Universidad de La Laguna

Escuela Superior de Ingeniería y Tecnología

Sección de Ingeniería Informática

Trabajo de Fin de Grado

Entorno de depuración de algoritmos de procesamiento de imágenes para Java

Debugging environment of Image Processing Algorithms for Java

Pedro Javier Núñez Rodríguez

Escuela Técnica Superior de Ingeniería Informática

La Laguna, 8 de julio de 2015

D. **José Francisco Sigut Saavedra**, con N.I.F. 43786043-T, profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor.

D. **Francisco José Fumero Batista** con N.I.F. 45731321-F, adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor.

C E R T I F I C A N

Que la presente memoria titulada:

"*Entorno de depuración de algoritmos de procesamiento de imágenes para Java*"

ha sido realizada bajo su dirección por D. Pedro Javier Núñez Rodríguez, con N.I.F. 54055388-Y.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 8 de julio de 2015.

Agradecimientos

Don José Francisco Sigut Saavedra por su apoyo con el proyecto y su dirección.

Don Francisco José Fumero Batista por su codirección y su implicación en el proyecto.

La Universidad de La Laguna por brindarme la oportunidad de desarrollar este proyecto.

Mis familiares y amigos por el apoyo recibido durante el desarrollo del mismo.

Licencia

© Esta obra está bajo una licencia de Creative Commons
Reconocimiento-NoComercial-CompartirIgual 4.0
Internacional.

// Resumen

El objetivo principal de este proyecto ha sido crear un entorno de depuración orientado al lenguaje Java que nos permita trabajar con algoritmos de procesamiento de imágenes.

Actualmente cuando depuramos un programa en Java que contiene variables de tipo imagen, si nos detenemos en un punto de ruptura y las observamos, no vemos más que valores numéricos, lo cual resulta muy poco intuitivo. Además, esto dificulta notablemente el seguimiento de los cambios producidos como consecuencia del procesamiento que se esté llevando a cabo.

Para solucionar este inconveniente, se ha procedido a crear un entorno de depuración sencillo, que pueda ser utilizado en aplicaciones programadas en Java. Este entorno permite el uso de puntos de ruptura e implementa funciones para hacer un seguimiento de las operaciones que se realizan sobre un píxel determinado de la imagen o sobre una región de la misma, mostrando los resultados de una manera gráfica y fácil de utilizar por la persona que esté depurando el algoritmo.

Palabras clave: Depurador, Imágenes, Java.

Abstract

The main objective of this project has been to create a debugging enviroment for the Java language which allows us to work with image processing algorithms.

Currently, when we debug a Java program which contains image variables, if we stop in a breaking point and we see it, we only see numerical values, which are not intuitive. Besides, this is a problem because it hampers to follow the changes realized by the processing.

This disadvantage was solved through the creation of an easy debugging environment which could be used in applications written in Java. This environment allows the use of breaking points and it implements functions to follow up the operations carried out on a certain pixel or area in the image. The results are shown in an easy and graphic way which can be used by the person debugging the algorithm.

Keywords: Debbugger, Images, Java.

Índice

Capítulo 1 2
1. Introducción 2
1.1. Antecedentes 4
1.2. Objetivos 6
1.3. Expectativas 6
Capítulo 2 7
2. Estudio de viabilidad 7
2.1. Java 7
2.2. Eclipse 8
2.3. Integración con el entorno de depuración de Eclipse 8
2.4. Problemas encontrados 11
Capítulo 3 13
3. Desarrollo del proyecto 13
3.1. Arquitectura de programación 13
3.2. La interfaz 15
3.3. Puntos de ruptura 17
3.4. Funciones 18
3.4.1. Barra de estado 19
3.4.2. Barra de Herramientas 19
3.4.2.1. Save 20
3.4.2.2. Save As 20
3.4.2.3. Stop 20
3.4.2.4. Resume 21
3.4.2.5. Seguimiento de un pixel 22
3.4.2.6. Recortar imagen 23
Capítulo 4 24
4. La librería 24
4.1. Creación de la librería 24
4.2. Cargar la librería 25
4.3. Usando la librería API 26
4.4. Librería independiente del entorno 26

Capítulo 5 27
5. Conclusiones y líneas futuras 27
5.1. Conclusiones 27
5.2. Líneas futuras 28
Capítulo 6 29
6. Summary and conclusions. 29
6.2. Future work 30
Presupuesto 31
Bibliografía 32

Índice de figuras

Figura 1. Ejemplo de imagen digital 2
Figura 2. Vista de depuración de una variable de tipo imagen con Eclipse 3
Figura 3. Código Formatter 4
Figura 4. Vista variables con Detail Formatter para BufferedImage 5
Figura 5. Logo Java 7
Figura 6. Logo Eclipse 8
Figura 7. Código para añadir vista Image a una perspectiva 9
Figura 8. Perspectiva Debug Image 10
Figura 9. Constructor del Activator del plugin 10
Figura 10. Código BreakpointLister para el plugin de Eclipse 12
Figura 11. Diagrama UML 14
Figura 12. Método que construye el canvas donde se dibuja 15
Figura 13. Método paintComponent 16
Figura 14. Vista de la interfaz 17
Figura 15. Constructor de la clase debugger 17
Figura 16. Método para establecer punto de ruptura 18
Figura 17. Evento de movimiento de ratón 19
Figura 18. Barra de herramientas del debugger 19
Figura 19. Evento botón de guardado 20
Figura 20. Confirmación salida Debugger 21
Figura 21. Seguimiento de un pixel 22
Figura 22. Exportando JAR file 24
Figura 23. Añadiendo Librería al proyecto 25
Figura 24. Ejemplo de proyecto que incorpora la librería en Eclipse 25

Capítulo 1

1. Introducción

Los entornos de depuración de código disponibles para Java suelen ser de propósito general, permitiendo depurar cualquier tipo de aplicaciones, sin tener en cuenta la interpretación que se le da a la información contenida en las variables o a los algoritmos que se están depurando.

En el caso particular de los algoritmos de procesamiento de imágenes, se hace especialmente necesario disponer de herramientas que faciliten la depuración cuando se ven involucradas variables de tipo *imagen*.

Una imagen digital no es más que un conjunto de valores ordenados de forma matricial. Cada uno de estos valores tiene codificado en su interior una serie de datos que nos indican el color final de ese punto o *pixel* en la imagen.

En el contexto de este proyecto, la codificación usada para los pixeles será el modelo RGB, que se trata de la composición del color en términos de intensidad de los colores primarios de la luz. El modelo RGB se basa en la síntesis aditiva, con la que es posible representar un color mediante la mezcla por adición de los tres colores de luz primarios: Red, Green y Blue.

Figura 1. Ejemplo de imagen digital

En la imagen de la Figura 1, si analizáramos un pixel de la zona de la pupila, éste sería un valor entero que tendría codificados unos valores RGB cercanos a R=0, G=0 y B=0, que simbolizan el color negro. Sin embargo, a la hora de depurar un programa que trabaje con imágenes, el problema que tenemos es que la información que se muestra es muy poco intuitiva.

En la figura 2 se muestra cómo se accede en el depurador de Eclipse a la información contenida en los pixeles de una variable de tipo *imagen*. Como podemos observar, aparece una lista desplegable con todos los atributos de la clase que representa a la imagen, en la que hemos seleccionado el valor del primer pixel de la misma. Como ya hemos dicho, este valor está codificado, por lo que no sabemos directamente a qué valores RGB corresponde.

Por otro lado, para acceder a cada pixel hay que navegar por las distintas listas, por lo que no es sencillo saber el valor RGB de unas coordenadas concretas de la imagen, ni tampoco hacer un seguimiento de los cambios producidos como consecuencia del procesamiento que se esté llevando a cabo.

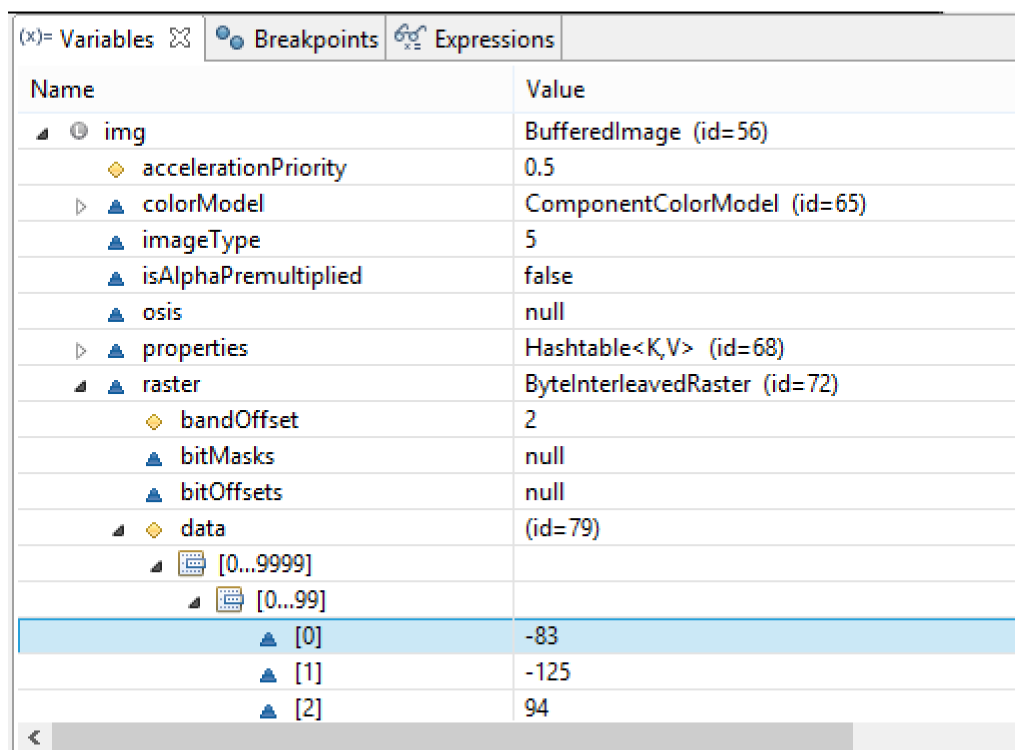

Figura 2. Vista de depuración de una variable de tipo imagen con Eclipse

Para solucionar este inconveniente, en este trabajo hemos desarrollado un entorno de depuración sencillo, que pueda ser utilizado en aplicaciones programadas en Java. Este entorno permite el uso de puntos de ruptura e implementa funciones para hacer un seguimiento de las operaciones que se realizan sobre un pixel determinado de la imagen o sobre una región de la misma, mostrando los resultados de una manera gráfica y fácil de utilizar por la persona que esté depurando el algoritmo.

1.1. Antecedentes

Hasta el momento de realizar este trabajo, no hemos podido encontrar ningún entorno de depuración ni librería para Java que nos permita mostrar una imagen a medida que un algoritmo trabaja con ella.

Si bien hay entornos de procesamiento de imágenes basados en Java, como ImageJ, que permiten escribir y depurar macros, éstas tienen que estar escritas en un lenguaje propio, por lo que nos distanciamos del objetivo de este proyecto, que requiere poder programar directamente en Java.

Así mismo, tampoco hemos encontrado un *plugin*[1] para Eclipse que permita realizar esta tarea. Hasta ahora cuando se tenía que analizar una imagen, en este entorno de desarrollo, se hacía mediante pequeños trucos como puede ser el uso de un *Detail Formatter*.

Al usar el *debbuger*[2] de Eclipse y detenernos en un punto de ruptura, cuando intentamos inspeccionar la variable en cuestión, vemos que para intentar acceder a los valores RGB de esta imagen tenemos que acceder al atributo `raster` (Fig. 2), y buscar los valores que están cambiando dentro de *arrays* de valores numéricos. Un procedimiento engorroso y que repetiríamos para cada punto de ruptura.

Con el uso del *Detail Formatter*, nombrado anteriormente, podemos mostrar los valores RGB de una imagen en un punto de ruptura del *debugger* de eclipse.

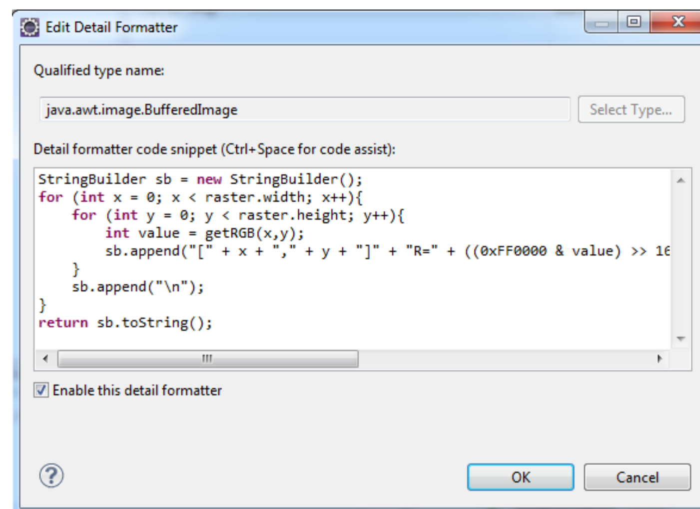

Figura 3. Código Formatter

---

1 es un programa que incrementa o aumenta las funcionalidades de un programa principal.
2 es un programa que permite probar y corregir los errores de otros programas.

El procedimiento para crear un *Detail Formatter* es simple. Sólo debemos pulsar sobre la variable que queremos controlar sobre la vista de variables y con el botón derecho elegir la opción "*Edit Detail Formatter*". A partir de aquí no es más que programar un pequeño código como el mostrado en la figura 3 que se ejecuta en cada punto de ruptura. En este caso nuestro código nos proporciona para cada pixel sus coordenadas y sus valores RGB, como se puede ver en la figura 4.

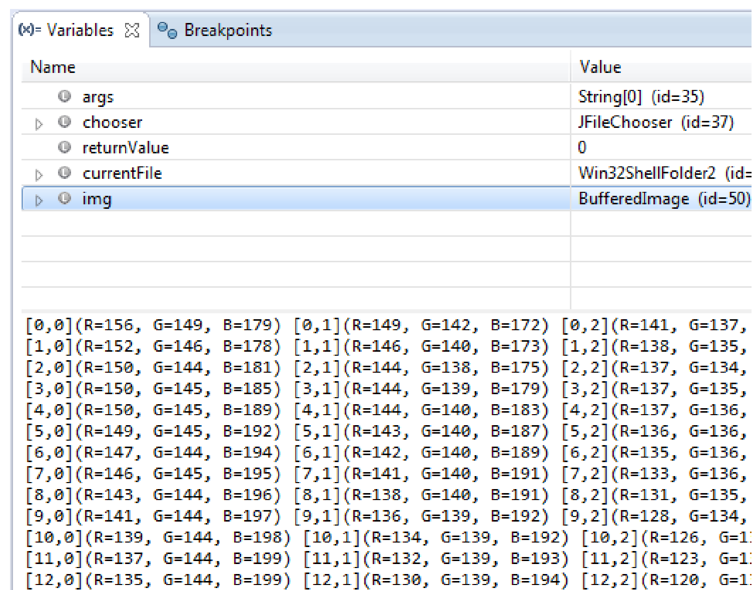

Figura 4. Vista variables con Detail Formatter para BufferedImage

Aunque es un truco que nos permite ver los valores de las componentes RGB de una imagen, tiene limitaciones, por ejemplo, si usamos una imagen muy grande producirá un error y no nos mostrará nada. Otro inconveniente es que solo nos muestra datos numéricos por consola, lo cual es poco intuitivo y nada práctico si queremos conocer el valor de pixeles exactos por ejemplo.

Entonces, como se indica, el Detail Formatter se queda corto a la hora de analizar cómo está siendo procesada la imagen sobre la que trabajamos. Como ingeniero, es necesario preguntarse cómo podemos resolver este problema, dado que nuestra misión principal consiste en la resolución de éstos. Por ello, se considera que la respuesta más fácil que se puede aplicar sería, poder ver la imagen, así como también, trabajar con ella, con el fin de realizar un análisis rápido de la evolución del algoritmo. A continuación, se definirán los objetivos.

1.2. Objetivos

En primer lugar, el principal objetivo de este proyecto ha sido el proporcionar una herramienta que nos permita mostrar la imagen sobre la que está trabajando el algoritmo en tiempo de ejecución, estableciendo un mecanismo de puntos de ruptura para poder parar en la parte del algoritmo que nos interese.

En segundo lugar, otro objetivo importante ha sido habilitar el acceso a los valores RGB de los pixeles de la imagen o guardar la imagen en un determinado punto para compararla más tarde.

Por último, se completan unos objetivos más específicos, haciendo hincapié en la facilidad de uso, añadiendo funciones para el seguimiento de zonas de la imagen así como de pixeles, que en un momento determinado necesitemos.

1.3. Expectativas

Elegí este proyecto porque me pareció interesante el tema del lenguaje Java y el diseño de interfaces. Aunque la que se desarrolla aquí no es una interfaz complicada, me permite profundizar en el tema, así como en la visión por computador que, a mi parecer, es un tema muy interesante y de actualidad. Por otro lado se ha trabajado sobre el entorno eclipse, un *IDE*[3] con el que estoy bastante familiarizado y del cual me ha permitido explorar nuevas posibilidades.

[3] (Integrated development environment) es una aplicación de software, que proporciona servicios integrales para facilitarle al programador el desarrollo de software.

Capítulo 2

2. Estudio de viabilidad

Llegados a este punto, es necesario explorar las distintas opciones que tenemos para abordar el problema. De entrada nos planteamos trabajar con el IDE Eclipse, ya que el proyecto se basa en el lenguaje JAVA. En cuanto a las posibles soluciones, se barajan dos posibilidades: una aplicación independiente o un plugin para el entorno Eclipse.

2.1. Java

Figura 5. Logo Java

Java es un lenguaje de propósito general, concurrente, orientado a objetos que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible.

El lenguaje de programación Java fue originalmente desarrollado por James Gosling de Sun Microsystems (la cual fue adquirida por la compañía Oracle) y publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems. Su sintaxis deriva en gran medida de C y C++, pero tiene menos utilidades de bajo nivel que cualquiera de ellos.

Una de las principales características por las que Java se ha hecho muy famoso es que es un lenguaje independiente de la plataforma. Esto es una gran ventaja porque el programa funcionará independientemente de la máquina donde se ejecute. Esto lo consigue porque se ha creado una Máquina de Java para cada sistema que hace de puente entre el sistema operativo y el programa de Java y posibilita que este último funcione perfectamente.

2.2. Eclipse

Figura 6. Logo Eclipse

Eclipse es una plataforma de desarrollo de código abierto basada en Java.

Sus principales características es que se basa en un sistema de perspectivas, que no son más que unas configuraciones determinadas según lo que queramos trabajar. Así pues, por ejemplo, dispone de una perspectiva para usar el potente *debugger* de que dispone, o para trabajar con otros lenguajes de programación como puede ser C++ o C.

El trabajo en Eclipse se organiza mediante proyectos que aglutinan toda la información sobre ese proyecto en un directorio de carpetas organizado automáticamente.

Por último, su característica más destacada quizás sea que es de código abierto y esto abre la puerta a infinidad de *plugins* que permiten personalizar aún más el IDE según nuestras necesidades.

2.3. Integración con el entorno de depuración de Eclipse

Como primera solución escogimos hacer un *plugin* para Eclipse, debido a que pensamos que era lo más eficiente y también lo más complicado, ya que, actualmente no encontramos ningún *plugin* que nos sirviera de alguna manera para resolver el problema de hacer el seguimiento o depuración de una imagen.

Principalmente se trataba de implementar una nueva perspectiva en Eclipse, llamada Debug Image, que nos mostrara en el modo debugger una vista Image, junto con la vista variables, en el momento de ejecutarla, en esa vista se debería mostrar la imagen en el punto de ruptura actual.

El Eclipse gracias a ser de código abierto dispone de un *plugin* para que nos proporciona las herramientas para crear otros *plugins*, valga la redundancia. Al descargar este *plugin* nos encontramos con que tenemos una nueva perspectiva específica que te ayuda empezar un *plugin* para eclipse con distintas plantillas muy útiles para entender el funcionamiento básico.

Así mismo se usa la *Java Development Tools (JDT)*, en concreto la librería *JDT Debug* que implementa las interfaces de la *Java Platform Debugger Architecture*, con las que accederemos a los métodos que necesitamos para la creación del *plugin*.

A la hora de empezar a programar un *plugin*, por ejemplo haciendo uso de las plantillas antes mencionadas, nos encontramos con que el denominador común de cualquier *plugin* es la clase `Activator`, que es la encargada del ciclo de vida del mismo.

La implementación de nuevas características se basa en añadir funcionalidad a distintos *extensions points*. Para añadir funcionalidades nuevas, esto se hace mediante `xml` en el fichero plugin.xml (Fig. 7).

```
<extension point="org.eclipse.ui.perspectiveExtensions">
      <perspectiveExtension
       targetID="debugimage.perspectives.DebugI">
      <view
          id="debugimage.ImageV"
          relationship="top"
          relative="org.eclipse.ui.perspectives">
      </view>
</perspectiveExtension>

</extension>
<extension point="org.eclipse.ui.views">
      <view id="org.eclipse.ui.views.ImageV"
          name="Image View"
          class="debugimage.ImageView"
          icon="icons\image_obj.gif"/>
</extension>
```

Figura 7. Código para añadir vista Image a una perspectiva

El Eclipse trabaja de forma que los *plugins* tienen una parte declarativa, la que vimos anteriormente en `xml` y una parte de código (las clases que implementan), de manera que los *plugins* solo cargan el código cuando este es necesario porque el usuario activa alguna opción que los llama. De esta manera aumenta la eficiencia, por esto cuando empezamos a crear un *plugin*, en la página de Eclipse se recomienda que nunca se inicie un plugin nada más iniciar el Eclipse.

Para nuestro proyecto deseamos crear una perspectiva nueva llama DebugImage para ello deberemos añadir una nueva vista, la cual será la encargada de dibujar la imagen, así como una perspectiva nueva para colocar las ventanas (Fig. 8) y las opciones de Eclipse que mejor convengan para nuestro cometido.

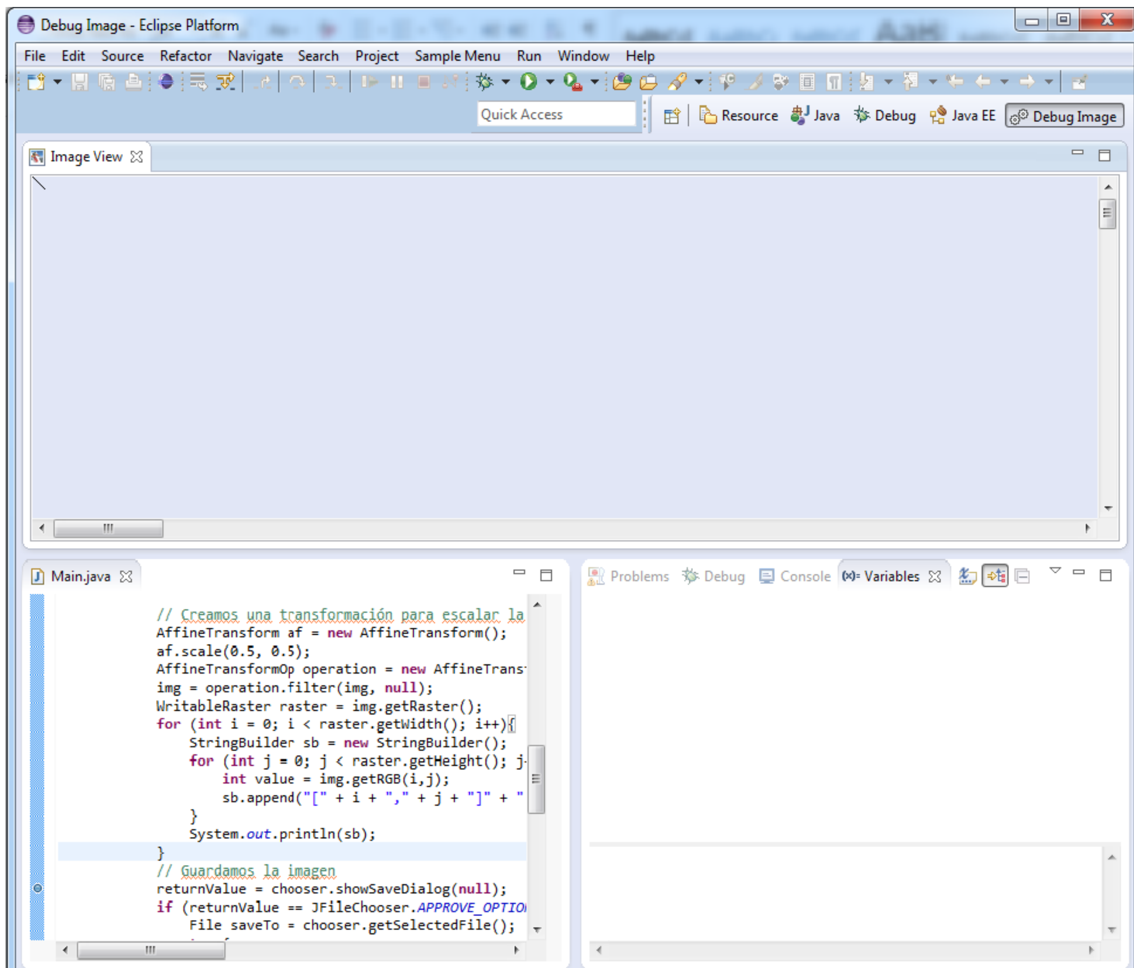

Figura 8. Perspectiva Debug Image

Una vez tenemos nuestra perspectiva, tenemos que modificar los *breakpoints* para que al llegar a un punto de ruptura, el programa se pare y si está en nuestra perspectiva acceder a las variables de tipo imagen y mostrarlas en la respectiva vista. Para ello debemos modificar el punto de extensión `"org.eclipse.jdt.debug.breakpointListeners"` añadimos un nuevo *point listener* (código en la figura 9).

Tenemos que añadir el nuevo *listener* al *debugger* para que al llegar a un punto de ruptura nos entre por nuestro código. Esto lo hacemos dentro del `activator` del *plugin*, que se activará cuando entren en nuestra perspectiva.

```
public Activator(){
    ImgView = new ImageView();
    ImgListener = new ImgBreakpointListener();
    JDIDebugModel.addJavaBreakpointListener(ImgListener);
}
```

Figura 9. Constructor del Activator del plugin

2.4. Problemas encontrados

Llegados a este punto de la implementación, nos encontramos con el problema de acceder a la variable imagen que está ejecutándose en ese momento. Y es que para acceder a esa información, directamente no se puede acceder a la variable tipo `BufferedImage`, ya que ésta se almacena en la pila de ejecución de Eclipse, y si intentamos acceder a ella no nos la reconoce, porque solo se guardan en la pila como tipos básicos.

Buscando solucionar el problema se intenta acceder a los tipos básicos para reconstruir la imagen desde el `raster`. En la pila del sistema, el `raster` se guarda en un `InterLeavedRaster`. Este tipo guarda los valores RGB de manera consecutiva, con lo cual una imagen de 100x100 pixeles en realidad será un *array* 300x300 de valores enteros, a los que debemos acceder, a parte del resto de información necesaria para reconstruir este tipo imagen, con lo cual la ejecución de este tipo de cálculo no será eficiente, el tiempo para reconstruir la imagen es demasiado y el programa permanecería ejecutando estos bucles durante demasiado tiempo.

Este bucle lo podemos ver en la figura 10 de la siguiente página, en el recuadro rojo: accedemos para cada iteración 3 veces, uno para cada componente de color, así mismo nos percatamos de que los valores en la pila de ejecución se guardan como `String` y por ello tenemos que convertirlos a los correspondientes tipos, y reconstruir cada valor RGB a partir de las 3 componentes leídas. Todo esto hace imposible que esta manera sea una forma eficiente de resolver el problema.

Por último, se busca acceder a la variable tipo `BufferedImage` de una manera más directa, para ahorrarnos la conversión anterior, pero ésta se guarda en una máquina virtual distinta a donde se ejecuta el programa actual, con lo cual es imposible acceder a ella.

Por todo lo explicado anteriormente se opta por desarrollar una API independiente que nos permita alcanzar los objetivos de este proyecto.

```java
int height = 0;
int width = 0;
int numBands = 0;
IVariable[] map = null;
RGB[] pixels = null;
try {
    this.setData(thread.getStackFrames());

    IVariable[] ImageData = getData()[0].getVariables();
    for (int i = 0; i < ImageData.length;i++){

        if (ImageData[i].getReferenceTypeName().equals("java.awt.image.BufferedImage")){

            for (int j = 0; j < ImageData[i].getValue().getVariables().length;j++){

                if (ImageData[i].getValue().getVariables()[j].getName().compareTo("raster") == 0){
                    IVariable[] RasterData = ImageData[i].getValue().getVariables()[j].getValue().getVariables();
                    System.out.println ("Examinando Raster... ");

                    for (int k = 0; k < RasterData.length;k++){
                        System.out.println(" Varible " + k + " " + RasterData[k].getName());

                        if (RasterData[k].getName().compareTo("height") == 0)
                            height = Integer.parseInt(RasterData[k].getValue().toString());

                        if (RasterData[k].getName().compareTo("width") == 0)
                            width = Integer.parseInt(RasterData[k].getValue().toString());

                        if (RasterData[k].getName().compareTo("numBands") == 0)
                            numBands = Integer.parseInt(RasterData[k].getValue().toString());

                        if (RasterData[k].getName().compareTo("data") == 0){
                            map = RasterData[k].getValue().getVariables();

                        }
                    }
                    int tam = height*width;
                    int y = 0;
                    int r = 0;
                    int g = 0;
                    int b = 0;
                    pixels = new RGB[tam];
                    for(int x = 0; x < map.length ; x=x+numBands){
                        b = Integer.parseInt(map[x].getValue().toString());
                        g = Integer.parseInt(map[x+1].getValue().toString());
                        r = Integer.parseInt(map[x+2].getValue().toString());
                        RGB pixel = new RGB((0x0FF & r),(0x0FF & g),(0x0FF & b));
                        pixels[y] = pixel;
                        y++;
                    }
```

Figura 10. Código BreakpointLister para el plugin de Eclipse

Capítulo 3

3. Desarrollo del proyecto

A partir de este punto se procede al desarrollo del proyecto, ya que se ha optado al final por un entorno independiente con una API para el programador. Se tratará de una librería que podremos añadir a nuestros programas para hacer uso de esta interfaz para depurar imágenes.

3.1. Arquitectura de programación

Se plantean 3 clases principales. La primera es la clase `ImageDebugger` que hereda de la clase `JFrame`, encargada de soportar la interfaz gráfica, que a su vez tiene una clase interna `actionListener`, encargada de manejar todos los eventos de la misma. Así mismo también implementa la interface `windowsListener`, que nos proporciona los métodos para el manejo de ventanas, tales como por ejemplo los eventos para el cierre de ventanas.

La segunda es la clase `ImageCanvas` que hereda de la clase `JPanel`, encargada de pintar las imágenes mediante el método `paint` de la clase `Graphics` y controlar todo lo necesario para la correcta visualización de la imagen, controlando siempre el repintado para que ésta se muestre como es debido. Así mismo maneja eventos de ratón, teniendo que implementar las interfaces `MouseListener` y `MouseMotionListener`, necesarias para obtener datos para funciones más específicas de las que hablaremos más adelante.

Por último, tenemos la clase `Debug`, que será la encargada de controlar la ejecución de los hilos pertinentes y de proporcionar métodos para que la librería sea llamada en los programas que queremos depurar.

En la página siguiente se muestra un diagrama UML con las clases, sus atributos y los principales métodos de cada una de ellas (Fig. 11).

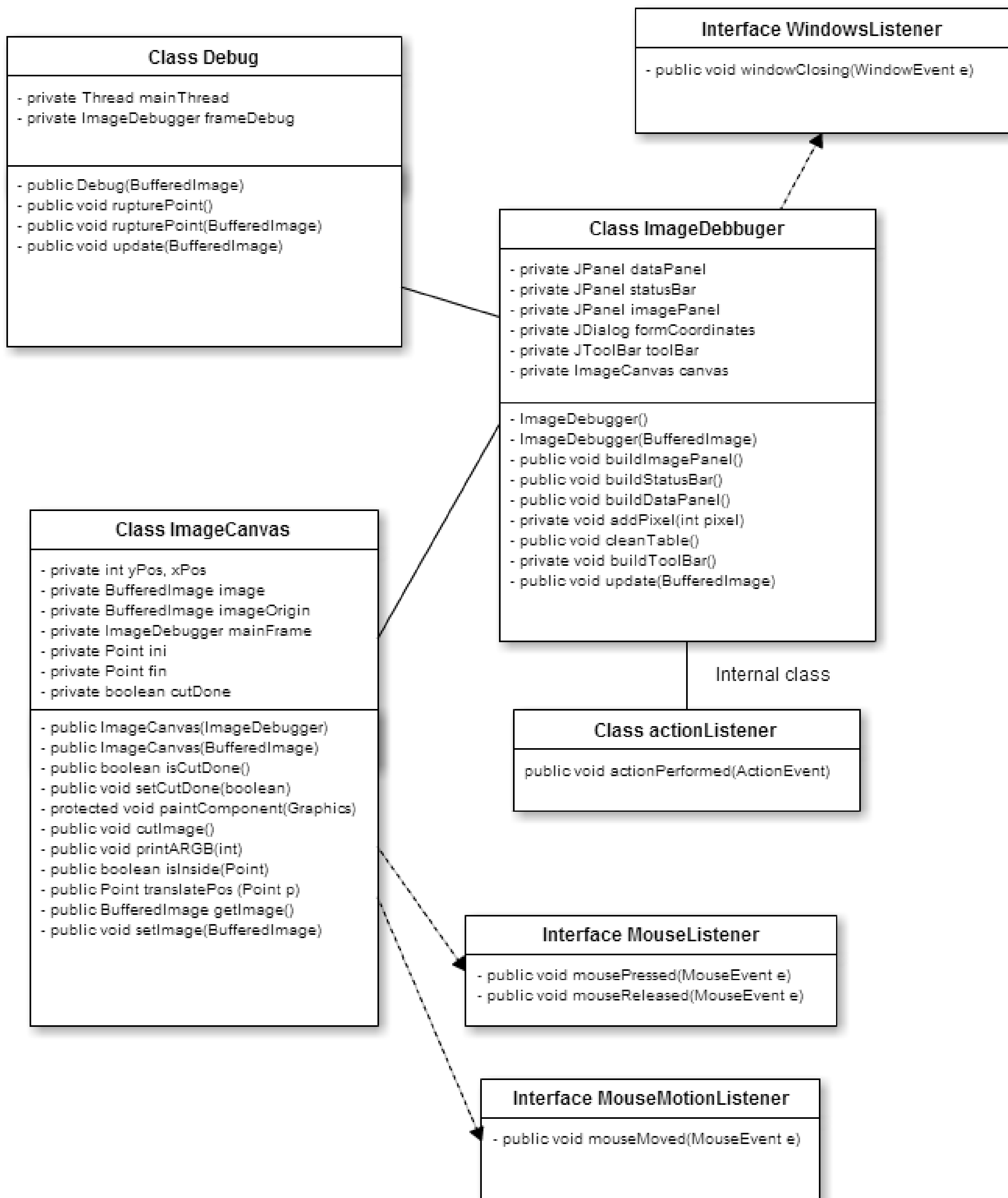

Figura 11. Diagrama UML

3.2. La interfaz

El primer punto a tener en cuenta es el diseño de la interfaz. Tenemos que encontrar la manera de dibujar una imagen, para ello haremos uso de la clase `JPanel` que dispone de métodos que nos permiten usar un panel como un lienzo donde dibujar, que es lo que nosotros buscamos, dibujar imágenes, para luego trabajar sobre ellas.

```
public void buildImagePanel(){
    imagePanel = new JPanel();
    canvas = new ImageCanvas(this);
    scrollPane = new JScrollPane(canvas);
    scrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
    scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
    imagePanel.setLayout(new BorderLayout());
    imagePanel.add(scrollPane,BorderLayout.CENTER);

}
```

Figura 12. Método que construye el canvas donde se dibuja

El diseño de la interfaz tendrá que tener un *frame*, una ventana que albergue toda la interfaz, que extenderá la clase `JFrame` y dentro de éste tendremos una serie de paneles que albergarán las distintas funciones que tiene el depurador. Centrémonos ahora en la más importante que es la de dibujar una imagen. Para ello se usa una clase que extiende de `JPanel`, con lo que podremos sobrescribir el método `paintComponent` (Fig. 13).

Hay que tener en cuenta varias consideraciones a la hora de dibujar:

- El tamaño del "lienzo", nunca va a ser igual que la imagen o es poco probable, por lo que habrá que calcular el punto inicial de la imagen para que ésta se muestre correctamente centrada, en el caso de que sea menor que el tamaño de la zona de dibujo, así como para cuando se redimensione la ventana del *debugger*, ésta se posicione siempre en el centro del mismo.

- Si la imagen es mayor deberemos tener unos *scrollBars* que nos permitan movernos vertical y horizontalmente por toda la imagen. Éstos se añaden al panel como se muestra en la figura 12.

- Deberemos guardar las coordenadas donde se empieza a dibujar la imagen, ya que en un futuro, si queremos añadir eventos de ratón éstos se aplicarán a todo el panel, y no sabremos por ejemplo si el ratón ha entrado en la zona de la imagen, sin las coordenadas de la misma.

```java
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Dimension sizePanel = super.getVisibleRect().getSize();
    Dimension sizeImage = new Dimension(image.getWidth(), image.getHeight());

    if ((sizePanel.height) >= sizeImage.height){
        int pixels = sizePanel.height - sizeImage.height;
        yPos = pixels / 2;
    }
    else{
        yPos = 0;
    }
    if ((sizePanel.width) >= sizeImage.width){
        int pixels = sizePanel.width - sizeImage.width;
        xPos = pixels / 2;
    }
    else{
        xPos = 0;
    }
    setPreferredSize(sizeImage);    //  Tamaño preferido del panel canvas
    g.drawImage(image, xPos, yPos, this);
}
```

Figura 13. Método paintComponent

Supongamos que la imagen se pinta en las coordenadas (30,30) y lo que hace el `paintComponent` es obtener el tamaño de la imagen, y el tamaño del panel visible. Si el tamaño de la imagen es mayor que el tamaño del panel eso quiere decir que se pintará en las coordenadas (0,0) y se hará uso de los *scrollbars* que automáticamente aparecerán ya que éstos solo aparecen si la imagen es mayor que el panel. En caso contrario calcularemos las coordenadas iniciales para que la imagen quede centrada de esta manera: restamos al tamaño del panel el de la imagen y para que quede centrada dividimos esa cantidad entre dos, para que sobre la misma distancia por arriba que por debajo.

Es importante hacer también hincapié en el método `setPreferredSize`, ya que sin él, no tiene referencia de qué tamaño tenemos predefinido y cuándo la imagen es mayor que el lienzo visible. Si no ponemos este método no se nos mostrarán los *scrollbars* y por tanto quedará parte de la imagen a la que no podremos acceder.

A continuación podemos ver el resultado final de la interfaz (Fig. 14).

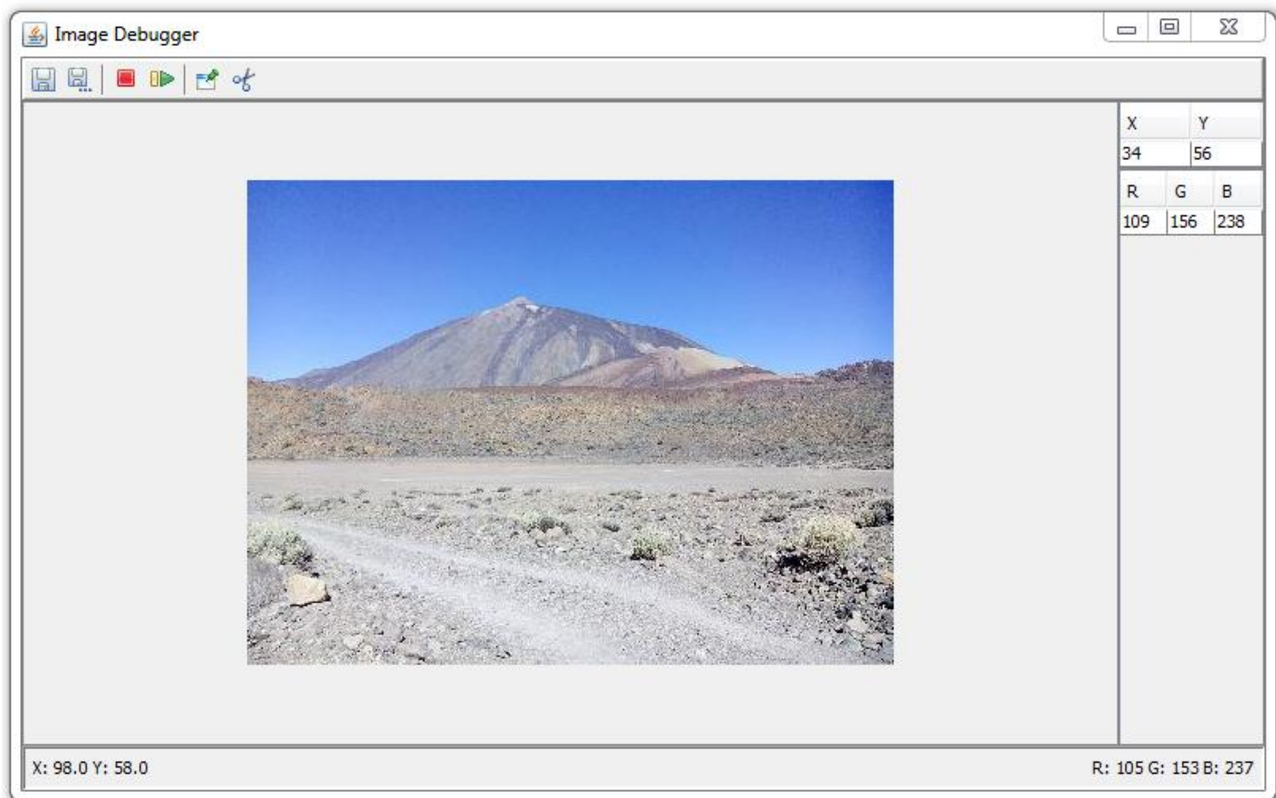

Figura 14. Vista de la interfaz

3.3. Puntos de ruptura

En un principio se optó por un `JDialog` para que cuando se abriera la ventana con la imagen se quedara bloqueado, con el *focus* en esta ventana; así nos servía como "punto de ruptura" ya que el programa no avanzaba hasta que ésta no se cerrara. Creemos que esto le restaba funcionalidad, además el *dialog* se creaba para cada nueva llamada al método que mostraba la imagen. De esta manera se opta por cambiarlo por un `JFrame`, que se crea desde que se llama al *debugger* y queda en modo oculto, hasta que se haga uso de un punto de ruptura. Cuando el programa llegue a un punto de ruptura, se para el algoritmo que estamos depurando y muestra la ventana del *debugger* que hasta ese momento estaba oculto.

```
/*
 * Constructor de clase inicializamos el frame del debugger con la imagen
 */
public Debug(BufferedImage img){
    frameDebug = new ImageDebugger();
    frameDebug.setImage(img);
    frameDebug.setMainThread(Thread.currentThread());
}
```

Figura 15. Constructor de la clase debugger

Para que esto sea viable es necesario el uso de hilos de ejecución. El mecanismo funciona de la siguiente manera: cuando se llama al constructor del *debugger* (Fig. 15), éste obtiene el hilo actual de ejecución y lo guarda, al mismo tiempo, se llama al constructor del *debugger*, que se quedara en modo oculto hasta que llegue a un punto de ruptura. Una vez en el punto de ruptura, el hilo actual de ejecución (método `currentThread`) cambiará su estado quedando en modo espera hasta que se le mande una notificación. Ésta podría ser al producirse el evento de cierre de ventana, mandamos la notificación al hilo para que continúe su ejecución y la ventana del *debugger* la ponemos de nuevo en modo oculto, con lo cual ya tendríamos nuestros puntos de ruptura (Figura 16), que paran el programa para que podamos analizar la imagen el tiempo necesario.

```
public void rupturePoint(BufferedImage image){
    mainThread = Thread.currentThread();
    try{
        synchronized (mainThread){
            update(image);
            mainThread.wait();
        }
    }
    catch (Exception e){
        e.printStackTrace();
    }
}
```

Figura 16. Método para establecer punto de ruptura

Hace falta mencionar que para que el hilo pueda entrar en modo espera es necesario sincronizarlo, así como también es necesario llamar al método `update`, para que actualice la imagen, que tiene guardada el objeto que maneja la interfaz, ya que ésta puede haber cambiado desde el último punto de ruptura.

3.4. Funciones

Después de obtener una interfaz básica para visualizar imágenes, así como un sistema de puntos de ruptura, podemos decir que la parte principal del proyecto está resuelta. Ahora falta añadir opciones que nos faciliten el depurar las imágenes, tales como seguimiento de un pixel, recortar la imagen, entre otras. Éstas se explicarán a continuación.

3.4.1. Barra de estado

El primer paso que se da en este sentido es añadir una barra de estado en la parte inferior que nos permita mediante eventos de ratón (al mover el ratón por la imagen) visualizar el pixel sobre el que estamos y las componentes RGB de ese pixel.

El principal inconveniente que se tiene que tener en cuenta aquí, es que el evento de ratón no se aplica a solo la imagen, sino a todo el panel donde se encuentra la misma, por lo que debemos asegurarnos que después de obtener las correspondientes coordenadas del ratón, éstas están dentro de las coordenadas donde se pinta la imagen.

```
public void mouseMoved(MouseEvent e) {
    Point p = new Point(e.getX(),e.getY());
    if (isInside(p)){
        p = translatePos(p);
        mainFrame.getLabel1().setText("X: " + p.getX() + " Y: " + p.getY());
        printARGB(image.getRGB(p.x,p.y));
    }
    else{
        mainFrame.getLabel1().setText("X: - Y: -");
        mainFrame.getLabel3().setText(" R: - G: - B: -" );
    }
}
```

Figura 17. Evento de movimiento de ratón

En la figura 17 vemos el código del evento para el movimiento de ratón. En el método `isInside` comprobamos que se encuentra dentro de la imagen, pero después debemos de traducir esas coordenadas, porque las coordenadas de la imagen y las del *canvas* donde se dibuja no coinciden. Por último si el punto que obtenemos del ratón no está dentro de la imagen, la barra de estado mostrara un “–“ en cada una de sus salidas.

3.4.2. Barra de Herramientas

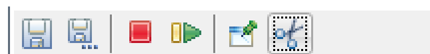

Figura 18. Barra de herramientas del debugger

Se añade una barra de herramientas para aglutinar el resto de funciones de que dispondrá el *debugger*. Pasaremos a hablar por separado de cada una de ellas.

3.4.2.1. Save

Cuando estamos depurando con imágenes, es interesante tener la posibilidad de guardar la imagen en un determinado punto del algoritmo para más adelante hacer comparaciones con una imagen X iteraciones después.

La función guardar nos crea un directorio predeterminado, donde se nos guardará la imagen actual que estemos visualizando en el depurador (Figura 19).

```
if (event.getSource() == buttonSave){
    String SistemaOperativo = System.getProperty("os.name");
    File folder;
    if (SistemaOperativo.contains("Win")){
        folder = new File("C:/ImageDebugger");
        folder.mkdirs();
    }
    else{
        folder = new File("\\ImageDebugger");
        folder.mkdirs();
    }
    String nameImage = ("Image" + numImage + ".jpg");
    numImage++;
    File image = new File(folder.getPath() + nameImage);
    try {
        ImageIO.write(canvas.getImage(), "jpg", image);
    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

Figura 19. Evento botón de guardado

Está preparado para detectar el sistema operativo en el que se está ejecutando. Mediante `System.getProperty (“os.name”)` obtenemos el nombre del sistema operativo, luego de comprobar si es Windows buscando la cadena de caracteres “Win” se crean un directorio de carpetas y dentro ir almacenando las imágenes, con el nombre Image seguido de un número que se va incrementando en 1 para que las siguientes imágenes no se sobrescriban y una extensión `“.jpg”.`

3.4.2.2. Save As

Es básicamente la misma que el apartado anterior, pero ésta nos permite seleccionar mediante una ventana dónde queremos guardar la imagen.

Para ello hacemos uso de la clase `JFileChooser`, que nos proporciona Java, que de manera casi automática nos abre un explorador de archivos para elegir el lugar donde queremos guardar la imagen que tengamos en el depurador actualmente.

3.4.2.3. Stop

Es el mecanismo para salir de la ejecución del *debbuger* y parar al mismo tiempo la ejecución del algoritmo. El *debugger* solo se puede parar desde este botón, en el cual el depurador se pararía sin previo aviso, o cerrando la ventana del *frame*, aunque esta nos pedirá confirmación de que se va a cerrar el programa (Fig. 20).

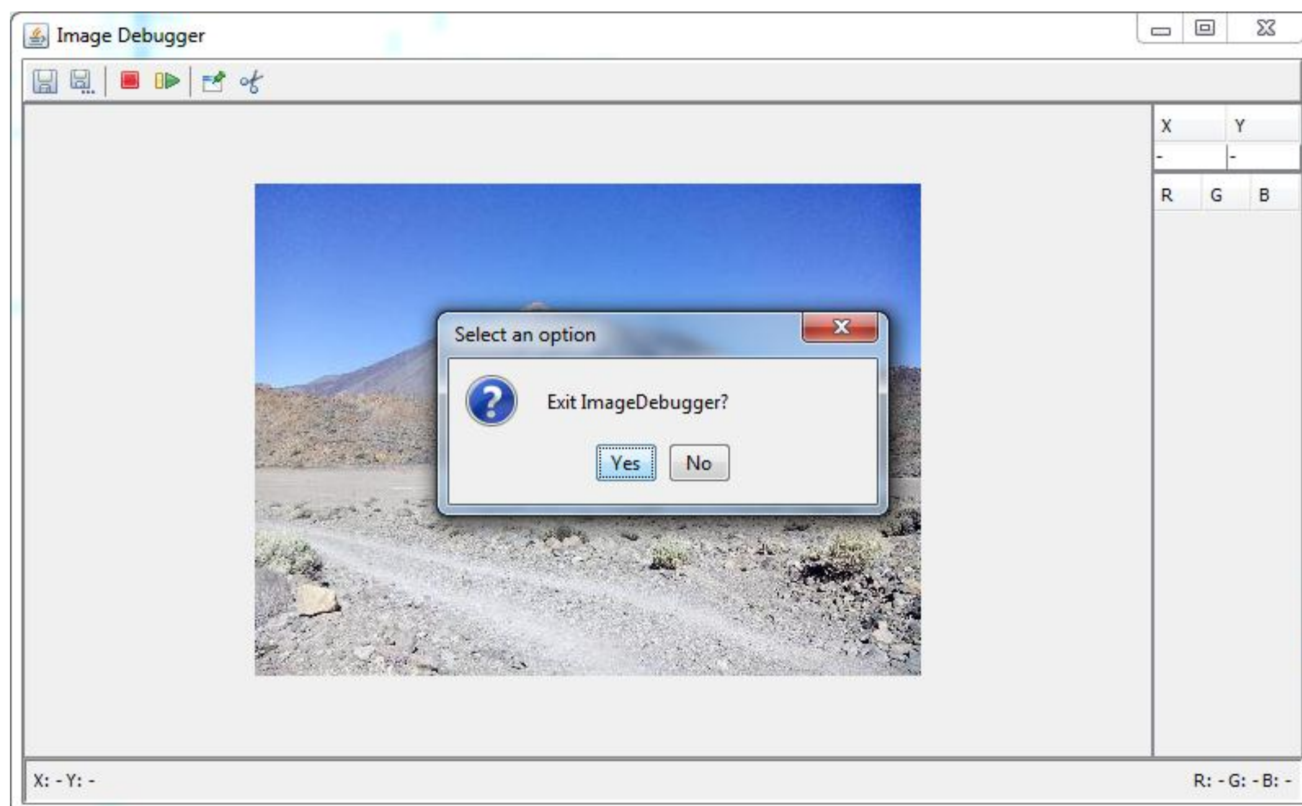

Figura 20. Confirmación salida Debugger

En ambos casos, el depurador se detiene a la vez que el programa que estamos depurando.

3.4.2.4. Resume

Se puede decir que es la función principal del proyecto ya que ésta es la encargada de notificar al hilo del programa que estamos depurando que éste puede continuar.

Para el uso de los hilos es necesario que los métodos donde se hacen uso de las llamadas a hilos, para parar la ejecución `thread.wait` y para notificar a los hilos que pueden continuar con su ejecución `thread.notify`, estén sincronizados. Por esto es que es necesario que la clase que maneja la interfaz `ImageDebug`, obtenga de la clase

`Debug`, el *thread* correspondiente al programa que estamos depurando para poder hacer la llamada al método `synchronized (mainThread).`

Resumiendo, ésta es la función que nos permite avanzar hasta detectar el siguiente punto de ruptura establecido; en ese punto el programa se vuelve a detener y muestra la interfaz con la imagen como se encuentra en el estado actual. El programa que estamos depurando no volverá a avanzar hasta que el botón “Resume” vuelva a ser pulsado.

3.4.2.5. Seguimiento de un pixel

Se trata de una funcionalidad que nos permite ver la evolución de un pixel para cada punto de ruptura. Para ello usamos una tabla, que nos indica las coordenadas del pixel que estamos siguiendo y cada fila es un punto de ruptura que pasa la imagen, mostrándonos las componentes de ese pixel (Fig. 21).

Así mismo esta función nos permite elegir el pixel en cualquier momento. Si las coordenadas del pixel cambian, la tabla se reinicia y comienza a mostrar valores para el nuevo pixel en seguimiento.

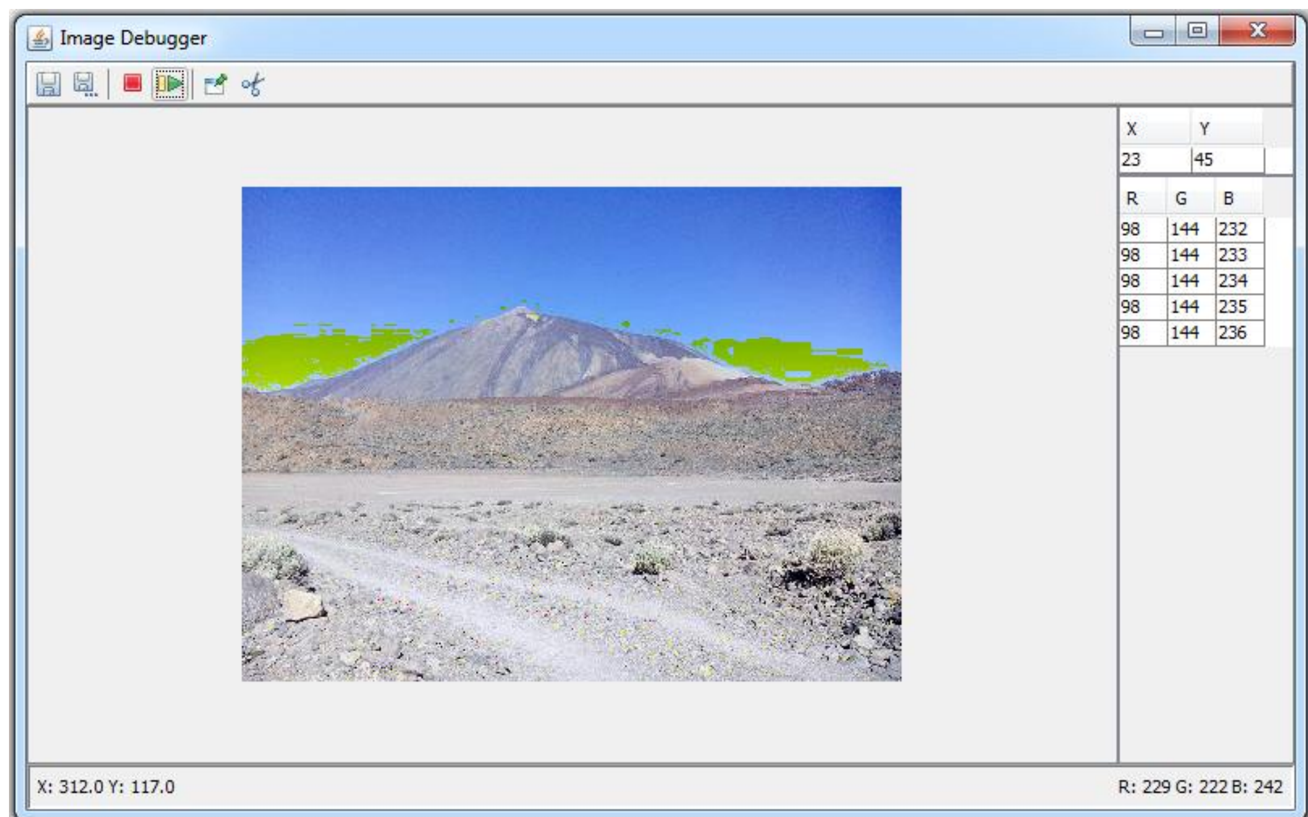

Figura 21. Seguimiento de un pixel

Varias consideraciones: en un principio se deseaba hacer esta función con eventos de ratón pulsando directamente sobre un punto de la imagen, pero creemos que sería poco práctico ya que seleccionar un pixel exacto sería complicado, por tanto se opta por poner un formulario. Cuando pulsamos el botón se nos muestra un formulario donde debemos introducir las coordenadas del pixel del que queremos hacer el seguimiento. Si introducimos unas coordenadas fuera de rango (el rango nos lo muestra en el mismo formulario) nos saltará un error, permitiéndonos volver a introducir las coordenadas en rango.

3.4.2.6. Recortar imagen

Puede que en algún momento una imagen sea demasiado grande, y queramos hacer un seguimiento de una zona específica de la misma más limitada; en ese caso deberemos recortar la imagen.

Esta función se activa mediante un botón *toggle*, que si está activado nos permite cortar la imagen mediante eventos de ratón, de la siguiente manera: pulsamos en donde queramos empezar a cortar y manteniendo pulsado arrastramos hasta donde queramos. Al soltar el ratón, si esta selección está dentro de los límites de la imagen ésta se reducirá a ese tamaño y a partir de este momento en futuros puntos de ruptura solo se dibujará el trozo que nosotros hayamos seleccionado.

Para mantener la imagen recortada deberemos tener un booleano que nos permita indicarle al *canvas*, que la imagen ha sido cortada, y unas coordenadas de la imagen cortada como atributo privado de la misma clase, para que al actualizar la imagen en un punto de ruptura, ésta pinte el trozo que nosotros hemos recortado.

Capítulo 4

4. La librería

En este capítulo trataremos la manera en que nuestro código pasa a exportarse como una librería y cómo cargarla en un proyecto independiente para hacer uso del depurador.

4.1. Creación de la librería

Llegados a este punto tenemos unas clases que nos proporcionan un depurador para algoritmos de imágenes, pero debemos crear una librería para que pueda ser usada en cualquier programa.

El proceso para crear una librería es sencillo. Usamos la opción exportar nuestro código como un archivo ".jar" (fig. 22) y guardarlo. De esta manera ya tenemos la librería preparada para cargarla en cualquier programa en que la necesitemos.

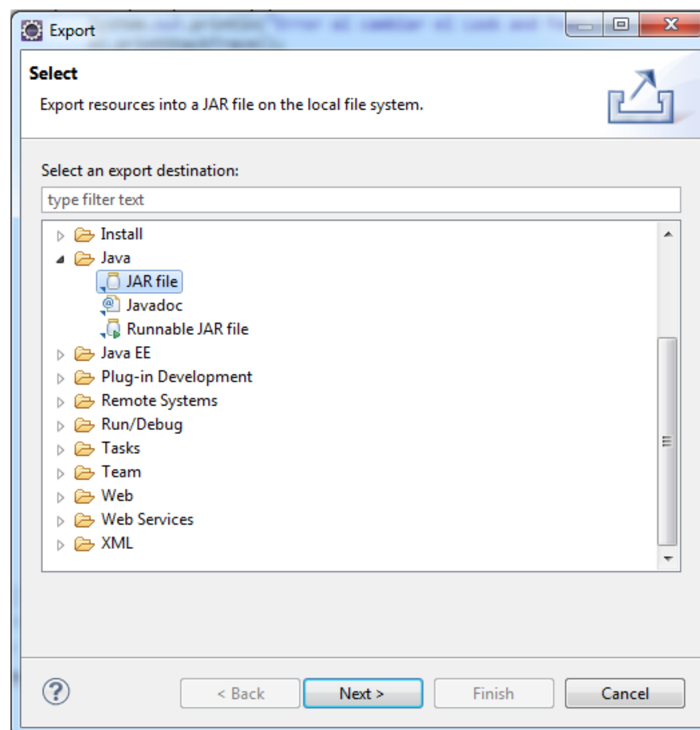

Figura 22. Exportando JAR file

4.2. Cargar la librería

Para usar la librería solo debemos cargarla en el programa en el que sea necesaria, de la siguiente manera, en la carpeta de nuestro proyecto en Eclipse accedemos a la opción "*Add External Archives…*" que se muestra en la figura 23.

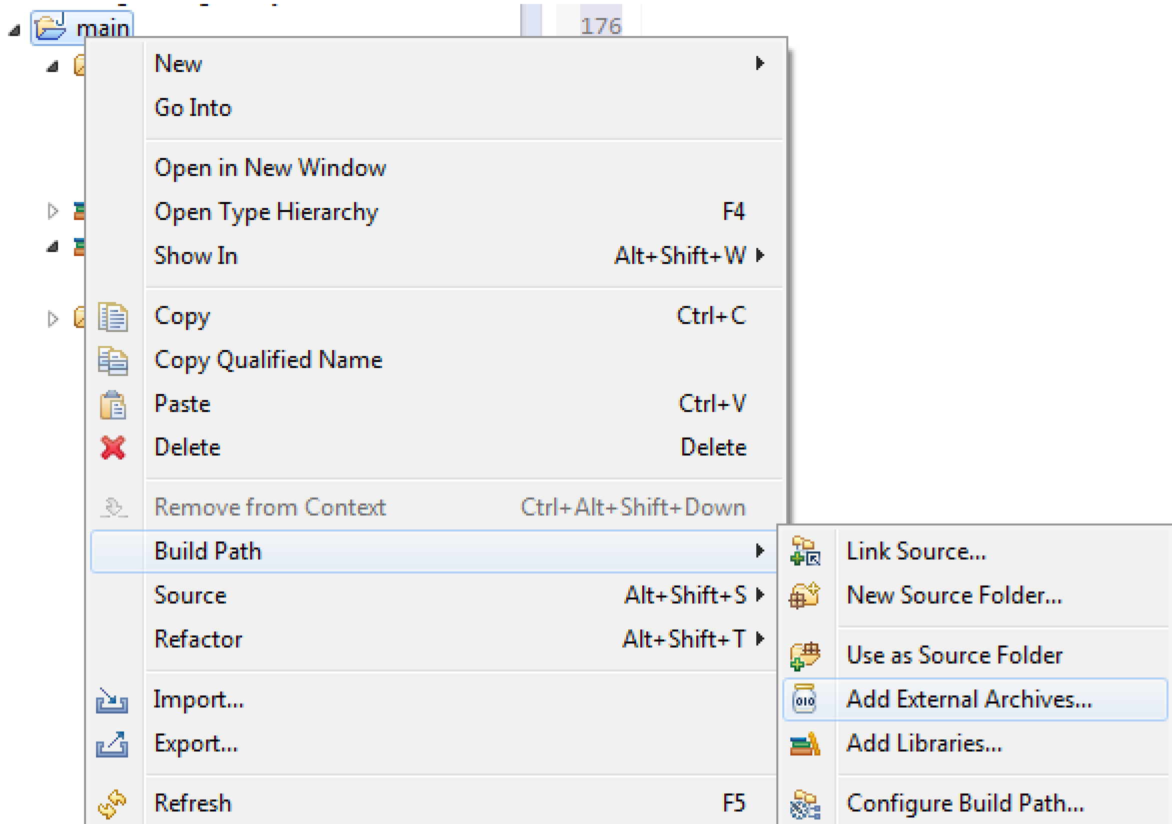

Figura 23. Añadiendo Librería al proyecto

Una vez cargada correctamente debe aparecernos en las carpetas de nuestro proyecto en el apartado "*Referenced Libraries*" (Fig. 24). Después de esto podemos hacer uso de los métodos de la misma.

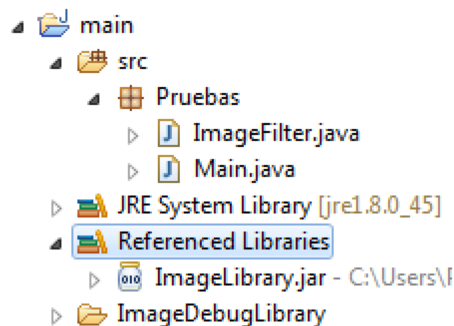

Figura 24. Ejemplo de proyecto que incorpora la librería en Eclipse

4.3. Usando la librería API

La librería una vez cargada en nuestro proyecto dispone solo de dos métodos. Los demás procedimientos de depuración se llevan a cabo mediante la interfaz gráfica de la que ya hemos hablado anteriormente.

Los métodos son los siguientes:

- `Debug (BufferedImage)`: Método encargado de inicializar el depurador

- `breakPoint (BufferedImage):` Método usado para establecer un punto de ruptura.

Una vez que establezcamos el primer punto de ruptura y se muestre el depurador, podemos ir ejecutando el programa mediante el botón resume que va ejecutando el código hasta encontrar la siguiente llamada `breakPoint`.

4.4. Librería independiente del entorno

Una de las ventajas de tener una librería independiente, es exactamente eso, que es independiente del entorno de trabajo, comparado con por ejemplo la otra posibilidad que barajamos en el estudio de viabilidad, el *plugin* para Eclipse.

Esto nos permite hacer uso de la librería fuera de entorno gráficos de desarrollo, por ejemplo en fichero de Linux, que estemos programando en consola, o multitud de entornos, en los que este depurador será de gran ayuda.

Capítulo 5

5. Conclusiones y líneas futuras.

5.1. Conclusiones

En este trabajo hemos desarrollado una librería que permite la depuración de aplicaciones en Java que realicen algún tipo de procesamiento de imágenes.

Desde un punto de vista técnico, el estudio de viabilidad llevado a cabo ha permitido descartar el desarrollo de un *plugin* para Eclipse. Esto se debe a que las características de las interfaces de Java involucradas en el proceso de depuración daban como resultado una ejecución extremadamente lenta para visualizar una imagen.

Si bien la integración con Eclipse parecía una solución más práctica, lo cierto es que la herramienta que hemos implementado finalmente no depende de un entorno de desarrollo concreto, por lo que puede ser integrada en cualquier aplicación programada en Java de una manera muy sencilla.

En conclusión, podemos decir que se ha alcanzado con éxito el objetivo propuesto y se han completado todas las tareas previstas en el proyecto del Trabajo de Fin de Grado.

A nivel personal y profesional, este proyecto me ha permitido adquirir un mayor conocimiento del lenguaje Java, así como explorar los entresijos de la programación de *plugins* para Eclipse. De esta experiencia, se han obtenido las siguientes conclusiones:

- Primeramente, se tiene que tener en cuenta que una buena planificación es fundamental a la hora de llevar a cabo un proyecto como éste, dado que hay que dejar margen para corregir errores, con el fin de llegar a la fecha estimada.
- Las asignaturas cursadas en la especialización de computación de la carrera han resultado de gran ayuda, ya que mucho de su contenido se ha aplicado en este proyecto y esto ha servido para profundizar más en la temática del mismo.
- Aunque este trabajo se ha realizado de manera individual, se aprecian los beneficios del trabajo en equipo, que puede ser esencial en otros ámbitos. En este caso, la colaboración de los tutores del proyecto es inestimable, puesto que me han transmitido las ventajas de mantener una mente abierta e intentar enfocar las cosas desde otra perspectiva a la hora de resolver problemas.
- Por último, afrontar los retos de programación requeridos en este trabajo, y los problemas que han surgido durante el mismo, así como las tareas de investigación y aprendizaje autodidacta, aportan la madurez necesaria para participar en el desarrollo de un proyecto real.

5.2. Líneas futuras.

Los siguientes pasos a realizar en el desarrollo de nuestra librería deberían incluir una etapa de pruebas, como puede ser comprobar la robustez de la aplicación frente a algoritmos muy largos o analizar los errores que puedan surgir a la hora de ejecutar algoritmos que trabajen con transformaciones de tamaño en las imágenes.

Así mismo, como posibles mejoras, se podrían añadir nuevas funcionalidades, tales como ver múltiples imágenes en tiempo de ejecución, añadir una vista de variables más detallada, incorporar una ejecución línea a línea, etc.

Por otro lado, si fuera posible salvar las dificultades técnicas señaladas en el estudio de viabilidad, sería interesante retomar en un futuro el desarrollo de un *plugin* que se integre en el entorno de depuración de Eclipse.

Capítulo 6

6. Summary and conclusions.

In this work we have developed a library that allows debugging Java applications which perform some image processing.

From a technical perspective, the study of viability carried out has allowed to reject the development of a plugin for Eclipse. This is because the characteristics of Java interfaces involved in the debbuging process gave as result an extremely slow execution to display an image.

While integration with Eclipse seemed a more practical solution, the truth is that we have implemented the tool eventually not depend on a specific development environment, so it can be integrated into any Java programmed application in a very simple way.

In conclusion, we can say that the intended objective has been successfully achieved and all the tasks envisaged in the preliminary project have been completed.

On a personal and professional level, this project has allowed me to gain a better understanding of the Java language, as well as exploring the programming of Eclipse plugins. From this experience, we have obtained the following conclusions:

First, you have to keep in mind that good planning is essential when carrying out a project like this, because you have to leave time to correct errors, in order to reach the estimated date.
The subjects studied in the specialization of computing have been of great help, since much of its content has been implemented in this project and this has served to deepen the theme of it.
Although this work has been done individually, the benefits of teamwork, which can be essential in other areas, are appreciated. In this case, the collaboration of the tutors of the project is invaluable, since they have given me the advantages of keeping an open mind and try to approach things from a different perspective when it comes to solving problems.
Finally, the challenges of programming required in this work, and the problems that have arisen during that period, and the tasks of research and self-learning, provide the maturity to participate in the development of a real project.

6.2. Future work

The next steps to take in developing our library should include a testing phase, such as checking the robustness of the application face lengthy algorithms or analyze the errors that may arise in implementing algorithms which work with transformations of size in images.

Also, as possible improvements could be added new features, such as viewing multiple images at runtime, adding a more detailed view of variables, add an execution line by line, etc.

On the other hand, if it were possible to save the technical difficulties identified in the viability study, it would be interesting to take in the future to develop a plugin that is integrated in the Eclipse debugging environment.

Presupuesto

Para este proyecto, teniendo en cuenta que lo desarrolla un único ingeniero informático y que se ha usado software de código abierto, por lo que no hay que pagar licencias de ningún tipo, tenemos la siguiente tabla de presupuesto.

Tarea	Horas	€/Hora	Total
Estudio inicial del tema	20	15 €	300 €
Estudio de viabilidad	70	15 €	1050 €
Diseño e Implementación	180	15 €	2700 €
Coste Total	**270**		**4050 €**

Bibliografía

Eclipse: IDE usado para el desarrollo del proyecto
https://www.eclipse.org/home/index.php

Java: Librería usada para el desarrollo del proyecto
https://www.java.com/es/

Java API: Interfaz de Programación de Aplicaciones para Java
http://docs.oracle.com/javase/7/docs/api/

ImageJ: Depurador de imágenes para Java
http://imagej.nih.gov/ij/