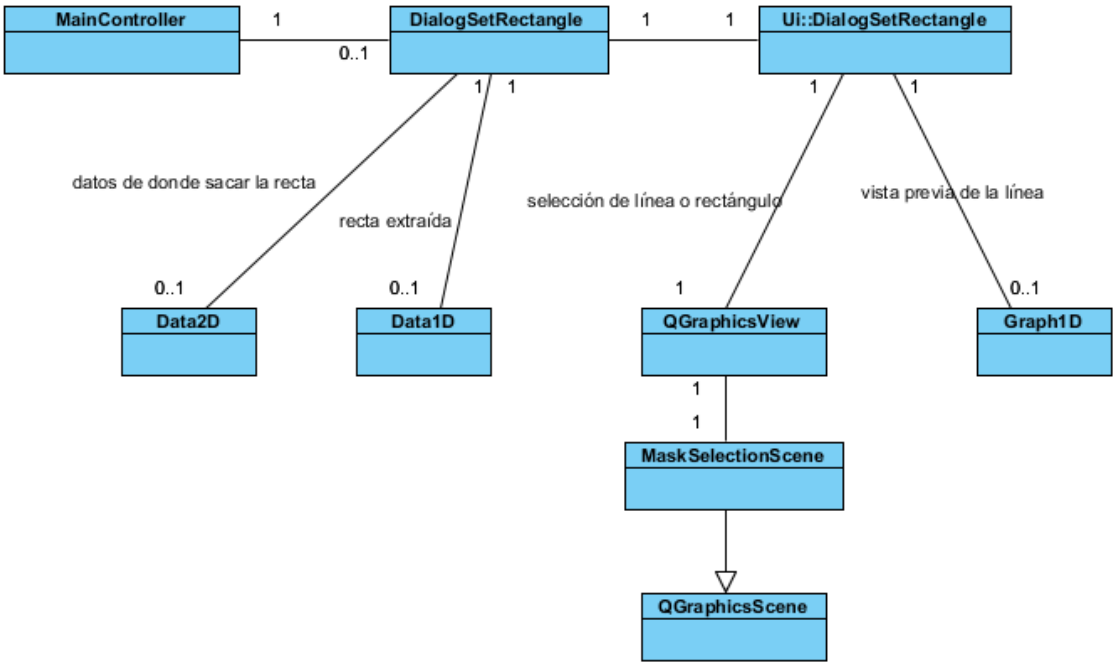

Figura 26. Diagrama de clases del diálogo de selección de línea

3.4.5. Diálogo de configuración de cámara

Este diálogo permite configurar los parámetros de la cámara mostrando información acerca de los valores y una vista previa. El resultado es:

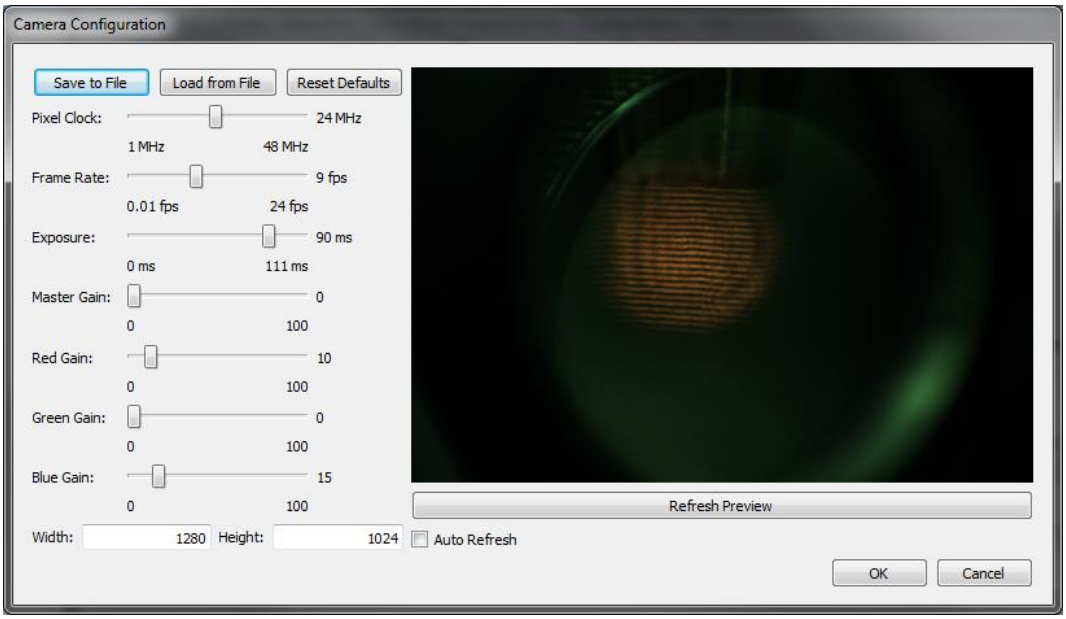

Figura 27. Imagen del diálogo de configuración de cámara

Y su diagrama es el siguiente:

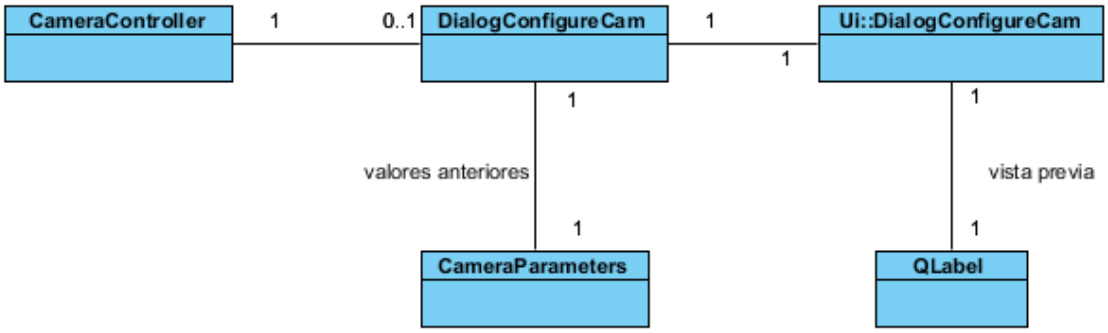

Figura 28. Diagrama del diálogo de configuración de la cámara

De este diagrama cabe destacar que trata directamente con el controlador de la cámara. Esto está diseñado así para su posible reutilización en otra aplicación totalmente distinta. Por otra parte se puede observar que incluye un objeto tipo CameraParameters en el que se guardan los parámetros de la cámara antes de llamar al diálogo, por si el usuario quiere cancelar.

3.4.6. Ventana de modo en vivo

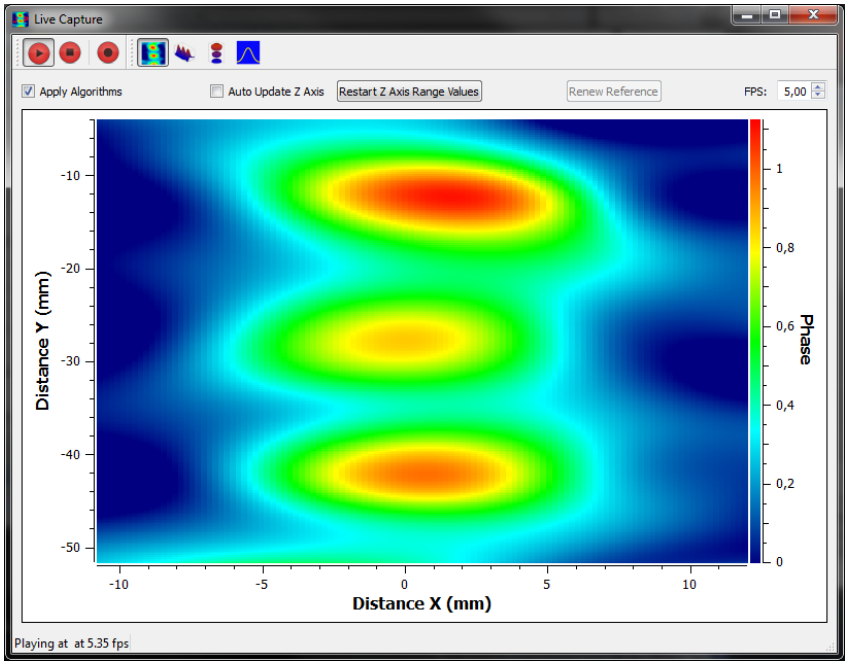

Figura 29. Imagen del la ventana del modo "en vivo"

La última interfaz que merece la pena detallar es la interfaz del modo “en vivo”. El resultado final se puede observar en la Figura 29.

Y el diagrama asociado, que utiliza una pila de widgets para mostrar los distintos gráficos es el siguiente:

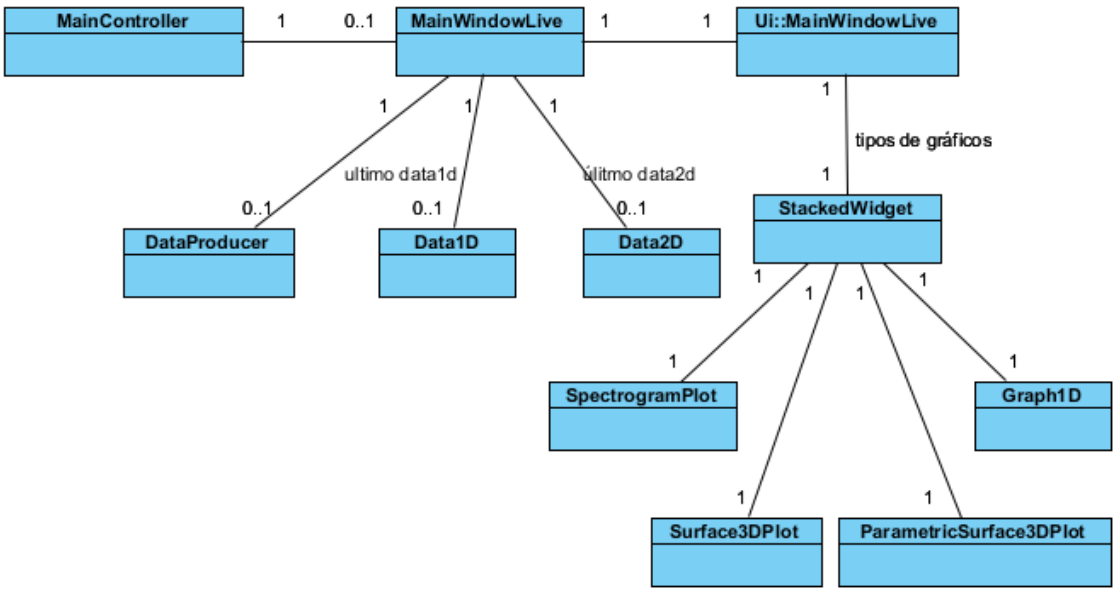

Figura 30. Diagrama de clases de la ventana del modo "en vivo"

Cabe destacar la relación con la clase DataProducer.

Esa clase permite el cálculo y procesado de datos en otro hilo, de forma que la interfaz no tiene que estar esperando ocupada por el cálculo y por lo tanto el usuario puede mover los gráficos anteriores o cambiar parámetros en la interfaz durante el cálculo. Este funcionamiento se explicará con detalle en el capítulo 5.5.1.

4. Diseño de datos

En esta sección se mostrarán los elementos de la aplicación relacionados con los datos.

4.1. Clases de datos

La mayor parte de las clases de datos ya se han explicado con detalle suficiente en las secciones anteriores, por lo que este apartado se centrará en los únicos objetos de datos puros del programa, los objetos tipo DataResult, que presentan un diagrama como el siguiente:

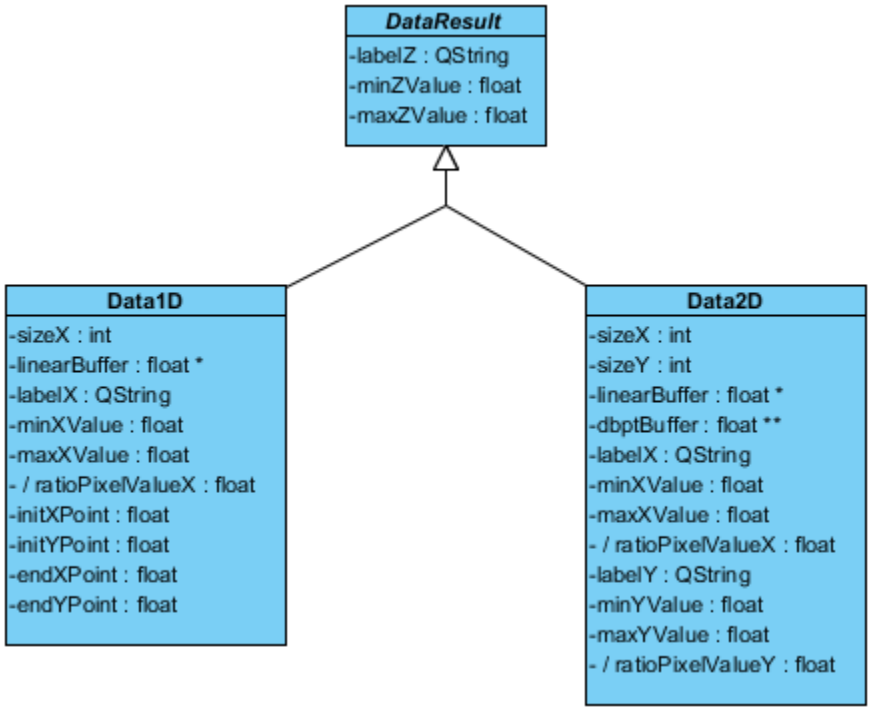

Figura 31. Diagrama de clases de datos

- DataResult: es una clase abstracta que contiene datos para representar con una variable dependiente. Tiene atributos con la etiqueta de la variable dependiente, así como los valores máximos y mínimos que toma.
- Data1D: es una clase que deriva de DataResult para una representación de datos que dependan de una sola variable independiente, una recta, que además ha sido extraída de otro objeto con dos variables independientes.

- sizeX y linearBuffer: Tamaño del buffer y buffer con los valores de la variable dependiente a lo largo de la recta.
- labelX: etiqueta de la variable independiente.
- minXValue y maxXValue: valores máximos y mínimos que toma la variable independiente.
- ratioPixelValueX: valor en unidades de distancia entre dos muestras contiguas para la variable independiente.
- initXPoint, initYPoint, endXPoint, endYPoint: coordenadas de los puntos inicial y final en el objeto original cuando se extrajo la recta.

- Data2D: es una clase que deriva de DataResult para una representación de datos que dependan de dos variables independientes: “X” e “Y”.
  - sizeX, sizeY: Número de muestras para cada una de las variables independientes.
  - linearBuffer: Buffer de tamaño sizeX·sizeY que contiene los datos de la variable dependiente alineados por filas.
  - dbptBuffer: Array de punteros para el acceso al buffer mediante doble indirección.
  - labelX: etiqueta de la variable independiente x.
  - minXValue y maxXvalue: valores máximos y mínimos que toma la variable independiente x.
  - ratioPixelValueX: valor en unidades de distancia entre dos muestras contiguas para la variable independiente x.
  - labelY: etiqueta de la variable independiente y.
  - minYValue y maxYValue: valores máximos y mínimos que toma la variable independiente y.
  - ratioPixelValueY: valor en unidades de distancia entre dos muestras contiguas para la variable independiente y.

4.2. Estructuras de archivo

Uno de los requisitos no funcionales (NFR-0004) de la aplicación hablaba de la comodidad de uso. Este requisito, junto con el requisito funcional añadido más tarde (FRQ-0007) acerca del guardado y carga de proyectos, es la motivación de este apartado: el guardado de información persistente en archivos.

En adelante se describen las estructuras de archivo usadas en la aplicación:

4.2.1. Archivo de guardado de la lista de algoritmos

El guardado de los algoritmos, implica guardar la información de una lista ordenada de objetos del tipo Algorithm.

Para realizar esa tarea se planteó la posibilidad de linealizar la información de dichos objetos, sin embargo, esa técnica tiene una gran desventaja: sin en un futuro se añadieran atributos nuevos a la clase Algorithm, los archivos anteriores dejarían de servir.

Esta idea, llevó directamente al uso de XML (Extensible Markup Language), un metalenguaje que permite crea una estructura de árbol en cuyos nodos se va encontrando la información. Este sistema tiene dos grandes ventajas:

- Es extensible, por lo que no presenta el problema con nuevas versiones comentado anteriormente.
- Crea un archivo de texto con estructura comprensible por el ser humano, por lo que es comprensible a simple vista, y puede ser modificado o creado sin necesidad del programa.

De esta forma, se crea un archivo con extensión ".alg" con la siguiente estructura:

-Utiliza un tipo de documento llamado AlgorithmList:

```
<!DOCTYPE AlgorithmList>
```

-Y crea un primer nivel llamado también AlgorithmList:

```
<AlgorithmList>
```

-Dentro de este primer nivel se encuentra un primer elemento, un número, la cuenta del número de algoritmos:

```
<Count>4</Count>
```

-Y a continuación una serie de nodos tipo Algorithm conteniendo cada uno de ellos el tipo y todos los parámetros del algoritmo asociado:

```
<Algorithm>
    <AlgorithmType>3</AlgorithmType>
    <AuxInt>0</AuxInt>
    <AuxFloat>0</AuxFloat>
    <AuxRectX1>42</AuxRectX1>
    <AuxRectX2>155</AuxRectX2>
    <AuxRectY1>20</AuxRectY1>
    <AuxRectY2>257</AuxRectY2>
</Algorithm>
```

Para terminar veamos un ejemplo de resultado con 4 algoritmos:

```
<!DOCTYPE AlgorithmList>
<AlgorithmList>
    <Count>4</Count>
    <Algorithm>
        <AlgorithmType>0</AlgorithmType>
        <AuxInt>0</AuxInt>
        <AuxFloat>0</AuxFloat>
        <AuxRectX1>0</AuxRectX1>
        <AuxRectX2>-1</AuxRectX2>
        <AuxRectY1>0</AuxRectY1>
        <AuxRectY2>-1</AuxRectY2>
    </Algorithm>
    <Algorithm>
        <AlgorithmType>8</AlgorithmType>
        <AuxInt>0</AuxInt>
        <AuxFloat>0</AuxFloat>
        <AuxRectX1>42</AuxRectX1>
        <AuxRectX2>155</AuxRectX2>
```

```
            <AuxRectY1>20</AuxRectY1>
            <AuxRectY2>257</AuxRectY2>
        </Algorithm>
        <Algorithm>
            <AlgorithmType>1</AlgorithmType>
            <AuxInt>0</AuxInt>
            <AuxFloat>0</AuxFloat>
            <AuxRectX1>42</AuxRectX1>
            <AuxRectX2>155</AuxRectX2>
            <AuxRectY1>20</AuxRectY1>
            <AuxRectY2>257</AuxRectY2>
        </Algorithm>
        <Algorithm>
            <AlgorithmType>9</AlgorithmType>
            <AuxInt>0</AuxInt>
            <AuxFloat>0</AuxFloat>
            <AuxRectX1>6</AuxRectX1>
            <AuxRectX2>110</AuxRectX2>
            <AuxRectY1>60</AuxRectY1>
            <AuxRectY2>60</AuxRectY2>
        </Algorithm>
    </AlgorithmList>
```

4.2.2. Archivo de guardado de la lista de imágenes

Por los mismos motivos que los discutidos en el apartado anterior, también se escoge XML como el lenguaje para el guardado de la lista de imágenes, con una salvedad: en este caso es necesario guardar una lista de objetos del tipo PhaseImage que además contienen un buffer con la información de la fase en cada punto.

Para compaginar esas dos situaciones, se crean dos archivos, uno con toda la información de los atributos de los objetos PhaseImage en formato XML, y con extensión ".iml", y otro en formato binario, con extensión ".dat", con los búferes de todas las imágenes, de forma que en el fichero XML exista una referencia al nombre del archivo binario.

De esta forma la estructura del archivo XML queda:

-Utiliza un tipo de documento llamado ImageList:

```
<!DOCTYPE ImageList>
```

-Y crea un primer nivel llamado también ImageList:

```
<ImageList>
```

-Dentro de este primer nivel se encuentra la cuenta del número de imágenes:

```
<Count>3</Count>
```

-Ahora se incluye la referencia al archivo de datos:

```
<DataFile>ImageData.dat</DataFile>
```

-Y a continuación una serie de nodos tipo Image conteniendo cada uno de ellos todos los parámetros de la imagen asociada, incluyendo los parámetros SizeX y SizeY, de forma que cada vez que se lea una nueva imagen, se sabe que se tiene que leer un buffer de dimensiones tamaño SizeX*SizeY del archivo de datos binarios:

```
<Image>
    <Label>Uno</Label>
    <PixelsFromOriginX>-96</PixelsFromOriginX>
    <PixelsFromOriginY>0</PixelsFromOriginY>
    <PixelsMmX>5</PixelsMmX>
    <PixelsMmY>5</PixelsMmY>
    <SizeX>193</SizeX>
    <SizeY>277</SizeY>
</Image>
```

Y un archivo completo queda de la siguiente forma:

```
<!DOCTYPE ImageList>
<ImageList>
    <Count>3</Count>
    <DataFile>ImageData.dat</DataFile>
    <Image>
        <Label>Uno</Label>
        <PixelsFromOriginX>-96</PixelsFromOriginX>
        <PixelsFromOriginY>0</PixelsFromOriginY>
```

```
            <PixelsMmX>5</PixelsMmX>
            <PixelsMmY>5</PixelsMmY>
            <SizeX>193</SizeX>
            <SizeY>277</SizeY>
        </Image>
        <Image>
            <Label>Dos</Label>
            <PixelsFromOriginX>-96</PixelsFromOriginX>
            <PixelsFromOriginY>0</PixelsFromOriginY>
            <PixelsMmX>5</PixelsMmX>
            <PixelsMmY>5</PixelsMmY>
            <SizeX>193</SizeX>
            <SizeY>277</SizeY>
        </Image>
        <Image>
            <Label>Tres</Label>
            <PixelsFromOriginX>-96</PixelsFromOriginX>
            <PixelsFromOriginY>0</PixelsFromOriginY>
            <PixelsMmX>5</PixelsMmX>
            <PixelsMmY>5</PixelsMmY>
            <SizeX>193</SizeX>
            <SizeY>277</SizeY>
        </Image>
    </ImageList>
```

4.2.3. Archivos de guardado de máscaras y operaciones

Para el guardado de máscaras y operaciones, dado que no se necesita guardar listas, sino un conjunto conocido de valores, se optó por la opción de crear archivos INI. Estos archivos no son más que archivos de texto con listas de parejas nombre_atributo = valor, que presentan también la extensibilidad tan deseada.

Existe el archivo con extensión ".msk1" que representa la máscara directa junto con el origen de coordenadas:

```
[General]
OriginX=588
OriginY=304
x1=492
```

```
x2=684
y1=304
y2=580
```

El archivo con extensión ".msk2" que representa la máscara de Fourier:

```
[General]
x1=1001
x2=1055
y1=428
y2=465
```

Y el archivo de operaciones, con extensión ".ops", que guarda otros parámetros y operaciones que se aplican a la referencia:

```
[General]
SourceHMirror=false
SourceVMirror=false
SourceRotation=0
FullImageSizeX=2048
FullImageSizeY=1024
PixelsMmX=@Variant(\0\0\0\x87@\xa0\0\0)
PixelsMmY=@Variant(\0\0\0\x87@\xa0\0\0)
```

Nótese que ese formato usado para PixelsMmX y PixelsMmY es la forma que utiliza Qt con su clase QSettings, usada para generar estos archivos, de guardar los números en coma flotante de precisión simple con el formato Variant.

4.2.4. Archivo de guardado de opciones de visualización

De la misma forma que para los parámetros de las máscaras y las operaciones, las opciones de visualización, que no son más que una serie de valores, se guardan en un archivo INI con extensión ".viw", quedando un resultado como el siguiente:

```
[General]
ZLabel=Phase
AutoScaleAxes=false
MinZValue=@Variant(\0\0\0\x87\0\0\0\0)
MaxZValue=@Variant(\0\0\0\x87?\xa6\x66\x66)
```

```
SequenceFPS=@Variant(\0\0\0\x87\x41\xf0\0\0)
ViewLevelCurves=false
Threshold=@Variant(\0\0\0\x87?333)
ApplyAlgorithmsLive=true
AutoScaleAxesLive=false
ThresholdLive=@Variant(\0\0\0\x87?333)
LiveFPS=@Variant(\0\0\0\x87@\xa0\0\0)
```

Los parámetros indican la etiqueta para el eje z, la opción de auto-escalado para la ventana principal, los valores máximos y mínimos si el auto-escalado no está activado, el número de imágenes por segundo que se muestran en la secuencia de la ventana principal, la opción de mostrar las curvas de nivel en el espectrograma, el valor límite para el gráfico paramétrico; así como otros parámetros para el modo “en-vivo” como es la opción de aplicar la lista de algoritmos, de auto-escalado de ejes, del valor límite para el gráfico paramétrico, y el número de imágenes que se deben procesar por segundo.

4.2.5. Archivo de guardado de parámetros de la cámara

Para los parámetros de la cámara se utiliza también un archivo INI con el siguiente resultado:

```
[General]
SizeX=1280
SizeY=1024
PixelClock=24
FrameRate=9.00268279947424
Exposure=12.7930833333333
MasterGain=0
RedGain=10
GreenGain=0
BlueGain=15
```

Este archivo, además de poder ser guardado por el usuario, también se almacena en una carpeta en la zona de aplicación para conservar los parámetros de una ejecución a la siguiente.

4.2.6. Guardado de proyecto: archivo “.jpp”

La clave acerca del guardado del proyecto está en torno a la cuestión: ¿qué es un proyecto?

En este caso se considera parte del proyecto a los elementos centrales que se han cargado o fijado a lo largo de la configuración del programa, salvo las preferencias del programa, que se consideran parámetros del programa en sí y no del proyecto.

Esto hace que guardar un proyecto sea sinónimo de guardar la imagen de la referencia, las operaciones realizadas sobre la referencia, las máscaras utilizadas para la referencia, la lista de algoritmos, la lista de imágenes, la configuración de la cámara y las opciones de visualización activadas.

Por lo tanto, vistos los elementos anteriores un proyecto consistirá en 9 archivos:

-ReferenceImage.bmp	Imagen de la referencia guardada sin pérdida.
-RefOperations.ops	Operaciones y parámetros para la referencia.
-DirectMask.msk1	Máscara directa y origen de coordenadas.
-FourierMask.msk2	Máscara de Fourier.
-AlgorithmList.alg	Lista de algoritmos.
-ImageList.iml	Lista de imágenes.
-ImageData.dat	Búferes de con los datos de la lista de imágenes.
-CameraSettings.ini	Parámetros de la cámara.
-ViewOptions.viw	Opciones de visualización seleccionadas.

Sin embargo, no es cómodo trabajar con 9 archivos, por lo que todos ellos se crearán en una carpeta temporal que después se comprimirá en un fichero comprimido ZIP marcado con la extensión “.jpp” (JetProcessing Project).

A continuación para usar ese fichero, el programa realizará la operación contraria: lo descomprimirá en una carpeta temporal, y leerá los distintos archivos.

Esta característica se puede ver reflejada en la siguiente imagen:

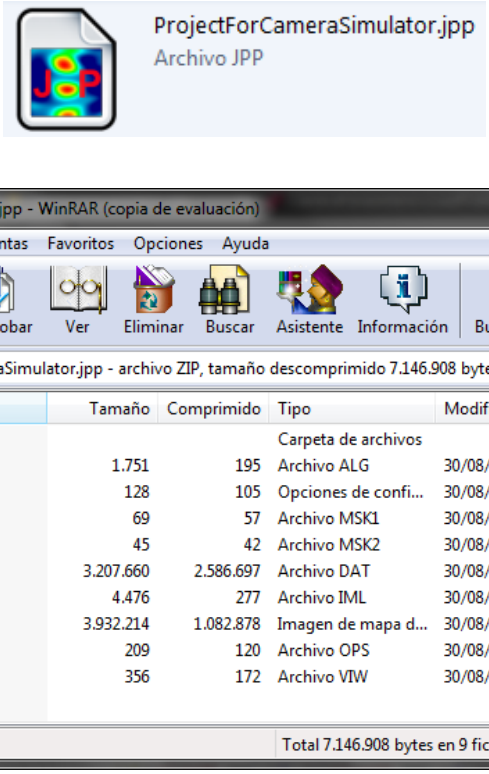

Figura 32. Archivo de proyecto

4.2.7. Archivos de guardado de preferencias del programa

Para las preferencias del programa se utiliza de la misma forma un archivo INI. Con el siguiente resultado:

```
[General]
Mode=1
UseDefaultMode=false
OutputFolder=D:/SalidaJetProcessing
SaveCameraPictures=false
ApplyUntilCurrentAlgorithm=true
SaveGraphicsViewSize=false
CustomSize=@Size(600 600)
DefaultImageFormat=png
FirstExecution=false
```

El primer parámetro indica que el modo por defecto es la CPU, la segunda línea indica que se debe preguntar al usuario antes de usar el modo por defecto, el tercero indica

la carpeta de salida, el cuarto indica que las imágenes tomadas por la cámara no se deben guardar de forma adicional en un directorio, el siguiente que sólo se debe aplicar hasta el algoritmo seleccionado en las vistas del gráfico, el siguiente indica que los gráficos se deben guardar con el tamaño personalizado, a continuación indica el tamaño personalizado de los gráficos, el formato por defecto y que no es la primera ejecución del programa.

Este archivo se salvará además en una carpeta creada por el programa en la zona de datos de aplicación del sistema operativo. Este hecho es importante, ya que en algunos sistemas operativos se requiere permisos de administrador para escribir en los directorios de instalación.

4.2.8. Archivo con los datos exportados

Tanto objetos del tipo Data1D como Data2D se pueden exportar como un archivo de texto para su uso en otras aplicaciones como MATLAB o Mathematica. El formato de estos archivos consiste en la información de cada punto, alineado en columnas con formato científico, separadas por tabulaciones.

Para los objetos tipo Data1D, la primera columna representa la variable independiente y la segunda la variable dependiente. De esta forma queda el archivo:

```
-2.600354e+00    6.966785e-01
-2.400327e+00    7.082317e-01
-2.200300e+00    7.188846e-01
-2.000272e+00    7.286259e-01
-1.800245e+00    7.374325e-01
-1.600218e+00    7.452854e-01
-1.400191e+00    7.521675e-01
-1.200163e+00    7.580559e-01
-1.000136e+00    7.629296e-01
.
.
.
```

Para el objeto tipo Data2D, la primera columna representa la variable independiente “X”, la segunda la variable independiente “Y” y la tercera la variable dependiente, quedando el archivo:

```
-9.791151e+00     -5.160000e+01     4.154086e-03
-9.589380e+00     -5.160000e+01     1.488143e-02
-9.387610e+00     -5.160000e+01     2.564687e-02
-9.185841e+00     -5.160000e+01     3.645432e-02
-8.984071e+00     -5.160000e+01     4.729223e-02
-8.782301e+00     -5.160000e+01     5.817211e-02
-8.580531e+00     -5.160000e+01     6.907496e-02
.
.
.
-9.791151e+00     -5.139915e+01     3.747642e-03
-9.589380e+00     -5.139915e+01     1.432240e-02
-9.387610e+00     -5.139915e+01     2.493906e-02
-9.185841e+00     -5.139915e+01     3.559011e-02
-8.984071e+00     -5.139915e+01     4.627544e-02
-8.782301e+00     -5.139915e+01     5.699128e-02
-8.580531e+00     -5.139915e+01     6.774148e-02
.
.
.
```

5. Vista de interacción

Con la vista de interacción se finalizará este modelo de diseño.

Dado que en el modelo de análisis ya se especificó una vista de interacción, en este documento se omitirán algunos diagramas, por carecer de interés añadido a los de análisis, para poder dedicar más tiempo al estudio de los diagramas que pongan de manifiesto el uso de estrategias o patrones que hayan sido claves en la fase de diseño.

5.1. Realización de cálculos

En los siguientes apartados se muestran las secuencias temporales que ocurren en la aplicación relacionadas con los cálculos que se realizan sobre las imágenes tomadas con la cámara o cargadas desde archivo para la obtención final del jet.

5.1.1. CUDA vs CPU: Factoría de imágenes

Como ya se indicó en los requisitos, es necesario que el programa tenga la capacidad de realizar los cálculos en paralelo con la tarjeta gráfica. Sin embargo, esa capacidad debe ser opcional para permitir que cuando un equipo no disponga de CUDA, o un usuario prefiera no usarlo, la aplicación sea capaz de trabajar con la CPU.

Para gestionar estas características, la estrategia elegida ha sido abstracción con herencia junto con el patrón Abstract Factory con Factory Method.

El objetivo del uso de este patrón es aislar en una sola parte del programa la decisión entre trabajar con CUDA o con la CPU, de forma que para el resto del código de la aplicación esta diferencia sea totalmente transparente.

El modo de funcionar de este patrón es el siguiente:

- Definimos una clase abstracta que represente a la clase que deba ser implementada de las dos formas, como en este caso es PhaseImage, con los métodos necesarios para los cálculos.

- Heredando de esta clase creamos las dos clases concretas que implementan la clase abstracta según cada una de las dos técnicas: CUDA o CPU, como en este caso son PhaseImageCPU cuyos métodos están implementados con la CPU y PhaseImageCUDA cuyos métodos están implementados con CUDA.

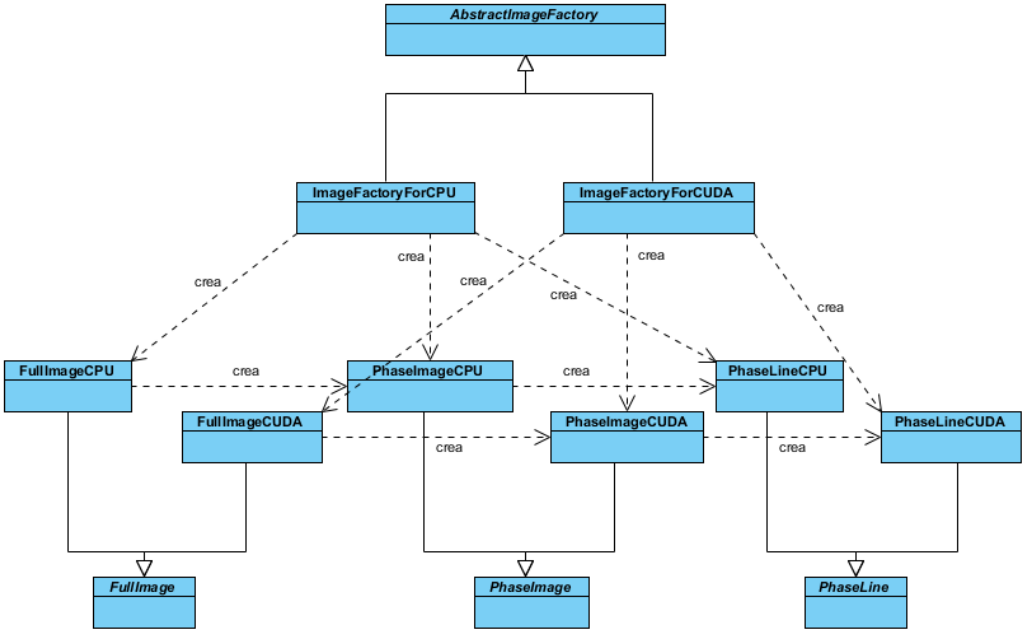

Figura 33. Abstract Factory con Factory Method para el cálculo con CUDA o CPU

Al trabajar de esa forma, conseguimos utilizar un puntero declarado como "PhaseImage *" en el resto del código, y con unos métodos que no dependen del hecho de trabajar con CUDA o con la CPU, aunque sí lo hará el objeto al que apunta el puntero, y por tanto la implementación de los métodos.

Sin embargo, ahora seguimos teniendo el problema de la creación de los objetos, dado que en ese momento es necesario saber si se debe usar el constructor de PhaseImageCPU o de PhaseImageCUDA. Ahí es donde entra en juego la factoría abstracta:

- Creamos una clase AbstractImageFactory que implemente los métodos necesarios para crear o copiar un objeto PhaseImage, devolviendo un puntero.

- Creamos dos factorías concretas, ImageFactoryForCPU e ImageFactoryForCUDA, que crean los objetos con el tipo de implementación indicado por su nombre.

De esta forma, quien necesite crear o copiar objetos de imagen, que en este caso es la clase MainController, siempre lo hará a través de una factoría a la que tendrá acceso a través de un puntero "AbstractImageFactory *" que apuntará a un tipo concreto de factoría. Y, como se comprueba en los tres capítulos siguientes, una vez iniciada la factoría, el código es totalmente transparente al tipo de implementación que se esté usando.

Al comenzar la ejecución del programa, el controlador principal MainController, detectará si CUDA está disponible.

- Si CUDA no está disponible, iniciará la factoría con una factoría de imágenes para el cálculo con la CPU.
- Si CUDA está disponible, pero hay que usar el modo por defecto sin preguntar, iniciará la factoría para el cálculo con la CPU o con CUDA, dependiendo del modo por defecto.
- Si CUDA está disponible, y no hay que usar el modo por defecto, se muestra una ventana para que el usuario elija, y se inicia la factoría con el modo elegido.

Este inicialización se puede observar en los siguientes diagramas:

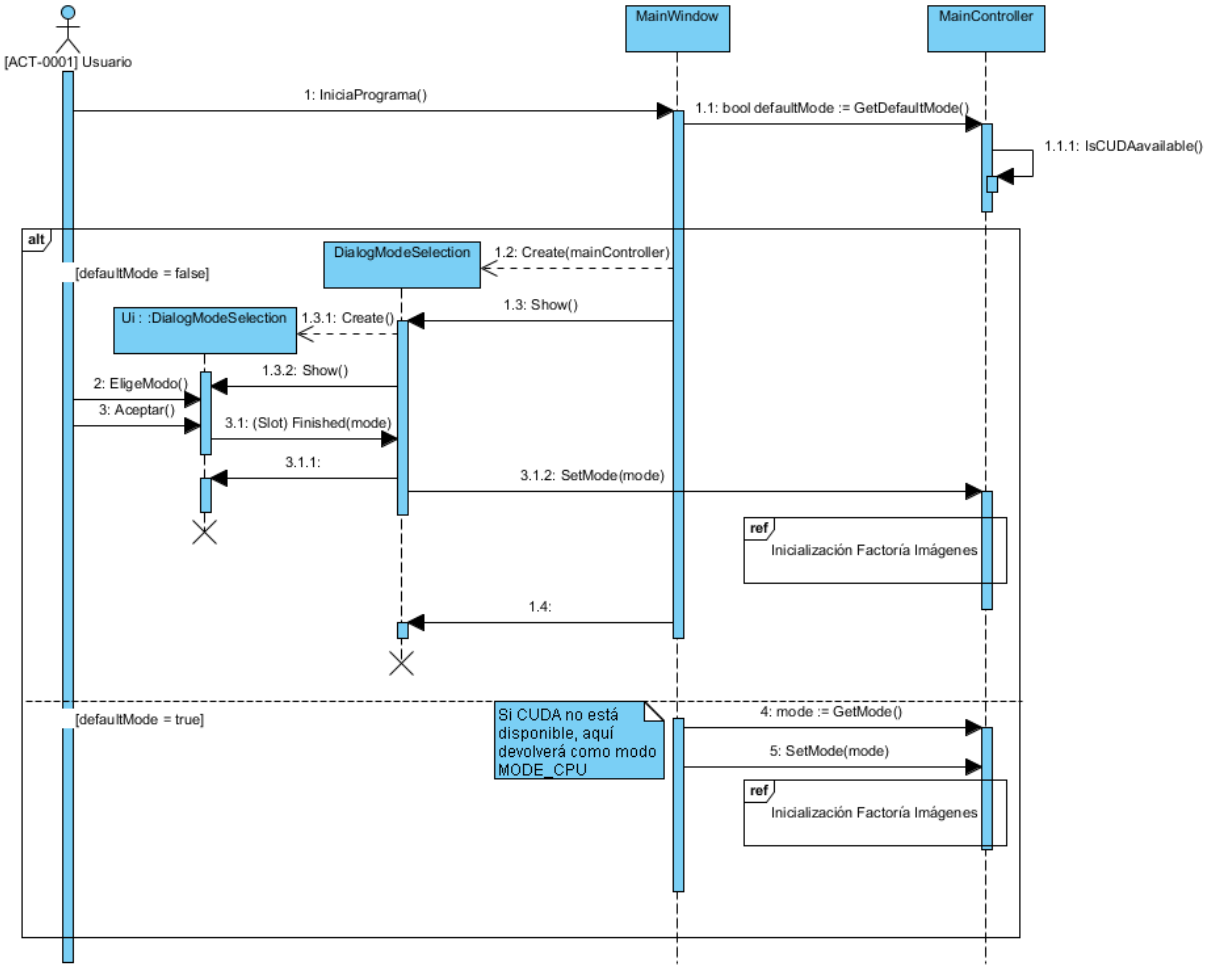

Figura 34. Diagrama de Secuencia: Selección de modo

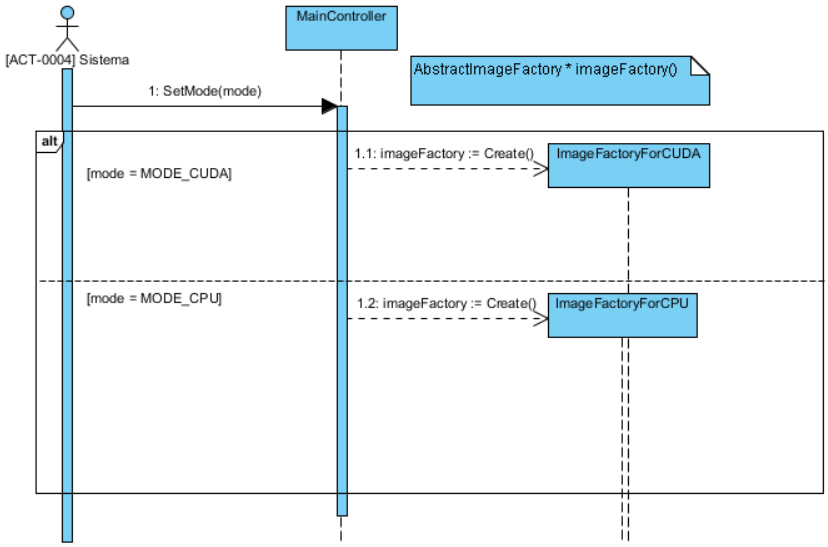

Figura 35. Diagrama de Secuencia: Iniciado de la factoría de imágenes

5.1.2. Procesado de la referencia

El primer paso a hora de obtener una imagen del jet de gas es procesar la referencia para obtener la fase.

Para ver esto, se va a utilizar el siguiente diagrama, en el que se va a suponer que la factoría seleccionada es la factoría de CUDA y que ya se tiene un objeto tipo QImage con la imagen de la referencia sin jet, así como todos los parámetros para el procesado. Por simplicidad del diagrama se ha obviado la comprobación de actualización de los productos en cada uno de los pasos (Es posible que si sólo se varió algún parámetro no sea necesario pasar por todas las fases), pero en la implementación final se comprueba y sólo se recalcula desde donde sea necesario.

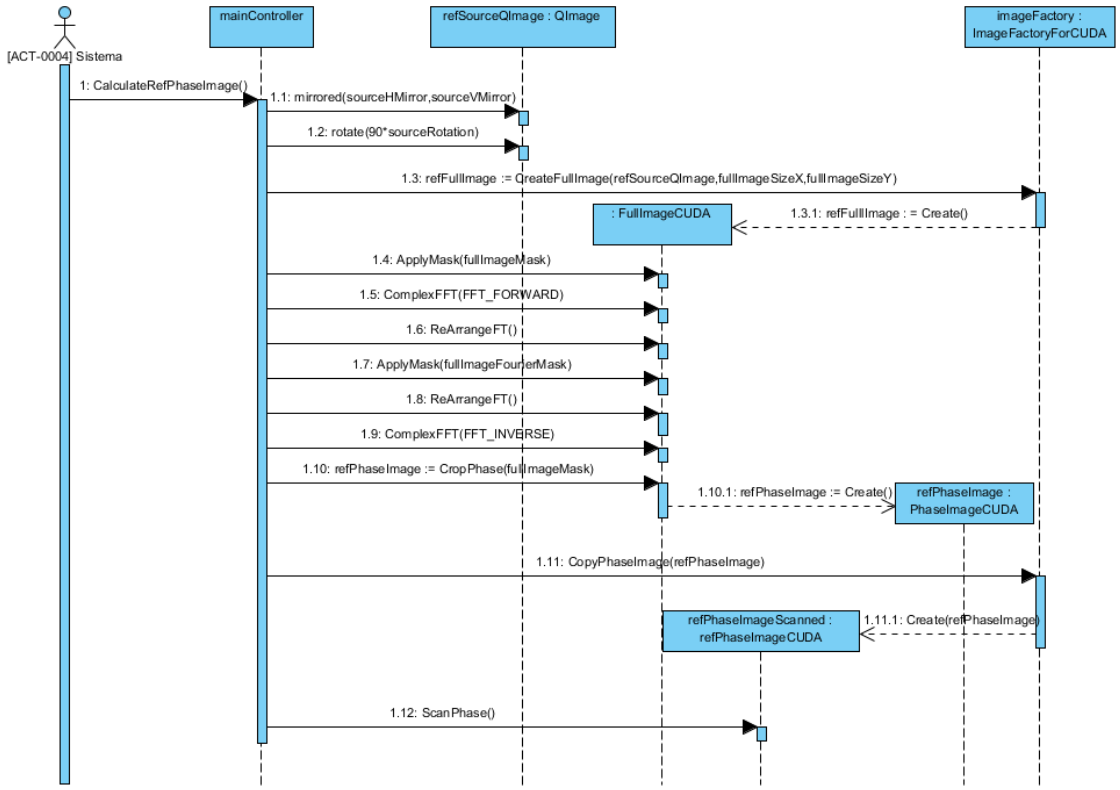

Figura 36. Diagrama de Secuencia: Procesado de la referencia

Se puede observar que para la creación y el copiado de los objetos relacionados con CUDA, o bien se usa la factoría, o bien se usa otro objeto relacionado con CUDA.

5.1.3. Pre-procesado de una imagen

El pre-procesado de una imagen es el procesado que se realiza a una imagen del patrón de interferencia con el jet de gas. Tras pre-procesar esa imagen con la referencia ya obtenida anteriormente, se obtiene la primera imagen del jet.

En este diagrama, seguimos suponiendo que se está utilizando CUDA, que tenemos una imagen de la referencia, y que tenemos un objeto tipo QImage, con la imagen que queremos pre-procesar. El diagrama es el siguiente:

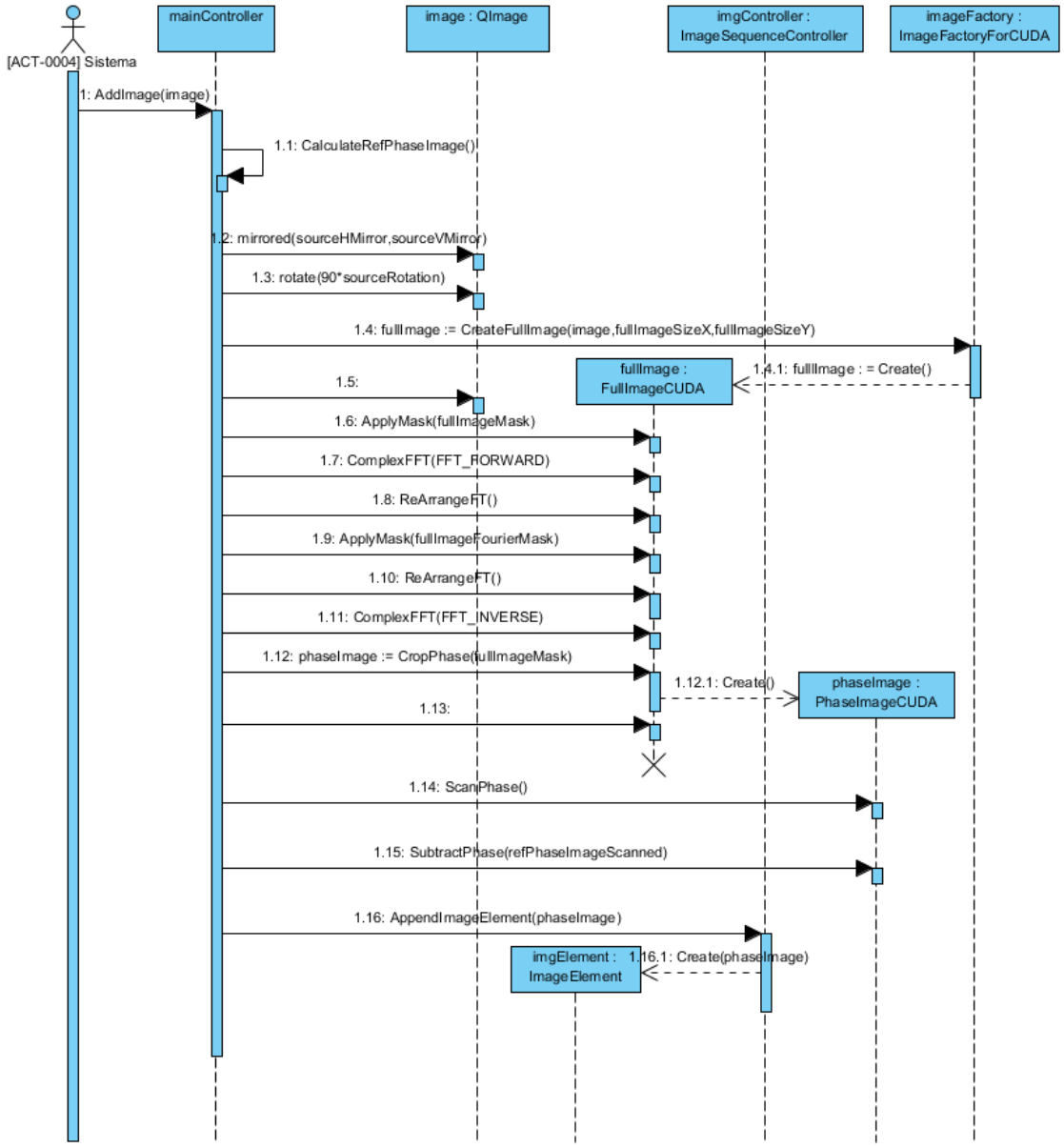

Figura 37. Diagrama de Secuencia: Pre-procesado de una imagen

Como se puede observar el proceso es esencialmente el mismo, salvo al final, cuando se resta la fase de la referencia a la fase obtenida para la imagen, obteniendo así la imagen, para añadir después la imagen a la lista.

5.1.4. Aplicación de algoritmos

Ya se ha obtenido la imagen del jet, sin embargo, es una imagen sucia y con distorsión, que se quiere limpiar, y quizá aplicar operaciones sobre ella.

En el siguiente diagrama se puede observar este proceso. Se supone que hay una imagen de la fase que procesar, así como una lista de algoritmos:

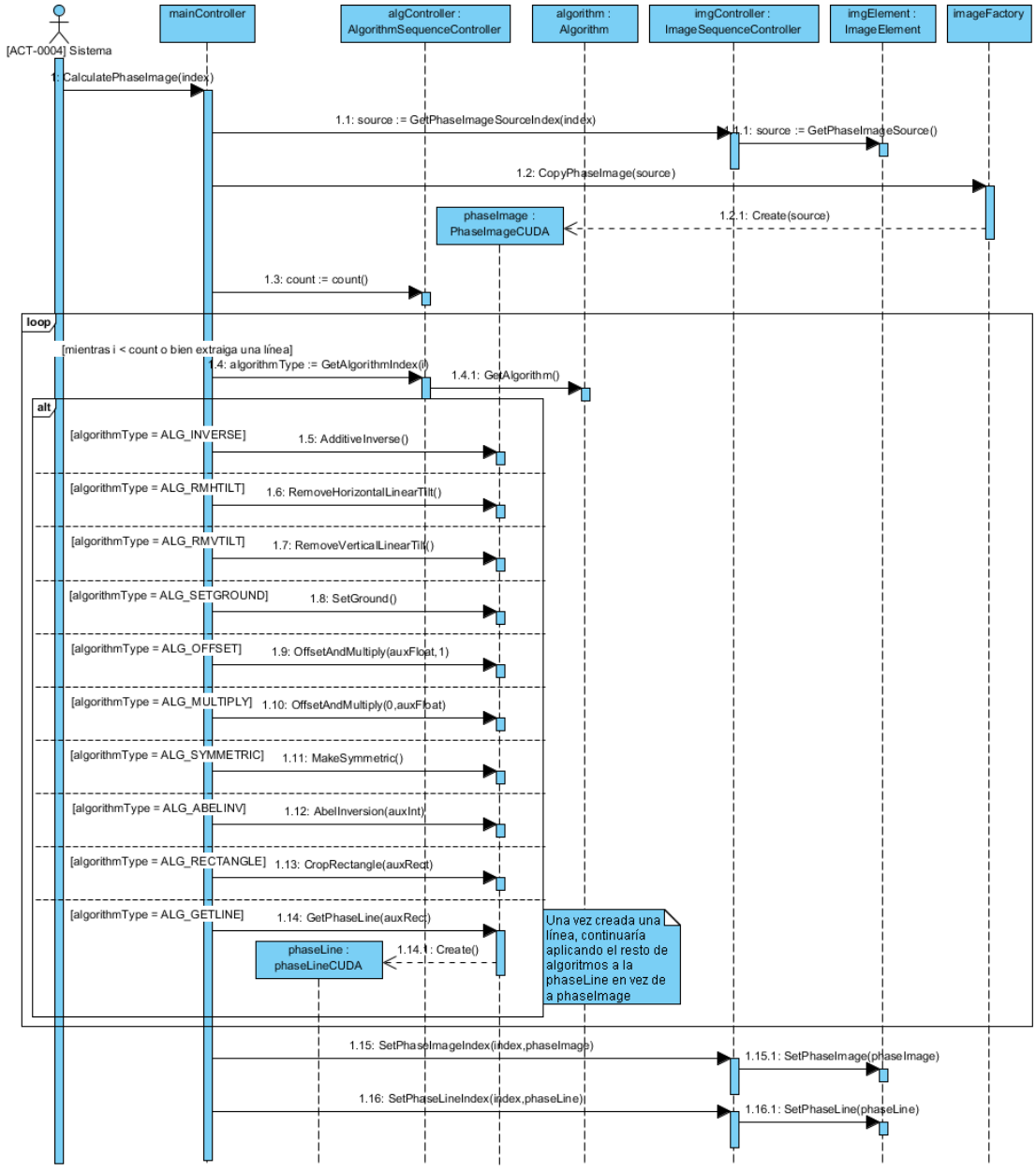

Figura 38. Diagrama de Secuencia: Aplicación de algoritmos

5.2. Controlador de la cámara

5.2.1. Inicialización del controlador: Factoría de cámaras

El controlador de la cámara está preparado para poder soportar dos tipos de cámaras: una cámara uEye o un simulador de cámara. Para gestionar esto, y por motivos análogos a los justificados en los apartados anteriores, se recurrirá también a la técnica de las factorías.

Con un diagrama de clases como el que se puede ver a continuación, se tienen todas las herramientas necesarias para utilizar una u otra cámara de forma transparente al resto del código:

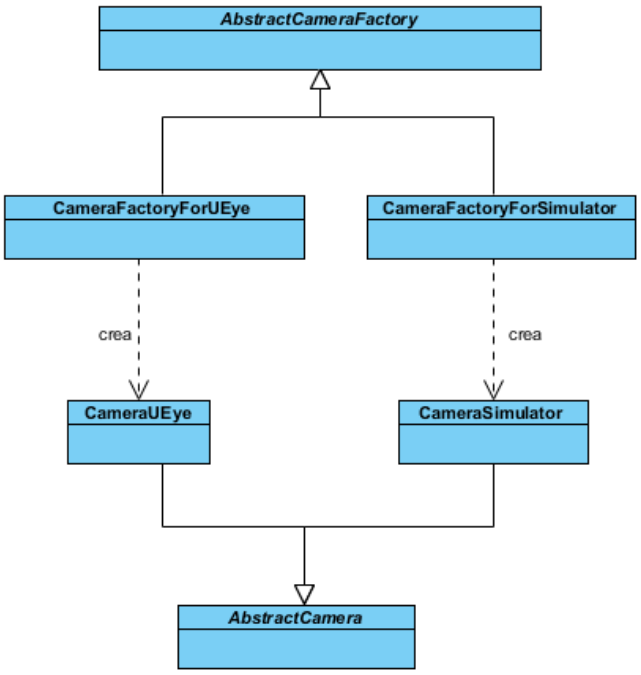

Figura 39. Abstract Factory con Factory Method para la cámara

En este caso, se utilizará el simulador si el usuario lo ha indicado por la línea de comandos, lanzando un programa con un argumento "–cs" o "–CameraSimulator". En otro caso, como se puede observar en el siguiente diagrama, al crear la clase del controlador de la cámara, se iniciará con la cámara de la API uEye:

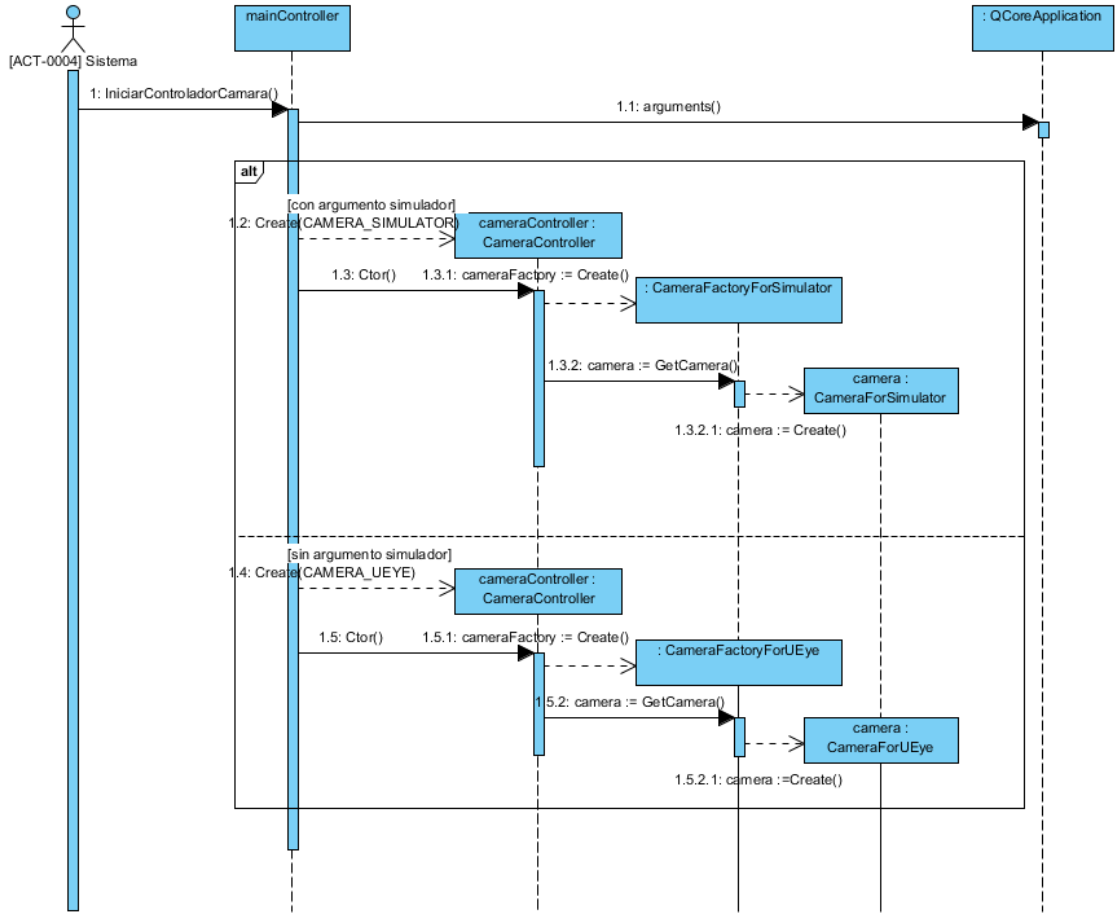

Figura 40. Diagrama de Secuencia: Iniciado de la factoría de cámaras

5.2.2. Toma de imágenes: Hilo independiente

Uno de los requisitos no funcionales del que ya se ha hablado, es la capacidad de la aplicación de mostrar el jet a tiempo real lo más rápido posible. Hasta ahora nos hemos preocupado de la parte del procesado la imagen, usando procesamiento en paralelo para su optimización, sin embargo hay otro punto crítico en el proceso: la toma de la imagen.

Típicamente para el tipo de experimento que se está realizando, la imagen se toma con una exposición de alrededor de 100 ms, y durante esos 100 ms, el sistema se queda esperando a que la célula CCD acabe de recoger fotones, es decir, bloqueado sin consumo de CPU desde que pide la imagen, hasta que la obtiene.

Como en este caso, no se puede permitir tan grandísima pérdida de tiempo, se ha recurrido a la utilización de un hilo en paralelo que se encargue de tomar las imágenes.

El objetivo es que el controlador de la cámara, tenga siempre disponible una imagen cuando se la pidan, mediante "GetSourcePicture" desde fuera, de forma que mientras se está procesando una imagen, un hilo que ejecute el controlador esté bloqueado tomando la siguiente.

El método del controlador que se ejecutará en el otro hilo será el método "run", que comenzará a funcionar a partir de que se llame al método "initialize" del controlador.

Sin embargo, el problema de esta técnica es que no garantiza que la imagen sea reciente cuando lo que se está tomando es una sola imagen, y no una secuencia (En el caso de una secuencia puede que la primera no lo fuera, pero el resto sí). La solución a esto es incluir un método que permita renovar la imagen "RenewPicture" de forma que cuando sea llamado, el controlador deseche la imagen anterior y tome una nueva.

La forma de coordinar estos dos hilos, se produce mediante el uso de semáforos, en este caso 3 semáforos y un mutex. Dos de los semáforos se utilizan para controlar el flujo productor consumidor, y el tercer semáforo para asegurar una finalización ordenada del hilo, al destruir el controlador. El mutex simplemente se utiliza para evitar que se cambien parámetros de la cámara mientras se está tomando una imagen.

El esquema de utilización de los semáforos es el siguiente (Sólo se muestra la parte del código relacionada con los semáforos):

Por una parte al iniciar el hilo (Con "start"):

```
bool CameraController::Initialize(){
    semaphoreProducer = new QSemaphore(0);
    semaphoreConsumer = new QSemaphore(0);
    semaphoreEnding = new QSemaphore(0);
    mutexPicture = new QMutex();
    //Libero el semáforo del productor
    semaphoreProducer->signal();
    this->start();
```

```
}
```

El hilo encargado de tomar la imagen:

```
for(;;){
      //Espero a que el cliente consuma
      semaphoreProducer->wait(INFINITE);
      //Si me han dicho que tengo que finalizar salgo del bucle
      if(semaphoreEnding->wait (0)==true) break;
      //Tomo la imagen
      mutexPicture->wait(INFINITE);
      buffer=camera->AcquireSourcePicture();
      mutexPicture->signal();
      //Aviso al cliente que ya hay disponible
      semaphoreConsumer->signal();
}
```

El método que renueva la imagen (Sólo tiene sentido renovarla si había una disponible):

```
if(semaphoreConsumer->wait(0)==true){
    delete buffer;
    buffer=NULL;
    semaphoreProducer->signal();
}
```

El método que permite obtener la imagen:

```
semaphoreConsumer->wait(INFINITE);
aux=buffer;
buffer=NULL;
semaphoreProducer->signal();
```

Un método cualquiera que varía un parámetro de la cámara:

```
mutexPicture->wait(INFINITE);
returnValue=camera->SetGain(command,value);
mutexPicture->signal();
```

Un método que interrumpa el hilo:

```
//Libero el semáforo de finalizar para que el hilo acabe
semaphoreEnding->signal();
//Libero el semáforo del productor, para asegurar que no se
//queda atascado en ese paso
semaphoreProducer->signal();
//Espero a que el hilo haya acabado (Al acabar envía
//una señal al controlador)
this->wait();
```

Y por último, podemos ver cómo funcionaría la parte productor-consumidor (Sin considerar el mutex o la finalización por simplicidad) en un diagrama de secuencia:

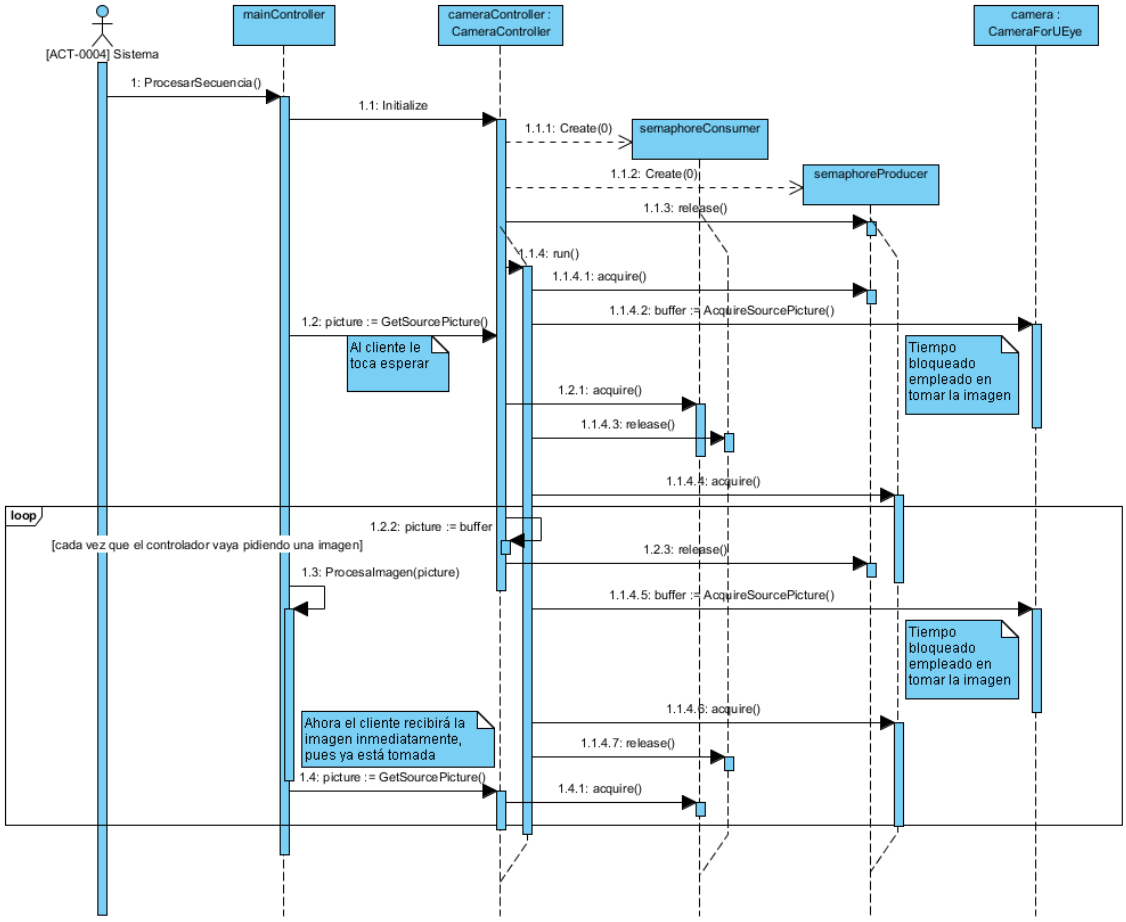

Figura 41. Diagrama de Secuencia: Hilo de toma de imágenes

5.3. Representación gráfica en la interfaz principal

Cuando se deben representar resultados que proceden de un cálculo previo hay dos opciones siendo la primera de ellas la siguiente:

- Cada vez que se modifica algún parámetro se realizan todos los cálculos.
- Para representar un gráfico, simplemente se recuperan los resultados del cálculo.

Sin embargo esa forma de trabajar es muy costosa para nuestra aplicación, ya que implicaría, por ejemplo, aplicar la lista de algoritmos a todas las imágenes cada vez que se modifica un solo parámetro de esa lista, cuando muchas veces, el resultado no nos interesará hasta haber variado una serie de de parámetros.

Para evitar eso, la forma de calcular es la siguiente:

- Cada vez que se modifica algún parámetro que afecte al cálculo, se guarda.
- Para representar un resultado, se pide al controlador el resultado de cierta imagen de la lista. El controlador sabrá entonces si tiene que realizar los cálculos para esa imagen, y si los tiene que realizar, lo hará antes de devolver el resultado.

5.3.1. Ejemplo de representación

En este sub-apartado se verá un ejemplo de la política de cálculo del programa explicada anteriormente. Suponemos que el usuario modifica un algoritmo y ha pasado a estar en una vista en la que el espectrograma está visible con una imagen seleccionada.

En ese caso, la secuencia de sucesos que ocurrirá, en la que se puede observar el comportamiento anterior, será la siguiente:

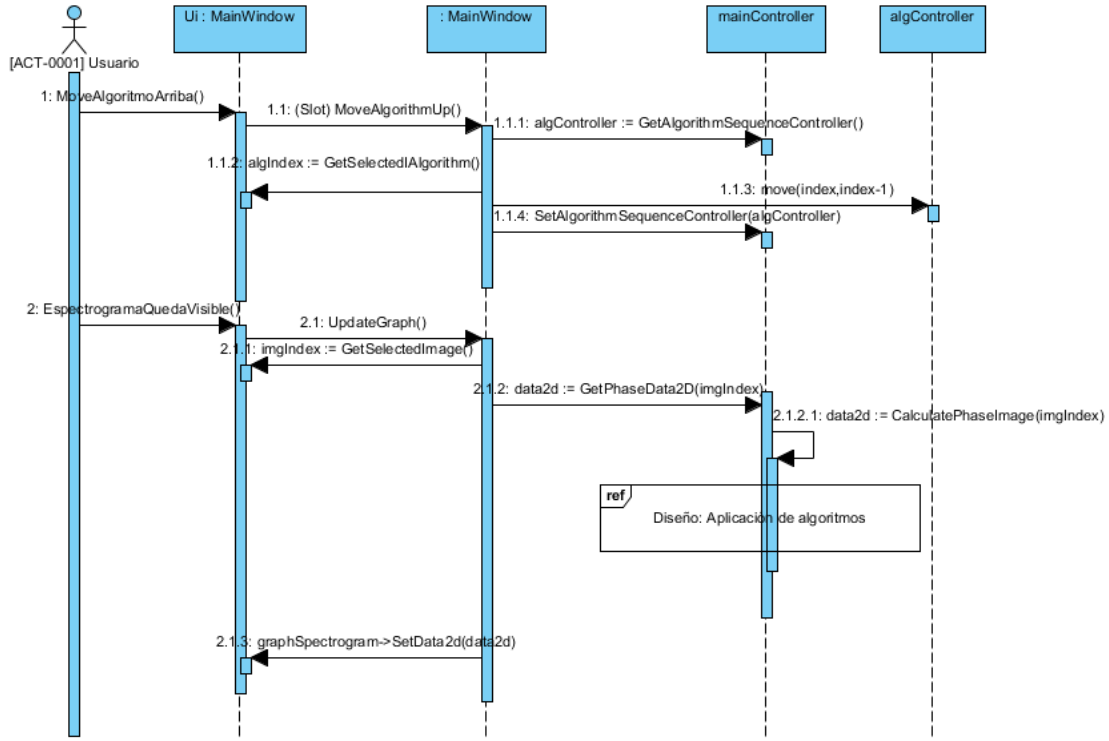

Figura 42. Diagrama de Secuencia: Representar gráficos en la ventana principal

5.4. Gestión de proyecto

En este apartado se describe el proceso de guardar y cargar el proyecto completo de un archivo.

Para cada uno de estos casos se muestra un diagrama de secuencia.

En general el guardado proceso consiste en:

1. Primero obtener la ruta donde se debe guardar.
2. Después guardar los archivos necesarios para el proyecto en una carpeta temporal.
3. Comprimir todos los archivos de la carpeta temporal en un archivo ZIP con extensión “.jpp” en la ruta indicada.

Para la carga, el proceso es análogo, descomprimiendo el proyecto en una capeta temporal y leyendo los archivos.

5.4.1. Guardar proyecto

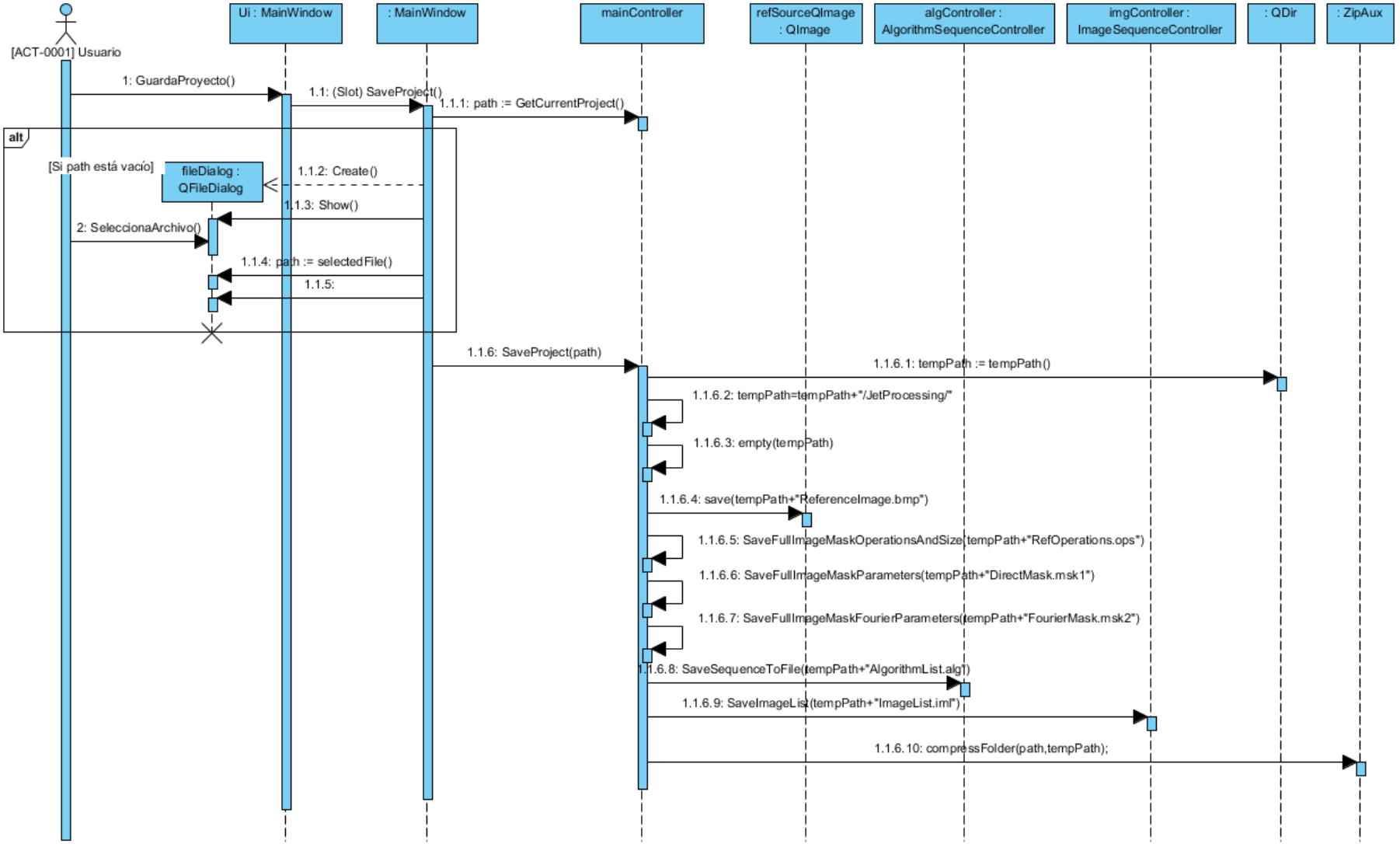

Figura 43. Diagrama de Secuencia: Guardar Proyecto

5.4.2. Cargar proyecto

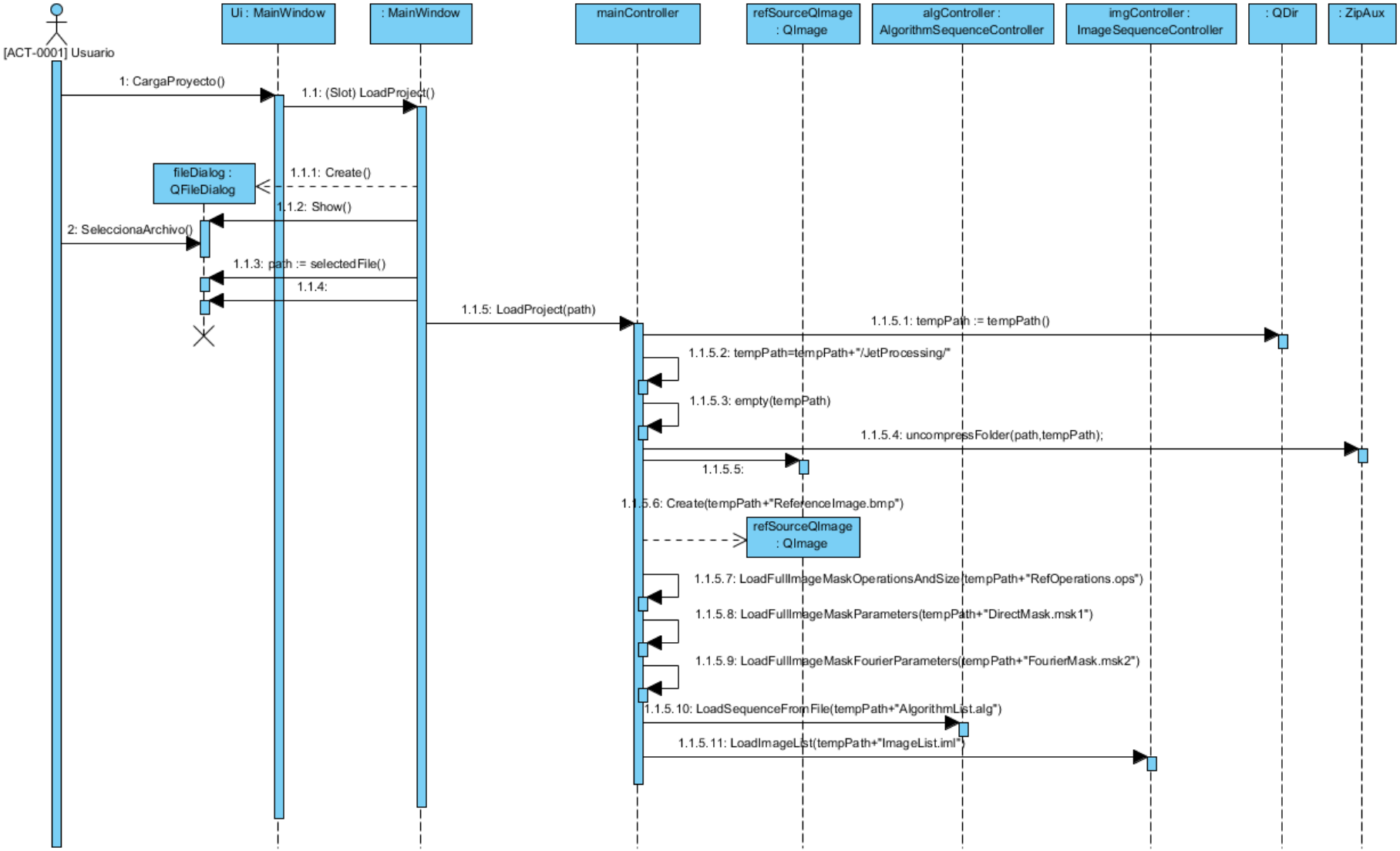

Figura 44. Diagrama de Secuencia: Cargar Proyecto

5.5. Modo "en vivo"

El modo "en vivo" consiste en la toma continua de imágenes y su procesado con la referencia.

La forma más sencilla de realizar eso es que el controlador de la interfaz pida al controlador principal que procese una imagen, el controlador principal la devuelva, y la interfaz la muestre.

No obstante, en esta aplicación para la que se busca una frecuencia de procesamiento lo más alta posible, esa forma de trabajar no es viable, ya que la mayor parte del tiempo tendría el control el controlador principal realizando el procesamiento, y el controlador de la interfaz se quedaría bloqueado, recibiendo el control durante sólo unos pocos milisegundos cada segundo, lo que haría imposible recibir y procesar acciones del usuario de forma suave y agradable para la experiencia de éste, como por ejemplo al girar un gráfico 3D.

5.5.1. Procesado de imágenes de la cámara en vivo: Hilo independiente

La forma de evitar el problema anterior es, de nuevo, recurrir a los hilos.

La secuencia será la siguiente:

- El controlador de la interfaz establece un temporizador para representar una imagen.
- Cuando llega una llamada del temporizador, un semáforo indicará si hay algo procesándose:
    - Si no se está procesando nada, se crea una instancia en otro hilo de la clase DataProducer, que mandará al controlador que tome una imagen y la procese.
    - Si se está procesando algo se programa otro temporizador para dentro de unos segundos, ya que sólo queremos estar procesando una cosa a la vez.

- Cuando el hilo que procesa acaba, envía una señal al controlador de la interfaz avisando de que ya tiene los datos, y se destruye, comenzando el ciclo de nuevo.
- El controlador de la interfaz recibe la señal y representa los datos.

Ése comportamiento se muestra en el siguiente diagrama:

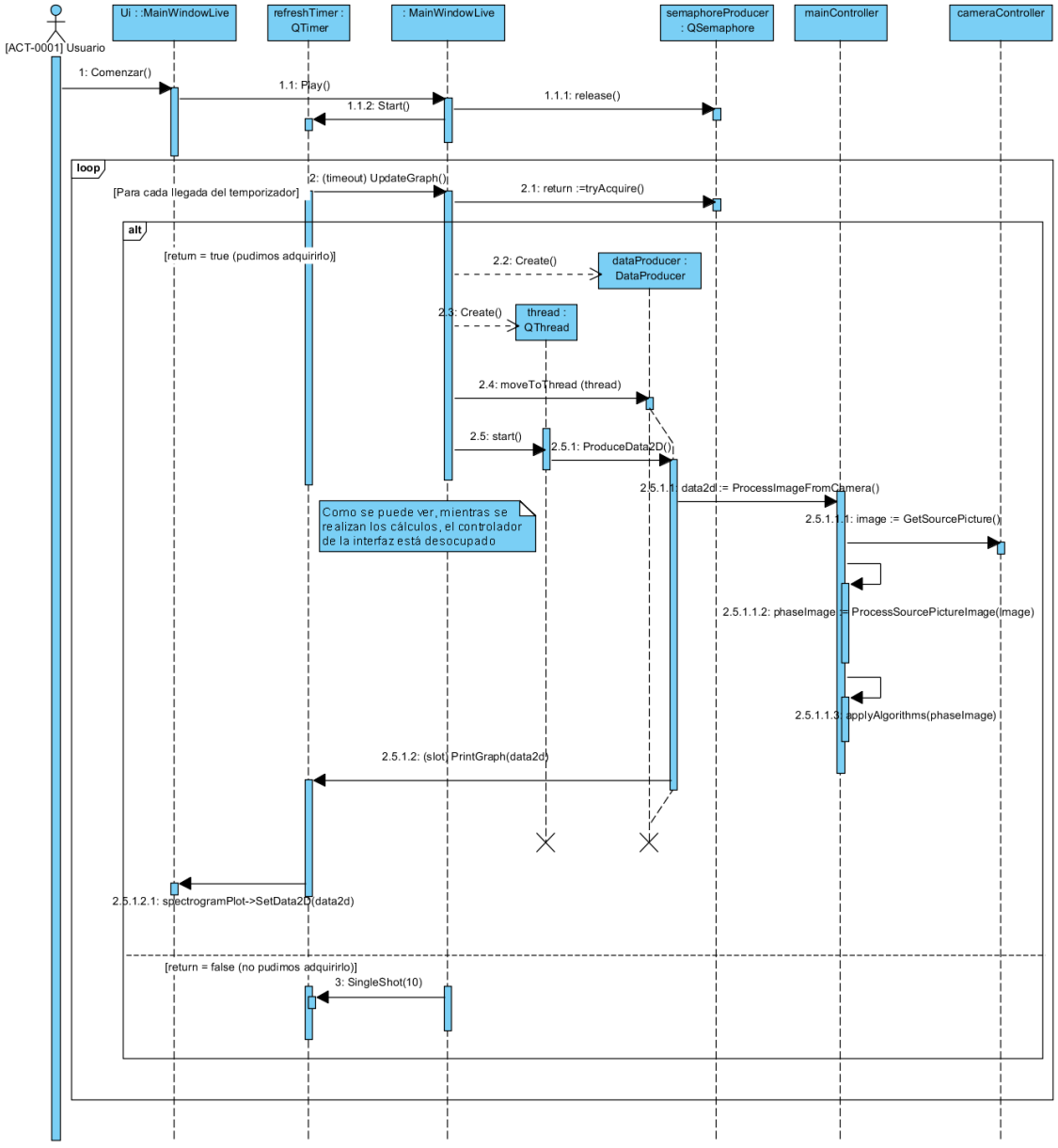

Figura 45. Diagrama de Secuencia: Procesado de imagen en modo "en vivo"

Como se puede observar, a partir de que el controlador de la interfaz llama a 2.5 start(), el procesamiento ocurre en un hilo diferente, y el controlador de la interfaz

está libre para recibir acciones del usuario, estando ocupado sólo durante el tiempo necesario para la representación del gráfico, tiempo lo suficientemente pequeño como para que no afecte a la experiencia del usuario.