Construcción de un sistema de caracterización global y local de imágenes digitales mediante extracción de rasgos basados en contenido.

Proyecto Cool Imaging.

L.A. González Jaime
Universidad de Granada
lgonzalezjaime@gmail.com

R.J. Palma Durán
Universidad de Granada
odracirnumira@gmail.com

10 de Julio de 2009

Licencia GNU FDL

Copyright (c) 2009 Luis A. González Jaime, Ricardo Juan Palma Durán. Se otorga permiso para copiar, distribuir y/o modificar este documento bajo los términos de la Licencia de Documentación Libre de GNU, Versión 1.2 o cualquier otra versión posterior publicada por la Free Software Foundation; sin Secciones Invariantes ni Textos de Cubierta Delantera ni Textos de Cubierta Trasera. Una copia de la licencia puede ser obtenida desde la web de la *GNU Free Documentation License*, http://www.gnu.org/copyleft/fdl.html.

Índice general

1. Introducción **11**

2. Planificación **13**
- 2.1. Motivación 13
  - 2.1.1. Objetivos 13
  - 2.1.2. Rendimiento 13
- 2.2. Recursos 14
  - 2.2.1. Personal 14
  - 2.2.2. Hardware 14
  - 2.2.3. Software 14
    - 2.2.3.1. Herramientas de desarrollo 14
    - 2.2.3.2. Bibliotecas 15
- 2.3. Filosofía de Trabajo 16
- 2.4. Modelo de desarrollo 16

3. Modelado de requisitos **17**
- 3.1. Alcance del proyecto 17
- 3.2. Sistema propuesto 17
  - 3.2.1. Requisitos funcionales 17
    - 3.2.1.1. Caracterización 19
  - 3.2.2. Requisitos no funcionales 21
- 3.3. Modelo funcional mediante casos de uso 21
  - 3.3.1. Identificación de actores 21
  - 3.3.2. Identificación de los casos de uso 21
    - 3.3.2.1. Paquete PaqueteDeImagenes 22
    - 3.3.2.2. Paquete TratamientoDeImagenes 26
    - 3.3.2.3. Paquete CaracterizaciónDeImágenes 42
  - 3.3.3. Diagrama de casos de uso 55
    - 3.3.3.1. TratamientoDeImagenes 55
    - 3.3.3.2. PaqueteDeImagenes 55
    - 3.3.3.3. CaracterizacionDeImagenes 55

4. Análisis **57**
- 4.1. Modelado estático 57
  - 4.1.1. Diccionario de clases 57
    - 4.1.1.1. Imagen 57
    - 4.1.1.2. ROI 59
    - 4.1.1.3. PaqueteImagenes 59
    - 4.1.1.4. MC 59
    - 4.1.1.5. VC 60
    - 4.1.1.6. OperacionTratamiento 60

4.1.1.7. OperacionCaracterizacion 60
4.1.1.8. ParametroOperacion 60
4.1.1.9. CadenaOperacionesTratamiento 60
4.1.1.10. GeneradorVC 61
4.1.1.11. InformeCaracterizacion 61
4.1.1.12. ProcesadorPaquetesImagenes 62
4.1.1.13. ConfiguracionProcesadorPaquetesImagenes 62
4.1.1.14. CaracterizadorPaquetesImagenes 62
4.1.1.15. ConfiguracionCaracterizadorPaquetesImagenes 62
4.2. Arquitectura del sistema 63
4.2.1. Arquitectura basada en *plugins* 63
4.2.2. Patrones de diseño y arquitectura del sistema 65
4.2.2.1. Patrón Modelo-Vista-Controlador 66
4.2.2.2. Patrón *Observer* 69
4.2.2.3. Patrón *Singleton* 70

5. Storyboard **73**
5.1. Barra de menú 74
5.2. Cool bar 76
5.3. Vistas 77
5.3.1. Menú de Tratamiento de Imágenes 77
5.3.2. Panel de Operación de Tratamiento 79
5.3.3. Paquetes de Imágenes 80
5.3.4. Cadenas de Operaciones 83
5.3.5. Menú de Caracterización de Imágenes 85
5.3.6. Panel de Operación de Caracterización 88
5.3.7. Caracterización de Imágenes 88
5.4. Editores 90
5.4.1. Editor Imagen 91
5.4.2. Editor Caracterización 92
5.4.2.1. Asistente de exportación 94

6. Diseño **99**
6.1. Arquitectura de *plugins* de Eclipse RCP 99
6.1.1. *Core Runtime* y *Eclipse UI* 100
6.1.2. Estructura de *plugins* de Cool Imaging 101
6.2. Manipulación básica de imágenes 104
6.2.1. Clase Imagen 104
6.2.1.1. Imagen 104
6.2.2. Representación gráfica de imágenes 104
6.2.2.1. DisplayJAIConZoom 104
6.2.2.2. DisplayJAIConGraficos 104
6.2.2.3. PanelImagen 106
6.2.2.4. PanelImagenConGraficos 106
6.3. Control de imágenes 106
6.3.1. ModeloImagen 107
6.3.2. VistaImagen 109
6.3.3. ConjuntoModeloImagen 109
6.3.4. VistaConjuntoModeloImagen 110
6.3.5. ControladorImagenes 110

6.4. Opciones de interactividad de imágenes 110
6.4.1. Opciones de interactividad simples 111
6.4.1.1. AccionAmpliarImagen 112
6.4.1.2. AccionReducirImagen 112
6.4.1.3. AccionReestablecerDimensionesImagen 112
6.4.1.4. AccionEliminarROI 112
6.4.2. Opciones de interactividad complejas 113
6.4.2.1. AccionAjustarImagen 116
6.4.2.2. AccionArrastrarImagen 116
6.4.2.3. AccionExtraerROI 116
6.4.2.4. AccionDefinirROI 116
6.4.2.5. AccionSubstraerROI 116
6.4.2.6. IGestorInteractividad 117
6.4.2.7. GestorInteractividadImagenes 117
6.4.2.8. PanelImagenInteractivo 117
6.5. Punto de extensión *operationApplication* 119
6.5.1. Interfaz *IOperadorAplicacion* 119
6.5.2. Punto de extensión *operationApplication* 122
6.6. Control de operaciones: menú de operaciones 122
6.6.1. Menú en árbol de las operaciones 124
6.6.1.1. NodoArbolOperaciones 125
6.6.1.2. NodoCategoria 125
6.6.1.3. NodoOperacion 125
6.6.1.4. ArbolOperaciones 125
6.6.1.5. MenuArbolOperacionesTratamientoImagenes y MenuArbolOperacionesCaracterizacionImagenes 125
6.7. Control de operaciones: inicialización 127
6.8. Control de operaciones: operaciones de tratamiento 128
6.8.1. ControladorOperadorTratamientoImagenes 128
6.8.2. PanelOperadorTratamientoImagenes 132
6.9. Control de operaciones: operaciones de caracterización 133
6.9.1. ControladorOperadorCaracterizacionImagenes 133
6.9.2. PanelOperadorCaracterizacionImagenes 137
6.10. Clase Configurador 141
6.10.1. Carga de las operaciones *IOperadorAplicacion* 141
6.10.2. Carga de las nuevas operaciones de JAI 142
6.10.3. Construcción de los árboles de operaciones 142
6.10.4. Comprobación de las bibliotecas JAI y MediaLib 142
6.10.5. Inicialización del *look and feel* de AWT-Swing 143
6.10.6. Lectura y escritura de variables de preferencia 143
6.11. Gestión de paquetes de imágenes 143
6.11.1. PaqueteImagenes 144
6.11.2. VistaPaqueteImagenes 144
6.11.3. ConjuntoPaqueteImagenes 147
6.11.4. VistaConjuntoPaqueteImagenes 147
6.11.5. GestorPaquetesImagenes 147
6.11.6. GestorVisualPaquetesImagenesAplicacion 147
6.11.7. VistaPaquetesImagenes 147
6.12. Cadenas de operaciones al detalle 148
6.12.1. Lógica de procesamiento de paquetes mediante cadenas de operaciones . 148

6.12.2. Gestión visual de las cadenas de operaciones . . . 149
6.13. Caracterización al detalle . . . 151
6.13.1. Informe de caracterización . . . 151
6.13.2. Lógica de creación de informes de caracterización . . . 155
6.13.3. Gestión visual de la caracterización de paquetes . . . 155
6.14. Lector XML del menú de las operaciones . . . 159
6.14.1. XML y DTD . . . 159
6.14.2. DTD elegido y estructura XML . . . 159
6.14.2.1. DTD . . . 159
6.14.2.2. Estructura XML . . . 160
6.14.3. Lector XML . . . 163
6.14.3.1. API XML elegida . . . 163
6.14.3.2. Diagrama de clases . . . 163
6.14.3.3. XMLTAG . . . 163
6.14.3.4. ArbolOperacionesHandler . . . 163
6.14.3.5. LectorArbolOperaciones . . . 163
6.14.3.6. LectorCarpetaMenu . . . 163
6.15. Internacionalización . . . 165
6.16. Barra de estado . . . 165

7. Pruebas de rendimiento 167
7.1. Paquetes de prueba . . . 167

8. Conclusiones 175

Índice de figuras

3.1. Diagrama de casos de uso del paquete TratamientoDeImagenes . . . 55
3.2. Diagrama de casos de uso del paquete PaqueteDeImagenes . . . 56
3.3. Diagrama de casos de uso del paquete CaracterizacionDeImagenes . . . 56

4.1. Diagrama de análisis del modelo de datos . . . 58
4.2. Un *plugin* . . . 65
4.3. Esquema de conexión básico entre dos *plugins* . . . 66
4.4. Arquitectura de un sistema basado en *plugins* . . . 67
4.5. Patrón Modelo-Vista-Controlador . . . 69
4.6. Patrón *Observer* . . . 70
4.7. Diagrama de clases del patrón *Singleton* . . . 71

5.1. Aspecto general de la aplicación . . . 75
5.2. Vista Menú de Tratamiento de Imagenes . . . 77
5.3. PanelOperacionTratamiento . . . 78
5.4. TabularPanelOperacionTratamiento . . . 78
5.5. TablaSeleccionImagenes . . . 78
5.6. PanelListadoCadenasOperacionesTratamiento (izquierda), y su menú contextual (derecha) . . . 79
5.7. PanelCadenaOperacionesTratamiento (izquierda), y su menú contextual (derecha) 79
5.8. Vista Paquetes de Imagenes . . . 80
5.9. PanelPaqueteImagenes (izquierda), y su menú contextual (derecha) . . . 81
5.10. PanelThumbnails (izquierda), y su menú contextual (derecha) . . . 81
5.11. PanelListadoPaqueteImagenes (izquierda), y su menú contextual (derecha) . . . 82
5.12. Vista Cadenas de Operaciones . . . 83
5.13. PanelConfiguracionProcesamientoPaquetes . . . 84
5.14. Vista Menú de Caracterización de Imagenes . . . 86
5.15. PanelOperacionCaracterizacion . . . 86
5.16. TabularPanelOperacionCaracterizacion . . . 87
5.17. PanelListadoGeneradoresVC (izquierda), y su menú contextual (derecha) . . . 87
5.18. PanelGeneradorVC (izquieda), y su menú contextual (derecha) . . . 87
5.19. Vista Caracterización de Imágenes en modo de procesamiento de paquetes de imágenes . . . 88
5.20. Vista Caracterización de Imágenes en modo de procesamiento de imágenes . . . 89
5.21. PanelConfiguracionCaracterizacion . . . 90
5.22. TablaEditorCaracterizacion . . . 93
5.23. Editor Caracterización . . . 94
5.24. Página primera del asistente de exportación . . . 95
5.25. Página segunda del asistente de exportación . . . 96
5.26. Última página del asistente de exportación . . . 96

6.1. *Plugin coolimaging* 102
6.2. Arquitectura de Cool Imaging 103
6.3. Diagrama de clases del módulo de manipulación básica de imágenes 105
6.4. Diagrama de clases del control de imágenes 108
6.5. Diagrama de clases de las opciones de interactividad que no requieren interacción del usuario con el panel gráfico de la imagen 113
6.6. Diagrama de clases de las opciones de interactividad que sí requieren interacción del usuario con el panel gráfico de la imagen 118
6.7. Diagrama de secuencia del flujo habitual de una operación en Cool Imaging . . . 123
6.8. Diagrama de clases del menú en árbol de las operaciones 126
6.9. Diagrama de secuencia de la inicialización de una operación de tratamiento de imágenes 129
6.10. Diagrama de clases de la gestión de las operaciones de tratamiento de imágenes 130
6.11. Diagrama de colaboración del uso directo de una operación de tratamiento de imágenes 134
6.12. Diagrama de secuencia de la inserción de una operación de tratamiento de imágenes en varias cadenas de operaciones 135
6.13. Diagrama de clases de la gestión de las operaciones de caracterización de imágenes 136
6.14. Diagrama de colaboración del uso directo de una operación de caracterización de imágenes 139
6.15. Diagrama de secuencia de la inserción de una operación de caracterización de imágenes en varios generadores de vector de caracterización 140
6.16. Clases básicas para la visualización de paquetes de imágenes 145
6.17. Vista de gestión de paquetes de imágenes 146
6.18. Diagrama de clases de la lógica de procesamiento de paquetes de imágenes mediante cadenas de operaciones 150
6.19. Diagrama de clases de la vista de cadenas de operaciones 152
6.20. Diagrama de clases del control de los informes de caracterización de la aplicación 154
6.21. Diagrama de clases de la lógica de caracterización de imágenes 156
6.22. Diagrama de clases de la vista de caracterización 158
6.23. DTD: procesamiento de imágenes 160
6.24. DTD: caracterización de imágenes 160
6.25. Ejemplo de menu XML de procesamiento de imágenes 161
6.26. Lineas al comienzo del documento XML 161
6.27. Lineas al comienzo del documento XML 162
6.28. Ejemplo de menu XML de caracterización de imágenes 162
6.29. Diagrama de clases del Lector XML 164
6.30. Diagrama de clases de la gestión de la barra de estado 166

8.1. *Product Backlog* (número de tareas) del proceso de desarrollo 177

Índice de tablas

2.1. Características de los ordenadores personales 14

3.1. Caso de uso *Crear paquete de imágenes* 23
3.2. Caso de uso *Guardar paquete de imágenes* 24
3.3. Caso de uso *Abrir paquetes de imágenes* 25
3.4. Caso de uso *Añadir imágenes a paquete de imágenes* 25
3.5. Caso de uso *Abrir imagen* 27
3.6. Caso de uso *Abrir imagen con ROI* 28
3.7. Caso de uso *Cerrar imagen* 29
3.8. Caso de uso *Guardar imagen* 30
3.9. Caso de uso *Ampliar imagen* 31
3.10. Caso de uso *Reducir imagen* 32
3.11. Caso de uso *Escalar área* 33
3.12. Caso de uso *Extraer ROI* 34
3.13. Caso de uso *Definir ROI* 34
3.14. Caso de uso *Aplicar operación de tratamiento* 35
3.15. Caso de uso *Crear cadena de operaciones* 36
3.16. Caso de uso *Guardar cadena de operaciones* 37
3.17. Caso de uso *Abrir cadenas de operaciones* 38
3.18. Caso de uso *Insertar operación en cadenas* 39
3.19. Caso de uso *Procesar paquetes de imágenes* 40
3.20. Caso de uso *Configurar procesamiento de paquetes* 41
3.21. Caso de uso *Aplicar operación de caracterización* 43
3.22. Caso de uso *Abrir informe de caracterización* 44
3.23. Caso de uso *Cerrar informe de caracterización* 45
3.24. Caso de uso *Guardar informe de caracterización* 46
3.25. Caso de uso *Crear generador de vector de caracterización* 47
3.26. Caso de uso *Guardar generador de vector de caracterización* 48
3.27. Caso de uso *Abrir generadores de vector de caracterización* 49
3.28. Caso de uso *Insertar operación en generadores de vector de caracterización* . . . 50
3.29. Caso de uso *Caracterizar paquetes de imágenes* 51
3.30. Caso de uso *Caracterizar imágenes* 52
3.31. Caso de uso *Configurar caracterización de imágenes* 53
3.32. Caso de uso *Cerrar aplicación* 54

7.1. Características de los ordenadores de pruebas 168
7.2. Pruebas de rendimiento en el Portátil 1 bajo Windows Vista. 170
7.3. Pruebas de rendimiento en el Portátil 1 bajo Ubuntu 9.04. 171
7.4. Pruebas de rendimiento en el Portátil 2 bajo Windows XP. 172
7.5. Pruebas de rendimiento en el Portátil 2 bajo Ubuntu 9.04. 173

Capítulo 1

Introducción

Igual que los ordenadores e Internet pasaron a ser algo cotidiano, las imágenes también han pasado a formar parte de nuestra vida diaria: las cámaras digitales o dispositivos similares son habituales hoy día, habiendo impulsado, así pues, una amplia gama de ámbitos donde el usuario final ha podido beneficiarse del tratamiento digital de imágenes. Aplicaciones de reconocimiento facial, reconocimiento de matrículas, seguridad, etc., son claros ejemplos de la repercusión que el tratamiento digital de imágenes ha conseguido en nuestra sociedad. El tratamiento digital de imágenes es un campo abierto, donde ciencia y tecnología interactúan para ofrecernos día tras día nuevas herramientas, con aplicaciones que se extienden desde el mero ocio hasta la más pura investigación científica.

Dentro de este panorama, nuestro objetivo es el de contribuir al desarrollo científico, mediante un proyecto de caracterización de imágenes mediante la extracción de rasgos basados en contenido, donde se puedan realizar estudios globales y locales. La idea inicial tuvo su origen en la posibilidad de crear una herramienta de caracterización de imágenes aplicable al campo de la biomedicina, para así ayudar a los profesionales médicos en su labor profesional. Sin embargo, dado que entendemos la caracterización de imágenes como un proceso que debería ser independiente del campo en el que se aplicase, decidimos abstraer el problema, optando por el diseño de una aplicación de carácter no sólo más general, sino también ampliable, que permitiera la incorporación de nuevas técnicas y algoritmos de caracterización.

La caracterización de imágenes tiene como objetivo asignar a una imagen una serie de valores matemáticos (usualmente estadísticos) que permitan describirla de forma objetiva. Mediante estos rasgos se pretende caracterizar, de algún modo, la imagen, para así poder distinguirla de otras.

Un proceso tan simple como éste tiene innumerables aplicaciones prácticas en el mundo real.

Dentro del campo de la biomedicina la caracterización de imágenes puede utilizarse para la detección de patologías en imágenes médicas. El profesional médico, por ejemplo, podría disponer de un banco de imágenes médicas de un cierto órgano con una patología determinada; una parte de las imágenes contendría órganos enfermos, mientras que el resto contendría órganos sanos. Mediante el proceso de caracterización de imágenes se podría describir mediante rasgos matemáticos las imágenes que contuvieran los órganos enfermos. Mediante los rasgos de caracterización extraídos se podría llevar a cabo un posterior proceso de clasificación a través de programas de análisis estadístico. Este proceso de clasificación permitiría distinguir los órganos enfermos de los órganos sanos.

Nuestro propósito es el de construir una herramienta de extracción de rasgos de caracterización de imágenes digitales. Si bien el proceso de caracterización tiene como último objetivo la clasificación de imágenes, nosotros nos quedaremos en el primer paso (y no por ello menos importante): la extracción de medidas de caracterización. Se pretende que estas medidas puedan ser posteriormente utilizadas por paquetes profesionales de análisis estadístico que puedan

llevar a cabo la clasificación final de imágenes.

Este documento incluye toda la documentación generada a lo largo del desarrollo del proyecto: no sólo la parte referente a la ingeniería del software, propia de cualquier proyecto informático, sino también todo lo que, a lo largo del desarrollo del proyecto, hemos considerado suficientemente relevante como para merecer una mención especial.

Queremos dedicar este proyecto a todas aquellas personas que, día tras día, nos han animado a seguir adelante. Destacamos, sobre todo, el apoyo recibido por:

- Joaquín Fernández Valdivia: por ser, simple y llanamente, un buen coordinador de proyecto. Por guiarnos por la senda correcta, y al mismo tiempo, no ponernos piedras en el camino.
- María Civantos Hueso: por diseñar gran parte de los iconos de la aplicación, y por tener que soportar, día a día, una sobredosis de Cool Imaging.
- Raúl Jiménez Ortega: por ayudarnos a dar publicidad al proyecto, así como aplicar interminables baterías de pruebas a la aplicación.
- José Francisco Mantas Serrano: por diseñar el logo de Cool Imaging, el *splash screen*, y la imagen de diálogo *about*.
- Servicio de Traducción Universitario de la Universidad de Granada (STU), por ofrecernos sus servicios y ayudarnos en la traducción de la aplicación al inglés, así como del manual de usuario.

Sin su apoyo, el camino habría sido mucho más duro, y posiblemente no habríamos llegado a donde hemos llegado. A todos ellos, gracias.

Reconocimientos

Cool Imaging ha sido galardonada con varios premios a lo largo de su joven existencia:

- Ganadora del premio Premio Emilio Herrera de la Universidad de Granada (2009).
- Finalista, en la categoría de Innovación, en la 3ª edición del Concurso Universitario de Software Libre Nacional (2009).
- Finalista en la 1ª edición del Concurso Universitario de Software Libre Granadino (2009).

Capítulo 2

Planificación

2.1. Motivación

2.1.1. Objetivos

El objetivo del proyecto a realizar es la construcción de un sistema de extracción de rasgos basados en contenido de imágenes digitales. El sistema debe permitir, en última instancia, caracterizar tanto global como localmente una imagen digital, mediante rasgos matemáticos de utilidad para la propia caracterización. La idea que nos mueve es la de crear una herramienta de propósito general que aglutine una gran diversidad de técnicas de caracterización basadas en contenido, la cual pueda ser usada por todo tipo de especialistas, independientemente del campo en el que se muevan. Al tratarse de una aplicación de tratamiento de imágenes, ésta debe incluir herramientas clásicas de manipulación de imágenes digitales, independientemente de la caracterización que posteriormente se pudiera hacer de éstas.

La caracterización de imágenes es un campo en continuo avance. El avance científico en este área permite el desarrollo de nuevas técnicas de caracterización. Es por ello que el proyecto debe ser desarrollado con la filosofía de *ampliabilidad* en mente: una estructura fácilmente extensible permitirá la incorporación de nuevas técnicas y algoritmos para los que no se haya dado soporte inicialmente.

El sistema debe mostrar una interfaz intuitiva y completa. Una aplicación dirigida a especialistas de todo tipo dentro del ámbito de la caracterización de imágenes debe ser especialmente fácil de usar. El usuario debe tener la posibilidad de experimentar con diferentes tipos de técnicas y algoritmos de caracterización y así obtener resultados cuantitativos con los que caracterizar imágenes. En este sentido, el sistema debe estar enfocado a la obtención de resultados.

Por último, el proyecto va a ser desarrollado con la idea de publicarlo bajo licencia GNU GPL, versión 3 o posterior, motivo por el cual se hará uso, exclusivamente, de componentes compatibles con esta licencia. El propósito es desarrollar un sistema basado en la filosofía del *software libre*, sin duda por nuestra convicción de que es el mejor modo de producir software de calidad.

2.1.2. Rendimiento

El rendimiento de una aplicación guiada por interfaz de usuario es un factor importante dentro del desarrollo del software, pues se pretende dar al usuario una sensación de interactividad ininterrumpida. El que nuestra aplicación trabaje con imágenes digitales (de, posiblemente, gran tamaño) hace surgir un posible problema de rendimiento, dado que ciertos algoritmos de tratamiento de imágenes digitales pueden hacer que el tiempo de respuesta se incremente considerablemente. Es por ello que se deberá tener especial cuidado en el diseño de los algoritmos

y estructuras relacionados con el tratamiento de imágenes digitales, ya que en su mayor parte el rendimiento de la aplicación va a estar condicionado por ellos.

2.2. Recursos

2.2.1. Personal

Pocos son los recursos de los que se dispone para la realización de este proyecto. En lo que respecta al equipo de desarrollo, está compuesto sola y exclusivamente por los autores del proyecto, Luis Antonio González Jaime y Ricardo Juan Palma Durán. Ambos nos dedicaremos al desarrollo del software en su totalidad, pasando por todas las fases necesarias: análisis de requerimientos, diseño, implementación, etc.

2.2.2. Hardware

Los recursos hardware están formados fundamentalmente por dos ordenadores personales, uno por cada integrante del grupo de desarrollo.

Componente	Ordenador 1	Ordenador 2
Marca	Asus	Acer
Microprocesador	Intel Pentium M 1,66 GHz	Intel Core 2 Duo P8400 2,26 GHz
Cache	2 MB L2 de 533 MHz	3 MB L2
Ram	DDR2 SDRAM 1024 MB (533 MHz)	DDR2 SDRAM 4096 MB
Tarjeta Gráfica	NVIDIA GeForce Go 7300	NVIDIA GeForce 9300M GS

Tabla 2.1: Características de los ordenadores personales

En la tabla 2.1, se muestran las características de ambos ordenadores personales.

2.2.3. Software

En lo que a software se refiere, el problema que nos guía es la elección de unas u otras herramientas que sean compatibles con la licencia GNU GPL, versión 1.2 o posterior, basadas en la filosofía del *software libre* y gratuitas.

En esta sección comentaremos todas las posibilidades barajadas y estudiadas, así como las elecciones que se han tomado. Detallaremos herramientas de programación como bibliotecas y otros paquetes software que serán incorporados en este proyecto.

2.2.3.1. Herramientas de desarrollo

Para facilitar el desarrollo del proyecto nos vemos en la necesidad de utilizar un *Entorno de Desarrollo Integrado* (IDE), así como un sistema de edición de documentos.

Partiendo de la intención de desarrollar un software multiplataforma, decidimos usar el lenguaje de programación **Java**.

Hoy en día existen dos IDEs para el desarrollo de aplicaciones *Java* que destacan por su potencia, a saber: *Eclipse* y *NetBeans*. Entre ambos, se decidió utilizar **Eclipse** debido a un mayor conocimiento por parte de los desarrolladores. Además, Eclipse es un entorno de programación más ligero, a diferencia de *NetBeans*, el cual degrada de forma apreciable el rendimiento de la máquina.

Sin embargo, el mayor motivo por el cual se decidió utilizar Eclipse está fuertemente ligado con la filosofía de extensibilidad con la que queríamos dotar a nuestra aplicación: se pensó en realizar un sistema fácilmente extensible, de manera que cualquier persona con unas nociones respetables de programación y desarrollo de software fuera capaz de ampliar su funcionalidad. La tecnología basada en *plugins* de Eclipse era, por sí misma, un perfecto referente, y merecía la pena estudiarlo a fondo. Tras investigar un poco más, se observó que el mismo IDE de Eclipse se basa en un *framework* que está al alcance de cualquiera, y que puede ser utilizado como base de cualquier aplicación. Por ello se pensó en utilizar la filosofía de trabajo de *Eclipse Rich Client Platform (EclipseRCP) [6]*, la cual es la base de Eclipse.

Así pues, aunque *NetBeans* ofrece grandes facilidades, como herramientas de creación de diagramas de clases o de casos de uso, o un entorno eficiente de creación de interfaces gráficas de usuario, la balanza se inclinó definitivamente por Eclipse. Sin embargo, para la creación de diagramas de clases o casos de uso sí se ha decidió utilizar *NetBeans*, por conseguir mejores resultados que los plugins de desarrollo UML conocidos para Eclipse.

Para la edición de la documentación, se planteó la posibilidad de usar *OpenOffice* o LaTeX. Ambas herramientas tienen una filosofía de trabajo totalmente diferente: mientras que *OpenOffice* es una herramienta de tipo WYSIWYG, donde *lo que lo que ves es lo que obtienes*, LaTeX toma la filosofía de un diseñador de libros, donde el autor sólo se preocupa de escribir el contenido del libro, que posteriormente es **compilado**, obteniendo unos resultados absolutamente profesionales.

Se decidió utilizar LaTeX debido a su gran potencialidad a la hora de la edición de textos científicos y matemáticos, así como por la apariencia profesional de los resultados que con él se obtienen.

2.2.3.2. Bibliotecas

Puesto que nuestra aplicación trabaja con imágenes, nos resultaría de ayuda la utilización de alguna biblioteca de tratamiento de imágenes que nos facilitara dicha tarea. Se planteó la utilización de la biblioteca OpenCV [1] , pero dicha biblioteca está orientada al lenguaje de programación C, por lo que se pensó en la integración de ambos lenguajes. Sin embargo, no tardamos en encontrar la biblioteca Java Advanced Imaging (JAI) [2] de Java, para el tratamiento de imágenes.

Al final, se decidió utilizar la biblioteca **JAI** debido a que ésta está orientada a la programación en *Java*, e incluso está optimizada para el procesamiento de imágenes a nivel hardware [1]. En su momento, se pensó que al trabajar sobre la máquina virtual, la eficiencia de JAI se vería afectada, y con ello el rendimiento global de la aplicación, pero las optimizaciones a nivel hardware permitieron que la idea de emplear JAI siguiera adelante.

JAI está optimizada para las plataformas *Sun/Solaris*, *Win32* y *Linux*. No todas las operaciones soportadas por JAI están optimizadas para todas las plataformas, ni todas las plataformas muestran el mismo grado de optimización. Así por ejemplo, las plataformas *Win32* que dan soporte a instrucciones de tipo MMX[2] muestran una aceleración hardware mayor a las plataformas que no dan soporte a dicho tipo de instrucciones. En todo caso, si la operación a realizar no soporta aceleración hardware, JAI hace uso de una implementación en *Java puro*, que conlleva el no usar ningún tipo de optimización.

[1] *A variety of implementations are possible, including highly-optimized implementations that can take advantage of hardware acceleration and the media capabilities of the platform, such as MMX on Intel processors and VIS on UltraSparc.* Seccion 1.3.7 en [3].

[2] Visitar http://en.wikipedia.org/wiki/MMX_(instruction_set) para más información.

2.3. Filosofía de Trabajo

Como se ha explicado en la sección 2.2.3.1, se decidió usar Eclipse, entre otras razones, por la posibilidad de hacer uso de la arquitectura *Eclipse RCP*. El decidir usar Eclipse RCP supone un fuerte impacto, no sólo en la filosofía de trabajo a seguir, sino en el diseño de la misma aplicación, que se verá condicionado a seguir una filosofía de desarrollo mediante *plugins*.

2.4. Modelo de desarrollo

La gestión del proyecto la llevaremos a cabo mediante la metodología ***Scrum*** [3], por su carácter iterativo e incremental. Como se ha comentado con anterioridad, se pretende construir una aplicación fácilmente ampliable, por lo que el proceso de desarrollo a seguir debe seguir una metodología poco rígida y de fácil modificación, como nos posibilita Scrum.

Aunque realmente no es necesario usar un método como Scrum para un equipo formado por dos personas, sí usaremos algunos de sus principios para facilitar la gestión y coordinación del proyecto.

No es la finalidad de este apartado explicar el método Scrum, aunque sí detallaremos algunos de los roles que utilizaremos de Scrum y quién asumirá dichos roles en nuestro proyecto.

- **ScrumMaster**: se asegura de que el proceso Scrum se utiliza como es debido. Es el encargado de hacer que las reglas se cumplan. En nuestro caso, este rol será asumido por ambos desarrolladores.

- **ProductOwner**: representa la voz del cliente. Redacta los requisitos, las prioriza y las coloca en el *Product Backlog*. Este rol será asumido por ambos desarrolladores.

- **Equipo**: tiene la responsabilidad de desarrollar y entregar el producto. Al igual que los roles anteriores será asumido por ambos desarrolladores.

- **Usuarios**: son las personas que usarán nuestro software. Este rol es desempeñado por ambos desarrolladores, así como el tutor del proyecto y otras personas que estén interesadas en utilizarlo, de forma que se tendrán en consideración sus opiniones para el desarrollo.

En nuestro caso concreto, prácticamente todos los roles son asumidos por los propios desarrolladores, como se ha mostrado anteriormente. El papel más importante es el del cliente, puesto que sin él el desarrollo de este software no tendría sentido. *Si un software no se utiliza, ¿ha sido escrito?*.

La duración de cada *sprint* será variable, dependiendo de las circunstancias y obligaciones de los desarrolladores; no obstante, un *sprint* seguirá siendo un periodo corto de tiempo (no superior a un mes) cuyo resultado será la obtención un producto potencialmente entregable.

El *product backlog* albergará un listado con todos los requerimientos definidos, de forma que tendrán que ir concluyéndose en los diferentes *sprints*. El *product backlog* no será invariable, sino que éste aumentará con nuevos requerimientos, o disminuirá eliminando otros obsoletos, acordes con la evolución del producto. En cada *sprint* se desarrollarán algunos de los requerimientos determinados en el *product backlog* según las prioridades que estimen oportunas desarrolladores y clientes.

Para facilitar la tarea de mantenimiento del *product backlog* se ha optado por utilizar la tecnología proporcionada por *Google Bloc de Notas* debido a que cubre todas nuestras exigencias, desde poder categorizar las notas a poder crearlas de forma simple y reordenarlas.

[3] Para más información sobre el método Scrum, visitar http://es.wikipedia.org/wiki/Scrum.

Capítulo 3

Modelado de requisitos

En este capítulo se recogen los requerimientos del software para este proyecto. Estos requerimientos se desprenden de las charlas habidas con el coordinador del proyecto, Joaquín Fernández Valdivia, así como de las ideas propias de los mismos autores.

3.1. Alcance del proyecto

La idea del proyecto es la de servir como herramienta investigadora dentro del tratamiento de imágenes digitales. El sistema debe permitir al usuario tratar imágenes digitales, tanto global como localmente. Aunque la aplicación incorpore de por sí numerosas técnicas de tratamiento de imágenes, como pueden ser algoritmos de filtrado, operadores aritméticos, operadores de contracción y dilatación, etc., una de las ideas fundamentales es la de construir un sistema extensible. Actualmente, una aplicación sin capacidad de ampliarse o actualizarse está condenada al fracaso. Él sistema debe permitir su fácil actualización con la mínima intervención del usuario. Estas actualizaciones deberían permitir ampliar la funcionalidad primaria de la aplicación, a saber, métodos y herramientas tratamiento digital de imágenes.

3.2. Sistema propuesto

3.2.1. Requisitos funcionales

1. El sistema permitirá cargar imágenes almacenadas localmente, y en gran número de formatos de almacenamiento, incluyendo:

    i. BMP
    ii. GIF.
    iii. JPG.
    iv. PNG.
    v. PBM (PNM).
    vi. PGM (PNM).
    vii. PPM (PNM).
    viii. TIFF.
    ix. FPX.

2. El sistema permitirá aplicar operaciones propias del tratamiento digital de imágenes a una o varias imágenes previamente seleccionadas. Estas operaciones incuyen:

   i. Filtrado lineal mediante kernels de convolución.
   ii. Filtrado de mediana.
   iii. Superposición de imágenes.
   iv. Composición de imágenes.
   v. Operaciones de tipo artimético: suma de imágenes; diferencia de imágenes; multiplicación de imágenes; exponenciación de imágenes, etc.
   vi. Operaciones de tipo lógico: and; or; not; xor.
   vii. Operaciones geométricas: rotación; escalado; inclinación (*shearing*); translación; transformación afin genérica.

3. El sistema permitirá guardar, en almacenamiento local, imágenes ya abiertas por el usuario u otras obtenidas como resultado de aplicar operaciones o modificaciones sobre otras. El sistema debe contemplar una gran variedad de métodos de almacenamiento, incluyendo:

   i. BMP
   ii. JPG.
   iii. PNG.
   iv. PBM (PNM).
   v. PGM (PNM).
   vi. PPM (PNM).
   vii. TIFF.

4. El sistema debe permitir cerrar una imagen ya abierta o derivada de otra(s).

5. El sistema debe permitir *hacer zoom* de la imagen, ya sea seleccionando un área a ampliar, o bien indicando que simplemente desea ampliar la imagen. Análogamente, el sistema debe permitir hacer zoom inverso, es decir, reducir de tamaño la imagen.

6. El sistema debe permitir extraer una región de interés (ROI) de una imagen seleccionada. El sistema debe ser especialmente eficaz y flexible en lo referente a la extracción de una región de interés.

7. El sistema debe permitir definir una ROI sobre una imagen seleccionada.

8. El sistema debe ser capaz de definir *paquetes de imágenes*. Un *paquete de imágenes* es un conjunto de imágenes agrupadas bajo un nombre. Los paquetes de imágenes deben poder ser almacenados en disco, así como leídos, para poder posteriormente cargarlos en la aplicación.

9. El sistema debe permitir definir *cadenas de operaciones*. Una *cadena de operaciones* es una secuencia ordenada de operaciones unarias (es decir, que toman como entrada una única imagen y que devuelve como resultado una única imagen) agrupadas bajo un mismo nombre. Las cadenas de operaciones deben poder ser almacenadas en disco, así como leídas, para poder cargarlas posteriormente en la aplicación.

10. El sistema debe permitir procesar un paquete de imágenes mediante una cadena de operaciones: cada imagen del paquete es procesada por la cadena de operaciones, obteniéndose una imagen resultado por cada imagen del paquete. Cada imagen resultado consiste en la respectiva imagen de entrada a la que se le han aplicado, en orden, las operaciones de la cadena de operaciones.

11. El sistema debe ofrecer información de utilidad de cada imagen: tamaño, tipo de espacio de color, número de bandas, etc.

12. El sistema debe permitir caracterizar imágenes. Caracterizar una imagen significa obtener uno o varios *vectores de caracterización*, los cuales contienen datos que caracterizan la imagen. Dichos vectores de caracterización son creados a partir de operaciones de caracterización. Cada operación de caracterización obtiene datos de caracterización a partir de la imagen, y dichos datos son usados para la creación de vectores de caracterización.

13. El sistema debe permitir definir *generadores de vector de caracterización*. Un *generador de vector de caracterización* es un conjunto de operaciones de caracterización agrupadas bajo un mismo nombre. Estos generadores son usados para la obtención de los vectores de caracterización de las imágenes. Los generadores de vector de caracterización deben poder ser almacenados en disco, así como leídos, para poder cargarlos posteriormente en la aplicación.

14. El sistema debe ser capaz de caracterizar paquetes de imágenes, y crear informes que condensen los datos de caracterización de todas las imágenes de los paquetes. Se debe permitir obtener medidas de resumen de los datos de caracterización de las imágenes caracterizadas. Estos informes deben poder ser almacenados en disco, así como leídos, para poder cargarlos posteriormente en la aplicación.

15. El sistema debe permitir exportar los datos de caracterización de imágenes a formatos que permitan un postprocesamiento. Se proponen los siguientes formatos:

    - Hoja de cálculo ODF (formato .ods).
    - Microsoft Excel 97/2000/XP (formato .xls).
    - Microsoft Excel 2007 XML (formato .xlsx).
    - Documento de texto (formato .dat).

16. El sistema debe permitir al usuario tener un conocimiento explícito de qué operaciones se están llevando a cabo en cada momento, y, en caso de que sea posible, cancelar las operaciones seleccionadas.

3.2.1.1. Caracterización

Caracterizar una imagen significa obtener uno o varios *vectores de caracterización* a partir de dicha imagen. Un *vector de caracterización* es una estructura de tipo vectorial, la cual contiene datos de caracterización numéricos obtenidos de la imagen. Un ejemplo de vector de caracterización sería $\{32{,}4; 98{,}4; 99{,}0\}$. Este vector de caracterización contiene tres datos de caracterización, cada uno obtenido de una cierta imagen de entrada, a través de la aplicación de una cierta operación de caracterización.

Los vectores de caracterización y las operaciones de caracterización están íntimamente relacionadas. Una operación de caracterización obtiene una medida numérica a partir de una imagen. Un vector de caracterización aglutina, en una estructura vectorial, las medidas numéricas obtenidas por varias operaciones de caracterización. Cada una de las medidas numéricas obtenidas por una operación de caracterización se llama *medida de caracterización*.

Supongamos que tenemos tres operaciones de caracterización. La primera de ellas obtiene la media de cada una de las bandas de la imagen. La segunda obtiene la desviación típica de todos los píxeles de todas las bandas de la imagen. La tercera obtiene la medida de entropía, sobre el histograma, de cada una de bandas de la imagen. Supongamos que la imagen de entrada tiene tres bandas (una de rojo, otra de verde y otra de azul). Los resultados (medidas de caracterización) de las tres operaciones podrían ser los siguientes:

- Media: $\{11{,}3; 45{,}6; 210{,}8\}$.
- Desviación típica: 10,5.
- Entropía: $\{6{,}3; 7{,}21; 6{,}01\}$.

En este caso se podría crear un vector de caracterización, a partir de dichas medidas de caracterización, como el siguiente:

$$\{\{11{,}3; 45{,}6; 210{,}8\}; 10{,}5; \{6{,}3; 7{,}21; 6{,}01\}\}$$

Un vector de caracterización debe ser generado a partir de cierta estructura que agrupe un repertorio de operaciones de caracterización. Para ello se considera la estructura *generador de vector de caracterización*. Un *generador de vector de caracterización*, simplemente, agrupa una serie de operaciones de caracterización, de modo que, cuando es aplicado a una imagen, se obtiene un vector de caracterización que contiene los datos generados por las operaciones de caracterización del generador.

Un vector de caracterización es una representación condensada de una imagen. El propósito final de la caracterización de imágenes es el de poder obtener medidas numéricas que, de algún modo, resuman (caractericen) una imagen. Un vector de caracterización, en última instancia, almacena un repertorio de medidas numéricas calculadas sobre una imagen. Cada una de dichas medidas numéricas (medidas de caracterización) representa una propiedad de la imagen. Así pues, el vector de caracterización representa, de algún modo, a la imagen de la que se obtuvo. Si bien un vector no equivale a la imagen original, es evidente que el vector contiene información a partir de la cual se pueden inferir características de la imagen inicial. Supongamos que, por ejemplo, trabajamos sobre una imagen de niveles de gris, y que disponemos de un vector de caracterización obtenido a partir de dicha imagen. Si el vector contiene la media de los píxeles de la imagen, que resulta ser, por ejemplo, 137, así como la desviación típica de los píxeles de la imagen, que resulta ser 0, se puede inferir que la imagen original es una imagen en la que todos los píxeles tienen un valor de 137. Éste, por supuesto, es un ejemplo bastante sencillo, pero la idea que quiere transmitir está clara: un vector de caracterización representa, de forma más o menos precisa, la imagen de la que es obtenido.

Uno de los aspectos en los que se quiere hacer hincapié es en el la caracterización de paquetes de imágenes. Al caracterizar paquetes de imágenes, obviamente, se obtendrá como resultado un gran número de vectores de caracterización, ya que a cada imagen de los paquetes se le calcularán uno o varios vectores de caracterización.

Surge así la necesidad de crear una estructura que sea capaz de visualizar los datos de caracterización tanto de imágenes independientes como de paquetes de imágenes. Dicha estructura es lo que se conoce como *informe de caracterización*.

Un informe de caracterización debe permitir observar, de forma estructurada, los vectores de caracterización de todas las imágenes procesadas. Si el procesamiento se ha llevado a cabo sobre paquetes de imágenes, el informe debe ser capaz de mostrar cada uno de los paquetes, de forma estructurada, así como cada una de las imágenes de cada paquete, y sus respectivos vectores de caracterización. Si el procesamiento se ha llevado sobre una o varias imágenes de forma independiente (no asociadas a paquetes de imágenes), el informe debe ser capaz de visualizar los vectores de caracterización calculados sobre dichas imágenes. Cuando hablamos de caracterizar imágenes de forma independiente, nos referimos al hecho de que la aplicación debe permitir caracterizar imágenes aisladas, aparte de paquetes de imágenes (es decir, imágenes que no estén asociadas a ningún paquete de imágenes).

El informe de caracterización debe ser capaz, además, de mostrar las medidas de resumen que pueden calcularse sobre vectores de caracterización. Recuérdese que una medida de resumen sobre varios vectores de caracterización es un único vector de caracterización que se ha obtenido

a partir de los de entrada, y que, de algún modo, los resumen (por ejemplo, la media de los vectores de caracterización).

El principal interés del informe de caracterización es presentar al usuario, de forma estructurada, toda esta información, para poder visualizarla de forma intuitiva y navegable.

3.2.2. Requisitos no funcionales

1. El sistema debe estar orientado a la facilidad de uso por parte del usuario. Por tanto, la aplicación será desarrollada como un clásico sistema de ventanas manejado fundamentalmente por ratón, con atajos de tecla y sistema de ayuda integrado.

2. El sistema debe proporcionar al usuario facilidad de navegación a través de las imágenes, sin que le sea complicado la visualización de una región concreta deseada.

3. El sistema debe proporcionar el clásico mecanismo de *arrastrar y soltar* (o *drag and drop*) que permite llevar acabo operaciones de arrastrado y soltado con elementos de utilidad, de modo que al usuario se le facilite en gran medida el manejo de la aplicación.

4. El sistema debe ser fácilmente extensible. Sin necesidad de recompilar o reinstalar la aplicación, la aplicación debería permitir añadir nuevos módulos de funcionalidad. En particular, el sistema debe dar soporte para añadir nuevas operaciones, tanto de tratamiento de imágenes como de caracterización.

5. Con la idea de desarrollar un sistema de software libre bajo licencia GNU GPL, se deberá tener especial cuidado en la elección de las herramientas usadas para el desarrollo. El producto final ha de regirse bajo las condiciones de esta licencia.

6. El software que se desarrollará deberá ser multiplataforma, siendo necesaria su compatibilidad con las plataformas win32, linux y macosx.

7. Se le entregará al cliente un manual de usuario donde se explique cómo llevar a cabo las tareas que se pueden realizar con el sistema.

3.3. Modelo funcional mediante casos de uso

Las necesidades expuestas anteriormente nos han llevado a realizar el modelado funcional de la aplicación mediante un modelo de casos de uso, que pasaremos a detallar a continuación.

3.3.1. Identificación de actores

El sistema no contempla más que un único actor estándar, con capacidad para acceder a toda la funcionalidad de la aplicación. En un entorno de trabajo abierto no tiene sentido el concepto de *tipo de usuario*, ya que la máxima finalidad es la de permitir al usuario, sea quien sea, explotar toda su capacidad. A este actor, de ahora en adelante, se le conocerá como *usuario*.

3.3.2. Identificación de los casos de uso

Los casos de uso de la aplicación pueden ser fácilmente divididos en paquetes, que los agrupan en casos de uso con una semántica relacionada.

3.3.2.1. Paquete PaqueteDeImagenes

Este paquete contiene los casos de uso asociados a la gestión de paquetes de imágenes dentro de la aplicación.

Nombre	Crear paquete de imágenes.
Resumen	Permite al usuario crear un paquete de imágenes.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario indica que desea crear un paquete de imágenes. 2. El sistema pide al usuario el nombre del paquete de imágenes. 3. El usuario especifica el nombre del paquete de imágenes. Si ya existe otro paquete con el mismo nombre, se debe introducir otro. 4. El usuario confirma la creación del paquete de imágenes. 5. El sistema crea el paquete de imágenes.
Curso alternativo	

Tabla 3.1: Caso de uso *Crear paquete de imágenes*

Nombre	Guardar paquete de imágenes.
Resumen	Permite al usuario guardar un paquete de imágenes en el almacenamiento local.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona el paquete de imágenes a guardar. 2. El usuario indica que quiere guardar el paquete de imágenes. 3. El sistema pide al usuario el nombre del paquete de imágenes. a. Si el fichero donde el usuario quiere guardar el paquete ya existe en el almacenamiento local, advierte de que ya existe un fichero con el mismo nombre, y pide confirmación para sobreescribirlo. a. Si el usuario confirma, se selecciona dicho nombre de archivo como destino donde guardar el paquete. b. Si el usuario no confirma, se vuelve al paso 3. 4. El sistema guarda el paquete de imágenes en el fichero especificado por el usuario.
Curso alternativo	

Tabla 3.2: Caso de uso *Guardar paquete de imágenes*

Nombre	Abrir paquetes de imágenes.
Resumen	Permite al usuario cargar en la aplicación varios paquetes de imágenes almacenados en el almacenamiento local.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario indica que desea cargar varios paquetes de imágenes. 2. El sistema pide al usuario los nombres de los archivos que contienen los paquetes de imágenes. 3. El sistema carga los paquetes de imágenes almacenados en los archivos especificados por el usuario.
Curso alternativo	

Tabla 3.3: Caso de uso *Abrir paquetes de imágenes*

Nombre	Añadir imágenes a paquete de imágenes.
Resumen	Permite al usuario añadir una o varias imágenes a un paquete de imágenes.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona el paquete o paquetes a los que añadir imágenes. 2. El usuario selecciona un repertorio de imágenes a añadir a los paquetes seleccionados. 3. El usuario indica al sistema que desea añadir las imágenes a los paquetes especificados. 4. El sistema añade a los paquetes las imágenes indicadas.
Curso alternativo	

Tabla 3.4: Caso de uso *Añadir imágenes a paquete de imágenes*

3.3.2.2. Paquete TratamientoDeImagenes

Este paquete contiene los casos de uso relacionados con el tratamiento de imágenes digitales dentro de la aplicación. Con ello nos referimos a la entrada y salida de imágenes, y manipulación mediante operaciones que permiten transformar imágenes de entrada en imágenes de salida.

Nombre	Abrir imagen.
Resumen	Permite cargar una imagen del almacenamiento local para su posterior tratamiento por la aplicación.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario especifica, del almacenamiento local, qué imagen desea cargar. 2. El sistema carga la imagen especificada por el usuario, y la muestra en pantalla.
Curso alternativo	2. La imagen especificada por el usuario no existe o no tiene un formato soportado por el sistema. El sistema muestra un mensaje de error, y finaliza el caso de uso.

Tabla 3.5: Caso de uso *Abrir imagen*

Nombre	Abrir imagen con ROI.
Resumen	Permite cargar una imagen del almacenamiento local para su posterior tratado por la aplicación. Si la imagen tiene un fichero de ROI asociado, la ROI es asociada a la imagen al ser cargada en la aplicación.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	
	1. El usuario especifica, del almacenamiento local, qué imagen desea cargar. 2. El sistema carga la imagen especificada por el usuario. Si existe un fichero de ROI con el mismo nombre que el de la imagen (omitiendo extensión), y con extensión «.roi», el sistema carga la ROI, y la asocia a la imagen cargada. 3. El sistema muestra la imagen cargada.
Curso alternativo	
	2. La imagen especificada por el usuario no existe o no tiene un formato soportado por el sistema. El sistema muestra un mensaje de error, y finaliza el caso de uso.

Tabla 3.6: Caso de uso *Abrir imagen con ROI*

Nombre	Cerrar imagen.
Resumen	Cierra una imagen cargada en la aplicación, bien porque el usuario la ha cargado desde el almacenamiento local, o bien porque se ha obtenido de otras imágenes, mediante algún tipo de manipulación.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona, de entre las imágenes cargadas, cuál quiere cerrar. 2. Si la imagen seleccionada no está guardada en el almacenamiento local, el sistema pide confirmación antes de cerrarla. 3. a. Si el usuario confirma positivamente, el sistema pide al usuario que especifique un nombre de archivo donde guardar la imagen, y guarda la imagen en dicho archivo. El sistema cierra la imagen seleccionada por el usuario, que deja de ser visible.
Curso alternativo	

Tabla 3.7: Caso de uso *Cerrar imagen*

Nombre	Guardar imagen.
Resumen	Guarda en el almacenamiento local una imagen cargada en la aplicación y previamente seleccionada por el usuario. Si la imagen dispone de una ROI asociada, ésta es guardada en otro fichero.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona, de entre las imágenes cargadas, cuál quiere guardar. 2. El sistema pide al usuario que especifique el nombre del fichero donde desea guardar la imagen seleccionada, así como el formato de almacenamiento. a. Si el fichero donde el usuario quiere guardar la imagen ya existe en el almacenamiento local, advierte de que ya existe un fichero con el mismo nombre, y pide confirmación para sobreescribirlo. a. Si el usuario confirma, se selecciona dicho nombre de archivo como destino donde guardar la imagen. b. Si el usuario no confirma, se vuelve al paso 2. 3. El sistema almacena la imagen en el fichero seleccionado y en el formato especificado. Si la imagen dispone de ROI, la almacena en un fichero del mismo nombre que el de la imagen (salvo extensión), y con extensión «.roi».
Curso alternativo	3. Si el nombre de la imagen coincide con el nombre de alguna de las imágenes abiertas en la aplicación, la imagen no es guardada. Al usuario se le muestra un mensaje de error informando de ello, y finaliza el caso de uso..

Tabla 3.8: Caso de uso *Guardar imagen*

Nombre	Ampliar imagen.
Resumen	Permite al usuario ampliar una imagen, para así poder visualizarla con mayor detalle. Este proceso se conoce comúnmente como *hacer zoom* de la imagen, y, en todo caso, no supone modificar la imagen subyacente, sino la manera de visualizarla.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona una imagen a ampliar. 2. El usuario indica al sistema que desea ampliar la imagen seleccionada. 3. El sistema amplia y muestra la imagen.
Curso alternativo	
Notas	Ver caso de uso *Reducir imagen*. El sistema no puede sobrepasar unos factores de zoom límites establecidos.

Tabla 3.9: Caso de uso *Ampliar imagen*

Nombre	Reducir imagen.
Resumen	Permite al usuario reducir el tamaño de una imagen, para así poder visualizar un mayor área de ésta. Este proceso se conoce comúnmente como *hacer zoom* de la imagen, y, en todo caso, no supone modificar la imagen subyacente, sino la manera de visualizarla.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona una imagen a reducir. 2. El usuario indica al sistema que desea reducir la imagen seleccionada. 3. El sistema reduce y muestra la imagen.
Curso alternativo	
Notas	Ver caso de uso *Ampliar imagen*. El sistema no puede sobrepasar unos factores de zoom límites establecidos.

Tabla 3.10: Caso de uso *Reducir imagen*

Nombre	Escalar área.
Resumen	Permite al usuario seleccionar un área rectangular de la imagen a visualizar. Dicho área será ampliada o reducida hasta ocupar un tamaño óptimo de visualización para el usuario. Este proceso se conoce comúnmente como *hacer zoom* de la imagen, y, en todo caso, no supone modificar la imagen subyacente, sino la manera de visualizarla.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	
	1. El usuario selecciona una imagen de la que desea visualizar un área rectangular específica. 2. El usuario indica al sistema el área rectangular que desea visualizar de dicha imagen, 3. El sistema amplia o reduce la imagen, visualizando el área rectangular especificada por el usuario anteriormente.
Curso alternativo	
Notas	Ver casos de uso *Ampliar imagen* y *Reducir imagen*. El sistema no puede sobrepasar unos factores de zoom límites establecidos.

Tabla 3.11: Caso de uso *Escalar área*

Nombre	Extraer ROI.
Resumen	Permite al usuario extraer una ROI de una imagen determinada. La ROI seleccionada será trasladada a una nueva imagen.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona una imagen de la que desea extraer una ROI. 2. El usuario selecciona una ROI arbitraria de la imagen seleccionada. 3. El sistema extraer la ROI especificada, formando una nueva imagen que sólo contiene la región especificada por la ROI definida.
Curso alternativo	

Tabla 3.12: Caso de uso *Extraer ROI*

Nombre	Definir ROI.
Resumen	Permite al usuario definir una ROI sobre una imagen determinada.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona una imagen sobre la que desea definir la ROI. 2. El usuario define, gráficamente, el aspecto de la ROI que quiere asociar a la imagen. 3. El sistema asocia la ROI definida a la imagen seleccionada.
Curso alternativo	

Tabla 3.13: Caso de uso *Definir ROI*

Nombre	Aplicar operación de tratamiento.
Resumen	Permite al usuario aplicar una operación de tratamiento de imágenes a una imagen seleccionada o a varias, dependiendo del tipo de operación.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona el tipo de operación que desea aplicar. 2. El sistema muestra al usuario los parámetros necesarios para llevar a cabo la operación. 3. El usuario introduce valores para dichos parámetros. 4. El usuario selecciona la imagen (o imágenes, en caso de requerirse varias), a la que quiere aplicar la operación. 5. El usuario confirma que desea aplicar la operación. 6. El sistema valida los parámetros introducidos. 7. El sistema aplica la operación a la imagen (o imágenes) seleccionada, generando una nueva imagen que es presentada en pantalla al usuario.
Curso alternativo	7. Si alguno de los parámetros introducidos no son válidos, el sistema muestra un mensaje de error al usuario, y se vuelve al paso 2.

Tabla 3.14: Caso de uso *Aplicar operación de tratamiento*

Nombre	Crear cadena de operaciones.
Resumen	Permite al usuario crear una cadena de operaciones.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario indica que desea crear una cadena de operaciones. 2. El sistema pide al usuario el nombre de la cadena de operaciones. 3. El usuario especifica el nombre de la cadena de operaciones. Si ya existe otra cadena con el mismo nombre, se debe introducir otro. 4. El usuario confirma la creación de la cadena de operaciones. 5. El sistema crea la cadena de operaciones.
Curso alternativo	

Tabla 3.15: Caso de uso *Crear cadena de operaciones*

Nombre	Guardar cadena de operaciones.
Resumen	Permite al usuario guardar una cadena de operaciones en el almacenamiento local.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	
	1. El usuario selecciona la cadena de operaciones a guardar. 2. El usuario indica que quiere guardar la cadena de operaciones. 3. El sistema pide al usuario el nombre del archivo donde guardar la cadena de operaciones. a. Si el fichero donde el usuario quiere guardar la cadena ya existe en el almacenamiento local, advierte de que ya existe un fichero con el mismo nombre, y pide confirmación para sobreescribirlo. a. Si el usuario confirma, se selecciona dicho nombre de archivo como destino donde guardar la cadena de operaciones. b. Si el usuario no confirma, se vuelve al paso 3. 4. El sistema guarda la cadena de operaciones en el fichero especificado por el usuario.
Curso alternativo	

Tabla 3.16: Caso de uso *Guardar cadena de operaciones*

Nombre	Abrir cadenas de operaciones.
Resumen	Permite al usuario cargar en la aplicación varias cadenas de operaciones almacenadas en el almacenamiento local.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario indica que desea cargar varias cadenas de operaciones. 2. El sistema pide al usuario los nombres de los archivos que contienen las cadenas de operaciones. 3. El sistema carga las cadenas de operaciones almacenadas en los archivos especificados por el usuario.
Curso alternativo	

Tabla 3.17: Caso de uso *Abrir cadenas de operaciones*

Nombre	Insertar operación en cadenas.
Resumen	Permite al usuario insertar una operación de tratamiento de imágenes en una o varias cadenas de operaciones.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona la operación que quiere insertar en las cadenas de operaciones. 2. El sistema muestra al usuario los parámetros necesarios para llevar a cabo la operación. 3. El usuario introduce valores para dichos parámetros. 4. El usuario selecciona las cadenas de operaciones en las que desea insertar la operación definida. 5. El usuario confirma que desea insertar la operación en las cadenas seleccionadas. 6. El sistema valida los parámetros introducidos. 7. El sistema inserta la operación en las cadenas seleccionadas.
Curso alternativo	7. Si alguno de los parámetros introducidos no son válidos, el sistema muestra un mensaje de error al usuario, y se vuelve al paso 2. Si la operación no es unaria (requiere una única imagen de entrada para obtener su resultado), se muestra un mensaje de error al usuario, y se vuelve al paso 2.

Tabla 3.18: Caso de uso *Insertar operación en cadenas*

Nombre	Procesar paquetes de imágenes.
Resumen	Permite al usuario procesar uno o varios paquetes de imágenes mediante una o varias cadenas de operaciones.
Dependencias	
Actores	Usuario
Precondiciones	
Curso normal	1. El usuario selecciona los paquetes de imágenes a procesar. 2. El usuario selecciona las cadenas de operaciones a aplicar a cada uno de los paquetes 3. El usuario indica al sistema que quiere procesar los paquetes especificados mediante las cadenas de operaciones indicadas. 4. Se llama al caso de uso *Configurar procesamiento de paquetes.* 5. Se procesan los paquetes de imágenes según la configuración establecida en el paso anterior, y se almacenan las imágenes procesadas en el directorio de salida.
Curso alternativo	

Tabla 3.19: Caso de uso *Procesar paquetes de imágenes*

Nombre	Configurar procesamiento de paquetes.
Resumen	Permite al usuario configurar ciertos detalles relativos al procesamiento de paquetes de imágenes mediante cadenas de operaciones.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario indica un pequeño patrón a añadir a los nombres de los ficheros de salida. 2. El usuario indica el formato de los ficheros de salida. 3. El usuario indica el directorio de salida de las imágenes procesadas. 4. El usuario especifica la ROI a usar durante el procesamiento. Se ofrecen tres alternativas: • Seleccionar un fichero de región de interés para ser aplicado sobre todas las imágenes. En este caso, la región de interés a procesar en todas las imágenes será la misma, es decir, la que está almacenada en el fichero. • Seleccionar la región de interés por defecto de cada imagen. La región de interés por defecto de una imagen está almacenada en un fichero cuyo nombre coincide con el del fichero de la imagen, y cuya extensión es «.roi». En este caso, para cada imagen, se buscará su fichero de región de interés asociado. Si no se encontrara alguno, dicha imagen sería procesada de forma completa, en vez de sólo la región de interés. • Especificar que no se quiere usar ninguna región de interés. En este caso, no se usará ninguna región de interés en el procesamiento de las imágenes. Todas las imágenes son procesadas de forma completa.
Curso alternativo	

Tabla 3.20: Caso de uso *Configurar procesamiento de paquetes*

3.3.2.3. Paquete CaracterizaciónDeImágenes

Este paquete contiene los casos de uso relacionados con la caracterización de imágenes dentro de la aplicación. Con ello nos referimos a la creación de informes de caracterización, así como a la entrada y salida de dichos informes.

Nombre	Aplicar operación de caracterización.
Resumen	Permite al usuario aplicar una operación de caracterización de imágenes a una imagen seleccionada.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona el tipo de operación que desea aplicar. 2. El sistema muestra al usuario los parámetros necesarios para llevar a cabo la operación. 3. El usuario introduce valores para dichos parámetros. 4. El usuario selecciona la imagen a la que quiere aplicar la operación de caracterización. 5. El usuario confirma que desea aplicar la operación. 6. El sistema valida los parámetros introducidos. 7. El sistema aplica la operación de caracterización a la imagen y genera, como resultado, un informe de caracterización que contiene la medida de caracterización computada sobre dicha imagen.
Curso alternativo	7. Si alguno de los parámetros introducidos no son válidos, el sistema muestra un mensaje de error al usuario, y se vuelve al paso 2.

Tabla 3.21: Caso de uso *Aplicar operación de caracterización*

Nombre	Abrir informe de caracterización.
Resumen	Permite cargar un informe de caracterización del almacenamiento local para su posterior tratamiento por la aplicación.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	
	1. El usuario especifica, del almacenamiento local, qué informe de caracterización desea cargar. 2. El sistema carga el informe de caracterización especificado por el usuario, y lo muestra en pantalla.
Curso alternativo	
	2. El informe de caracterización especificado por el usuario no existe o no tiene un formato soportado por el sistema. El sistema muestra un mensaje de error, y finaliza el caso de uso.

Tabla 3.22: Caso de uso *Abrir informe de caracterización*

Nombre	Cerrar informe de caracterización.
Resumen	Cierra un informe de caracterización cargado en la aplicación
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona, de entre los informes de caracterización cargados, cuál quiere cerrar. 2. Si el informe de caracterización seleccionado no está guardado en el almacenamiento local, el sistema pide confirmación antes de cerrarlo. 3. a. Si el usuario confirma positivamente, el sistema pide al usuario que especifique un nombre de archivo donde guardar el informe, y guarda el informe en dicho archivo. El sistema cierra el informe seleccionado por el usuario, que deja de ser visible.
Curso alternativo	

Tabla 3.23: Caso de uso *Cerrar informe de caracterización*

Nombre	Guardar informe de caracterización.
Resumen	Guarda en el almacenamiento local un informe de caracterización previamente seleccionado por el usuario.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona, de entre los informes de caracterización cargados, cual quiere guardar. 2. El sistema pide al usuario que especifique el nombre del fichero donde desea guardar la imagen seleccionada, así como el formato de almacenamiento. a. Si el fichero donde el usuario quiere guardar el informe de caracterización ya existe en el almacenamiento local, advierte de que ya existe un fichero con el mismo nombre, y pide confirmación para sobreescribirlo. a. Si el usuario confirma, se selecciona dicho fichero como destino donde almacenar el informe de caracterización. b. Si el usuario no confirma, se vuelve al paso 2. 3. El sistema almacena el informe en el fichero seleccionado.
Curso alternativo	3. Si el nombre del fichero especificado coincide con el nombre de alguno de los informes de caracterización abiertos en la aplicación, el informe no es guardado. Al usuario se le muestra un mensaje de error indicándolo, y finaliza el caso de uso.

Tabla 3.24: Caso de uso *Guardar informe de caracterización*

Nombre	Crear generador de vector de caracterización.
Resumen	Permite al usuario crear un generador de vector de caracterización.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario indica que desea crear un generador de vector de caracterización. 2. El sistema pide al usuario el nombre del generador de vector de caracterización. 3. El usuario especifica el nombre del generador de vector de caracterización. Si ya existe otro generador con el mismo nombre, se debe introducir otro. 4. El usuario confirma la creación del generador de vector de caracterización. 5. El sistema crea el generador de vector de caracterización.
Curso alternativo	

Tabla 3.25: Caso de uso *Crear generador de vector de caracterización*

Nombre	Guardar generador de vector de caracterización.
Resumen	Permite al usuario guardar un generador de vector de caracterización en el almacenamiento local.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona el generador de vector de caracterización a guardar. 2. El usuario indica que quiere guardar el generador de vector de caracterización. 3. El sistema pide al usuario el nombre del archivo donde guardar el generador. a. Si el fichero donde el usuario quiere guardar el generador ya existe en el almacenamiento local, advierte de que ya existe un fichero con el mismo nombre, y pide confirmación para sobreescribirlo. a. Si el usuario confirma, se selecciona dicho nombre de archivo como destino donde guardar el generador de vector de caracterización. b. Si el usuario no confirma, se vuelve al paso 3. 4. El sistema guarda el generador de vector de caracterización en el fichero especificado por el usuario.
Curso alternativo	

Tabla 3.26: Caso de uso *Guardar generador de vector de caracterización*

Nombre	Abrir generadores de vector de caracterización.
Resumen	Permite al usuario cargar en la aplicación varios generadores de vector de caracterización almacenados en el almacenamiento local.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario indica que desea cargar varios generadores de vector de caracterización. 2. El sistema pide al usuario los nombres de los archivos que contienen los generadores de vector de caracterización. 3. El sistema carga los generadores de vector de caracterización almacenados en los archivos especificados por el usuario.
Curso alternativo	

Tabla 3.27: Caso de uso *Abrir generadores de vector de caracterización*

Nombre	Insertar operación en generadores de vector de caracterización.
Resumen	Permite al usuario insertar una operación de caracterización de imágenes en uno o varios generadores de vector de caracterización.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona la operación que quiere insertar en los generadores de vector de caracterización. 2. El sistema muestra al usuario los parámetros necesarios para llevar a cabo la operación. 3. El usuario introduce valores para dichos parámetros. 4. El usuario selecciona los generadores de vector de caracterización en los que insertar la operación definida. 5. El usuario confirma que desea insertar la operación en los generadores seleccionados. 6. El sistema valida los parámetros introducidos. 7. El sistema inserta la operación en los generadores de vector de caracterización seleccionados.
Curso alternativo	7. Si alguno de los parámetros introducidos no son válidos, el sistema muestra un mensaje de error al usuario, y se vuelve al paso 2.

Tabla 3.28: Caso de uso *Insertar operación en generadores de vector de caracterización*

Nombre	Caracterizar paquetes de imágenes.
Resumen	Permite al usuario caracterizar las imágenes de uno o varios paquetes de imágenes mediante uno o varios generadores de vector de caracterización.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona los paquetes de imágenes a caracterizar. 2. El usuario selecciona los generadores de vector de caracterización con los que caracterizar las imágenes de los paquetes de imágenes. 3. El usuario indica al sistema que quiere caracterizar los paquetes especificados mediante los generadores de vector de caracterización indicados. 4. Se llama al caso de uso *Configurar caracterización de imágenes*. 5. Se caracterizan los paquetes de imágenes según la configuración establecida en el paso anterior, y se crea un informe de caracterización que contiene toda la información de la caracterización. Dicho informe de caracterización es mostrado al usuario.
Curso alternativo	

Tabla 3.29: Caso de uso *Caracterizar paquetes de imágenes*

Nombre	Caracterizar imágenes.
Resumen	Permite al usuario caracterizar una serie de imágenes seleccionadas, mediante uno o varios generadores de vector de caracterización. Las imágenes a caracterizar no están asociadas a un paquete de imágenes concreto.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario selecciona las imágenes a caracterizar. 2. El usuario selecciona los generadores de vector de caracterización con los que caracterizar las imágenes seleccionadas. 3. El usuario indica al sistema que quiere caracterizar las imágenes especificados mediante los generadores de vector de caracterización indicados. 4. Se llama al caso de uso *Configurar caracterización de imágenes*. 5. Se caracterizan las imágenes según la configuración establecida en el paso anterior, y se crea un informe de caracterización que contiene toda la información de la caracterización. Dicho informe de caracterización es mostrado al usuario.
Curso alternativo	

Tabla 3.30: Caso de uso *Caracterizar imágenes*

Nombre	Configurar caracterización de imágenes.
Resumen	Permite al usuario configurar ciertos detalles relativos a la caracterización de varias imágenes que van a ser caracterizadas de forma conjunta
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario especifica la ROI a usar durante la caracterización. Se ofrecen tres alternativas: • Seleccionar un fichero de región de interés para ser aplicado sobre todas las imágenes. En este caso, la región de interés a caracterizar en todas las imágenes será la misma, es decir, la que está almacenada en el fichero. • Seleccionar la región de interés por defecto de cada imagen. La región de interés por defecto de una imagen está almacenada en un fichero cuyo nombre coincide con el del fichero de la imagen, y cuya extensión es «.roi». En este caso, para cada imagen, se buscará su fichero de región de interés asociado. Si no se encontrara alguno, dicha imagen sería caracterizada de forma completa, en vez de sólo la región de interés. • Especificar que no se quiere usar ninguna región de interés. En este caso, no se usará ninguna región de interés en la caracterización de las imágenes. Todas las imágenes son caraterizadas de forma completa. 2. El usuario especifica si quiere calcular medidas de resumen del informe de caracterización, es decir, medidas de resumen sobre los vectores de caracterización del informe.
Curso alternativo	

Tabla 3.31: Caso de uso *Configurar caracterización de imágenes*

Nombre	Cerrar aplicación.
Resumen	Permite al usuario cerrar la aplicación.
Dependencias	
Actores	Usuario.
Precondiciones	
Curso normal	1. El usuario indica que desea cerrar la aplicación. a. Si la aplicación contiene alguna imagen abierta y no guardada en disco, pide al usuario confirmación para guardarla antes de cerrar. I. Si el usuario confirma positivamente, el sistema pide al usuario el nombre del fichero donde guardar la imagen, y la guarda. II. Si el usuario confirma negativamente, el sistema no almacena la imagen. 2. La aplicación se cierra y finaliza el caso de uso.
Curso alternativo	

Tabla 3.32: Caso de uso *Cerrar aplicación*

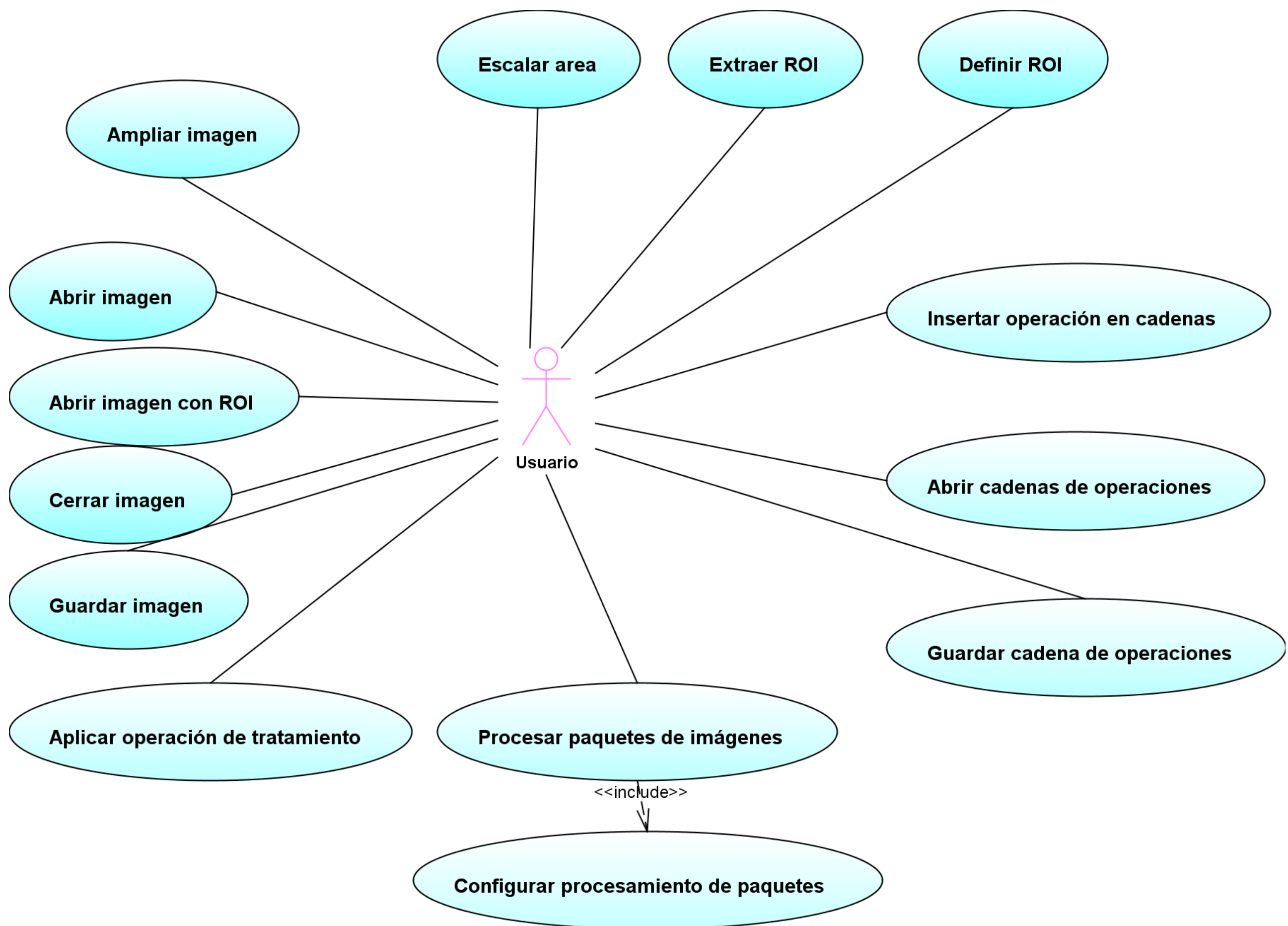

Figura 3.1: Diagrama de casos de uso del paquete TratamientoDeImagenes

3.3.3. Diagrama de casos de uso

A continuación se muestran los diagramas de casos de uso de cada un de los paquetes definidos en la sección 3.3.2.

3.3.3.1. TratamientoDeImagenes

El diagrama de casos de uso del paquete TratamientoDeImagenes se muestra en la figura 3.1.

3.3.3.2. PaqueteDeImagenes

El diagrama de casos de uso del paquete PaqueteDeImagenes se muestra en la figura 3.2.

3.3.3.3. CaracterizacionDeImagenes

El diagrama de casos de uso del paquete CaracterizacionDeImagenes se muestra en la figura 3.3.

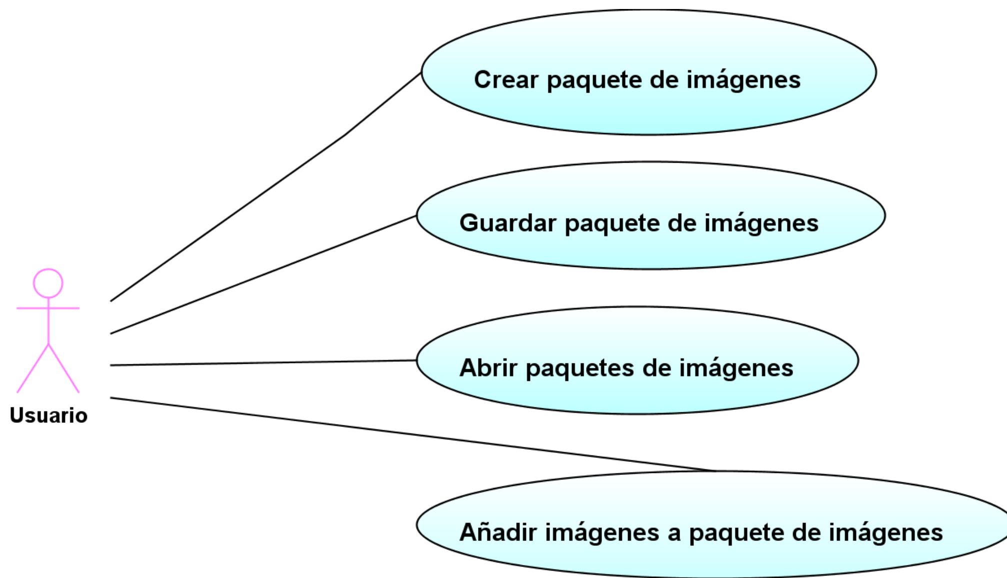

Figura 3.2: Diagrama de casos de uso del paquete PaqueteDeImagenes

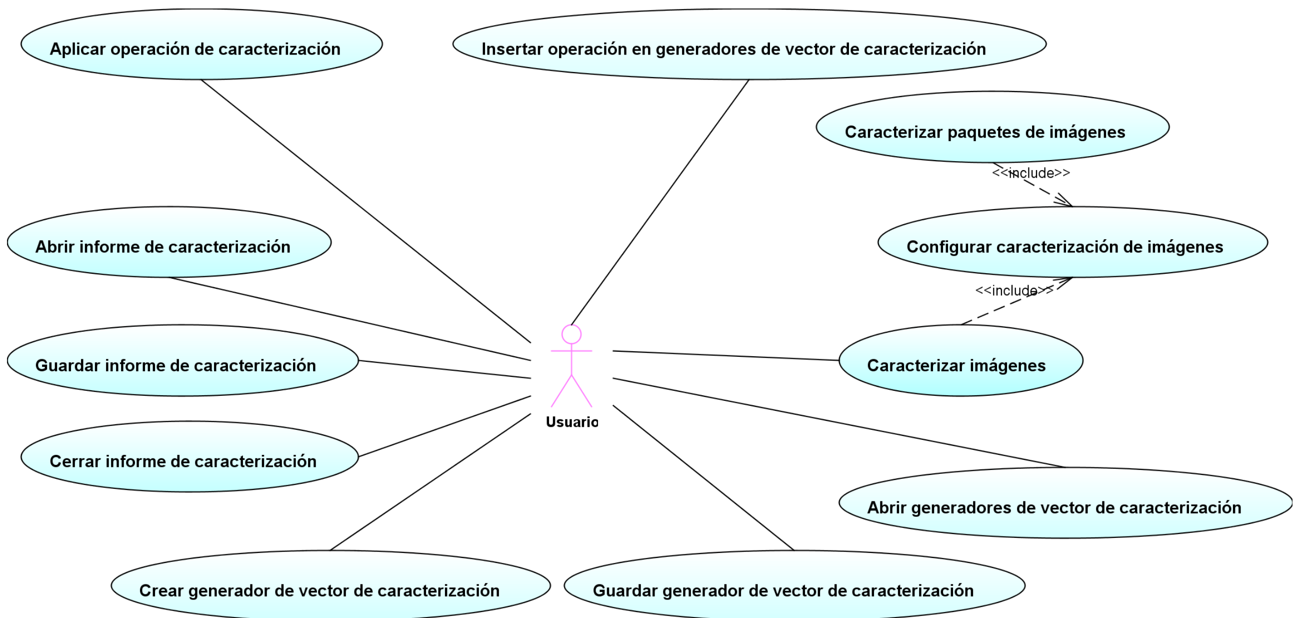

Figura 3.3: Diagrama de casos de uso del paquete CaracterizacionDeImagenes

Capítulo 4

Análisis

En la fase de análisis se describe el sistema en el dominio del problema. La principal motivación de la fase de análisis es la de modelar el problema a resolver mediante clases conceptuales y las relaciones existentes entre ellas.

En nuestro caso, nos centraremos en las principales clases conceptuales del dominio del problema. Sabemos, por experiencia, que una fase de análisis demasiado detallada no aporta beneficios al proceso de desarrollo del software. En muchas situaciones, la fase de análisis cae en el error de aporta información superflua. Siguiendo la línea del modelado de requisitos, utilizaremos una de las herramientas que nos ofrece UML para describir el dominio del problema, el diagrama de clases, que es uno de los modelos estáticos del sistema. Por conveniencia, hemos decidido dejar de un lado los diagramas de colaboración de la fase de análisis. El modelo de datos asociado al dominio del problema es lo suficientemente simple como para que seamos conscientes de que la mayor parte del trabajo de ingeniería recaerá en la fase del diseño. Así pues, y considerando trabajos previos de ingeniería del software, hemos decidido posponer la realización de los diagramas de colaboración a la siguiente etapa del proceso del software: el diseño. Los diagramas de secuencia y de colaboración resultan ser una herramienta muy útil utilizada sobre el diagrama de clases de **diseño**, donde se puede indicar con exactitud la lógica de las principales operaciones del diagrama.

Por tanto, nuestra fase de diseño se centrará en describir conceptualmente las principales clases del problema a resolver. Describiremos, además, ciertas componentes que aprovechamos de la arquitectura RCP de Eclipse para dotar de extensibilidad a nuestra aplicación. Con ello, esperamos definir de forma precisa los puntos claves del problema. Lo demás, a saber, la definición de la estructura de clases Java que implementen el sistema, será dejado para la fase de diseño (sección 6).

4.1. Modelado estático

Durante la fase del modelado estático se identifican las clases de diseño, así como sus atributos y las relaciones existentes entre ellas.

El diagrama de clases de análisis del modelo de datos de nuestro problema es el que se representa en la figura 4.1.

4.1.1. Diccionario de clases

4.1.1.1. Imagen

La clase *Imagen* es una de las principales clases a considerar a la hora de analizar el problema desde un punto de vista abstracto. Cool Imaging es una herramienta de tratamiento y de

Figura 4.1: Diagrama de análisis del modelo de datos

caracterización de imágenes. Por tanto, debe dar un soporte robusto para la manipulación de éstas.

Cool Imaging hace uso de la librería JAI (*Java Advanced Imaging*) para la manipulación de imágenes. Por tanto, el aspecto relativo al análisis de una entidad de tipo *imagen* está condicionado a lo que la librería nos ofrece.

JAI y Java 2D API[1] se solapan en ciertos aspectos. En lo que respecta a la representación de una imagen, JAI considera que es necesario disponer de:

- Un *Raster*: un *Raster* almacena los datos de la imagen. Dependiendo de la imagen, los datos pueden organizarse de muy diferentes maneras dentro de un Raster.

- Un *SampleModel*: el *SampleModel* se encarga de, dado un *Raster*, obtener los píxeles de la imagen. Se podría decir que el *SampleModel conoce* cómo están organizados los datos dentro del *Raster*, y que, por tanto, puede obtener los píxeles de la imagen a partir de éste.

- Un *ColorModel*: el *ColorModel* de una imagen se encarga de, dado un píxel producido por el *SampleModel*, convertir las muestras del píxel a muestras de color. En ciertos contextos, el *ColorModel* es prescindible, pero hay innumerables circunstancias donde se hace necesario.

En este sentido, JAI y Java 2D API son muy similares, pues ambas requieren, para la representación de imágenes, los mismos elementos[2].

El modelo de imágenes de JAI, además, considera la posibilidad de añadir *propiedades* a una imagen. Una *propiedad* no es más que un objeto que se puede asociar a la imagen, y que, de algún modo, puede resultar de utilidad.

4.1.1.2. ROI

La clase *ROI* representa una región de interés sobre una imagen (*Imagen*). Una *Imagen* puede o no tener asociada una *ROI*.

4.1.1.3. PaqueteImagenes

La clase *PaqueteImagenes* representa un conjunto de imágenes (paquete de imágenes). Todo objeto de la clase *PaqueteImagenes*, simplemente, almacena varios objetos de tipo *Imagen*.

El propósito de la clase *PaqueteImagenes* es poder agrupar de forma lógica varias imágenes. Para ello, la clase *PaqueteImagenes* consta de un identificador, que permite distinguirlo de otros paquetes de imágenes.

4.1.1.4. MC

La clase *MC* representa una *medida de caracterización*. Una *medida de caracterización* representa una medida numérica que recoge una propiedad de una imagen.

[1] Java 2D API es la API ofrecida de forma estándar en el lenguaje Java, para la manipulación de imágenes.

[2] En realidad, esto no es estrictamente cierto, pues Java 2D API ofrece diversos modelos de representación de imágenes. El modelo *Immediate Imaging Model* es el que consta de grandes similitudes con las clases que representan imágenes en JAI. Para más información acerca de los modelos de imágenes en Java, se recomienda leer *Programmer's Guide to the Java 2D API*, la guía oficial de Sun, y *Java Media APIs: Cross-Platform Imaging, Media, and Visualization*, de Alejandro Terrazas, John Ostuni y Michael Barlow.

4.1.1.5. VC

La clase *VC* representa un *vector de caracterización*. Un *vector de caracterización* representa un conjunto de medidas de caracterización calculadas sobre una imagen. Así, un *VC* no es más que un repertorio de objetos de tipo *MC*.

4.1.1.6. OperacionTratamiento

La clase *OperacionTratamiento* representa una operación que es capaz de tratar digitalmente de cero a varias imágenes de entrada, para producir una imagen de salida.

El esquema general de funcionamiento de una *OperacionTratamiento* es sencillo: partiendo de las imágenes de entrada (en caso de haberlas), y de los parámetros de la operación, se lleva a cabo un cómputo que tiene como resultado la obtención de una imagen de salida. Un parámetro de cómputo de una *OperacionTratamiento* está representado mediante un objeto de la clase *ParametroOperacion*. En general, cualquier dato que la operación requiera para obtener su resultado, como puede ser las imágenes de entrada, o datos específicos de la operación (por ejemplo, el valor del ángulo a rotar una imagen en una operación de rotación), están encapsulados mediante objetos de la clase *ParametroOperador*.

Se define, pues, un método llamado *operar()*, encargado de realizar el cómputo de la operación.

4.1.1.7. OperacionCaracterizacion

La clase *OperacionCaracterizacion* representa una operación capaz de obtener, a partir de una imagen, una *medida de caracterización* (*MC*). Recuérdese que una medida de caracterización es la unidad fundamental de información en la caracterización de imágenes, siendo un dato numérico, con estructura, computado a partir de la imagen.

El esquema de funcionamiento de una *OperacionCaracterizacio* es análogo al de una *OperacionTratamiento*: parte de una imagen de entrada, y a partir de ella y de los parámetros de la operación, lleva a cabo un cómputo que tiene como resultado la obtención de la medida de caracterización. Además, al igual que con una *OperacionCaracterizacion*, los parámetros de cómputo requeridos por la operación se representan mediante la clase *ParametroOperacion*.

Se define, pues, un método llamado *operar()*, encargado de realizar el cómputo de la operación.

4.1.1.8. ParametroOperacion

La clase *ParametroOperacion* representa un parámetro usado por una *OperacionTratamiento* o por una *OperacionCaracterizacion* para la obtención de sus resultados.

El esquema de funcionamiento de las operaciones es bastante sencillo: una operación requiere parámetros para obtener su resultado. Dichos parámetros están representados mediante objetos de la clase *ParametroOperacion*. Cada objeto de esta clase representa un parámetro de la operación, y almacena del valor de dicho parámetro.

4.1.1.9. CadenaOperacionesTratamiento

Una *CandenaOperacionesTratamiento* es una secuencia ordenada de operaciones de tratamiento de imágenes (*OperacionTratamiento*), con la capacidad de poder aplicarlas a una cierta imagen de entrada. La idea de una *CadenaOperacionesTratamiento* es la de poder aplicar varias operaciones, de forma consecutiva, a una imagen.

Cada una de las operaciones que contiene la *CadenaOperacionesTratamiento* debe admitir una única imagen de entrada. Así, la imagen de entrada a la cadena es procesada por la primera

operación, produciendo una primera imagen resultado. Esta primera imagen resultado será procesada por la segunda operación de la cadena, produciendo una segunda imagen resultado. El proceso se repite hasta que no hay más operaciones dentro de la cadena. La última imagen obtenida es el resultado devuelto por la cadena de operaciones. El método *operar()* se encarga de llevar a cabo este proceso, devolviendo como resultado la imagen procesada.

Toda *CadenaOperacionesTratamiento*, además, consta de un identificador, que permite diferenciarla de otras cadenas.

4.1.1.10. GeneradorVC

Un *GeneradorVC* representa un repertorio de operaciones de caracterización de imágenes (*OperacionCaracterizacion*), con la capacidad de poder aplicarlas a una imagen de entrada, para obtener como resultado un determinado vector de caracterización (*VC*).

Un *GeneradorVC* hace uso de cada una de las operaciones de caracterización que contiene, aplicándolas a la imagen de entrada. Como resultado, obtiene varias medidas de caracterización (*MC*), una por cada operación aplicada a la imagen de entrada. Con todas dichas *MC*, el *GeneradorVC* construye un vector de caracterización (*VC*), el cual contiene todas las *MC* creadas.

El método *operar()* de la clase *GeneradorVC* se encarga de llevar a cabo este proceso.

Todo *GeneradorVC* contiene un identificador que permite diferenciarlo de otros generadores.

4.1.1.11. InformeCaracterizacion

La clase *InformeCaracterizacion* representa un *informe de caracterización*. Un *informe de caracterización* se encarga de recoger toda la información relativa a la caracterización de una o varias imágenes, que pueden provenir de paquetes de imágenes (*PaqueteImagenes*) o no. Tal y como se especificó en la sección 3.2.1.1, un informe de caracterización debe permitir al usuario, fundamentalmente, visualizar todos los vectores de caracterización asociados a todas las imágenes caracterizadas.

Un proceso de caracterización puede llevarse a cabo a nivel de paquetes de imágenes o bien a nivel de imágenes aisladas (imágenes que no están asociadas a ningún paquete de imágenes).

Si la caracterización se realiza a nivel de paquetes, el informe almacena la correspondencia entre cada imagen y el paquete del que previene. Para cada imagen, además, almacena un listado con todos los vectores de caracterización obtenidos de dicha imagen. Recuérdese que un informe de caracterización puede incluir *medidas de resumen*, calculadas a partir de varios vectores de caracterización. Cada paquete de imágenes de un informe de caracterización puede incluir varias medidas de resumen de los vectores de caracterización de las imágenes de dicho paquete.

Cuando la caracterización se realiza a nivel de imágenes aisladas, su estructura es similar: almacena, para cada una de las imágenes caracterizadas, un listado con todos los vectores de caracterización obtenidos de dicha imagen. El informe de caracterización puede incluir medidas de resumen de los vectores de caracterización de las imágenes aisladas.

En cualquier caso, tiene sentido que la información almacenada por el *InformeCaracterizacion* esté estructurada de forma jerárquica. La clase *InformeCaracterizacion* debe ofrecer métodos para añadir nueva información al informe, de la manera más cómoda posible.

La clase *InformeCaracterizacion* ofrece dos métodos que permiten añadir contenido al informe. Estos métodos se llaman *registrar()*, y permiten registrar una *MC* en un *VC* de una *Imagen* del *InformeCaracterizacion*. La imagen puede estar asociada a un paquete de imágenes o no.

4.1.1.12. ProcesadorPaquetesImagenes

La clase *ProcesadorPaquetesImagenes* representa una clase con la capacidad de procesar, de forma automática, varios paquetes de imágenes, mediante varias cadenas de operaciones.

Un *ProcesadorPaquetesImagenes* almacena varios paquetes de imágenes (*PaqueteImagenes*) así como varias cadenas de operaciones de tratamiento (*CadenaOperacionesTratamiento*). El *ProcesadorPaquetesImagenes* procesa todas las imágenes de todos los *PaqueteImagenes* que tiene asociados, mediante todas las *CadenaOperacionesTratamiento* que tiene asociadas.

El procesamiento de las imágenes de los paquetes tiene asociados ciertos parámetros de configuración que determinan cómo se lleva a cabo el procesamiento. Dichos parámetros están especificados en un objeto de tipo *ConfiguracionProcesadorPaquetesImagenes*.

El algoritmo de procesamiento del *ProcesadorPaquetesImagenes* recorre todos los paquetes de imágenes. Para cada paquete, recorre todas sus imágenes, y para cada imagen, la procesa con cada una de las cadenas de operaciones. Así pues, por cada imagen de cada paquete, se obtienen varias imágenes de salida, una por cada cadena de operaciones. Cada una de estas imágenes de salida es almacenada en disco.

El método *procesar(ConfiguracionProcesadorPaquetesImagenes)* realiza el procesamiento de los paquetes de imágenes.

4.1.1.13. ConfiguracionProcesadorPaquetesImagenes

Esta clase contiene los parámetros de configuración que especifican ciertos detalles del procesamiento de paquetes de imágenes que es llevado a cabo por la clase *ProcesadorPaqueteImagenes*.

4.1.1.14. CaracterizadorPaquetesImagenes

La clase *CaracterizadorPaquetesImagenes* es la encargada de llevar a cabo la caracterización de varios paquetes de imágenes (*PaqueteImagenes*) mediante varios generadores de vector de caracterización (*GeneradorVC*).

La caracterización de los paquetes de imágenes tiene como resultado la creación de un informe de caracterización (*InformeCaracterizacion*). El *CaracterizadorPaquetesImagenes* toma cada una de las imágenes de los paquetes de imágenes, y caracteriza a cada una mediante todos los *GeneradorVC* que almacena. Así, por cada imagen, se obtienen tantos vectores de caracterización (*VC*) como *GeneradorVC* hay asociados al *CaracterizadorPaquetesImagenes*.

La caracterización de las imágenes de los paquetes tiene asociados ciertos parámetros de configuración que determinan cómo se lleva a cabo la caracterización. Dichos parámetros están especificados en un objeto de tipo *ConfiguracionCaracterizadorPaquetesImagenes*.

El método *procesar(ConfiguracionCaracterizadorPaquetesImagenes)* lleva a cabo la caracterización de las imágenes de los paquetes, produciendo como resultado un *InformeCaracterizacion*.

4.1.1.15. ConfiguracionCaracterizadorPaquetesImagenes

Esta clase contiene los parámetros de configuración que especifican ciertos detalles de la caracterización de paquetes de imágenes que es llevada a cabo por la clase *CaracterizadorPaquetesImagenes*.

4.2. Arquitectura del sistema

Una vez que hemos realizado un análisis detallado del dominio del problema, debemos analizar qué posibilidades tenemos para poder implementar la aplicación que satisfaga las necesidades anteriores. La estructura completa de la aplicación, descrita como subsistemas estructurales y las relaciones entre ellos, es lo que se conoce como la *arquitectura del sistema*.

4.2.1. Arquitectura basada en *plugins*

La arquitectura elegida para el sistema ha estado fuertemente condicionada por el requerimiento funcional de la extensibilidad.

Dentro de la ingeniería del software profesional, es común hacer referencia al concepto de *sistema extensible*. Sin embargo, el concepto de extensibilidad de un sistema no se mide en términos absolutos, pues un sistema puede gozar de distintos grados de extensibilidad. Así, por ejemplo, hay sistemas que pueden considerarse fácilmente extensibles aun considerando el hecho de que, para añadirles nueva funcionalidad, sea necesario recompilarlos íntegra o parcialmente. Estos sistemas podrían considerarse extensibles si, por ejemplo, añadir nueva funcionalidad al código fuente fuera relativamente sencillo, y no desencadenara un efecto dominó de cambios imprevisibles.

Sin embargo, este grado de extensibilidad está obsoleto en la actualidad, y sólo es usado en sistemas de baja importancia en los que el hecho de tener que recompilarlos y redistribuirlos a los usuarios no se ve como un verdadero problema.

Una aplicación de gran tamaño y cuya comunidad de usuarios puede ascender hasta las miles o decenas de miles de personas, requiere una solución más robusta, que no requiera la recompilación y redistribución de la aplicación íntegra (cuyo tamaño puede llegar hasta los cientos de mega bytes). Es así como surge la filosofía del diseño de aplicaciones mediante arquitectura de *plugins*.

En un sistema basado en *plugins*, todas las componentes del sistema están encapsuladas en *plugins*, los cuales están interconectados entre sí. Un *plugin* no es más que una caja negra que ofrece al exterior una interfaz pública que puede ser usada por otros *plugins*.

Un sistema basado en *plugins* consta de un *plugin* principal (de ahora en adelante *PP*), encargado de arrancar el sistema, y una serie de *plugins* que, conectados al principal, y a su vez interconectados entre sí, permiten su funcionamiento.

La arquitectura basada en *plugins* aporta dos grandes ventajas al desarrollo de software:

1. **Alta modularidad**: ya que la misma arquitectura obliga a los desarrolladores a diseñar módulos independientes, con una interfaz pública bien definida desde el principio.

2. **Extensibilidad**: dentro de la interfaz pública del *plugin* existe lo que se conoce como *puntos de extensión*. Un *punto de extensión* permite a un *plugin* definir un *modo* a través del cual, otros *plugins* externos, pueden extender su funcionalidad.

La interfaz pública de un *plugin* se divide en dos partes bien diferenciadas.

Por un lado, un *plugin* puede definir clases e interfaces públicas. Esto significa que cualquier otro *plugin* puede hacer uso de las clases e interfaces que están marcadas como públicas.

Por otro lado, la interfaz pública del *plugin* puede definir uno o varios puntos de extensión (no es obligatorio definir puntos de extensión). Con toda seguridad, el aspecto más relevante de la arquitectura basada en *plugins* es su extensibilidad mediante el sistema de puntos de extensión.

Un punto de extensión permite a un *plugin* especificar un *cierto modo* de extender su funcionalidad. Supongamos, por ejemplo, que se pretende diseñar un reproductor de audio

capaz de reproducir ficheros en distintos formatos. Sería interesante que la aplicación permitiera añadir nuevos formatos de reproducción de ficheros de música. Así, en caso de que se inventara un nuevo formato, la aplicación no quedaría obsoleta, pues podría incorporarlo, y así mantenerse actualizada.

Basándonos en la arquitectura de *plugins*, la aplicación podría diseñarse como un único *PP*, el cual definiera un punto de extensión que permitiera añadir al sistema nuevos formatos de reproducción de ficheros de música. Es más, la aplicación podría definir otros puntos de extensión que permitieran extender su funcionalidad más allá de la mera adición de nuevos formatos de reproducción.

Un *plugin* debe declarar aquellos puntos de extensión que quiere ofrecer al mundo exterior. En general, un punto de extensión se define a través de una interfaz o una clase abstracta. Esto significa que, todo *plugin* que quiera extender la funcionalidad del *plugin* que declara el punto de extensión, deberá ofrecer una clase que, o bien implemente la interfaz, o bien herede de la clase abstracta que define el punto de extensión.

Siguiendo con el ejemplo del reproductor de música, el punto de extensión del *PP* podría definirse mediante una interfaz que declarara los métodos básicos necesarios para la reproducción de un fichero de música. Para todo formato capaz de ser reproducido mediante la aplicación, debería existir una clase que implementara dicha interfaz. Dicha interfaz, por ejemplo, podría ser como la siguiente:

```
public interface IReproductorFormato{
   /*
    * Devuelve un String con el formato de lectura soportado por
    * este reproductor. Un ejemplo podría ser "mp3".
    */
   public String getFormatoSoportado();

   /*
    * Este método abre el fichero de entrada, que se supone que
    * está en el formato devuelto por el método
    * "getFormatoSoportado()", y reproduce el audio almacenado
    * en él.
    */
   public void reproducirFichero(String nombreFichero);
}
```

El primer método de la interfaz, *getFormatoSoportado()*, devolvería un String representando el formato soportado por el lector. El método *reproducirFichero()* se encargaría de reproducir un fichero de música en el formato soportado por el lector.

Tendríamos pues que el *PP* declararía un punto de extensión a través de la interfaz *IReprocutorFormato*. Este punto de extensión debe tener un nombre o identificador, que podría ser, por ejemplo, *NuevoFormato*. Si ahora se quisiera añadir soporte para un nuevo formato de audio, bastaría crear un *plugin* externo (*PE* de ahora en adelante), que contuviera una clase, digamos *MiFormato*, que implementara la interfaz *IReproductorFormato*. La clase *MiFormato* se encargaría de la reproducción de ficheros de audio en el formato que se quisiera añadir. Este *PE* debería *conectarse* al *PP* a través del punto de extensión *NuevoFormato*, para que el *PP* tuviera conocimiento del nuevo formato añadido.

El *PP*, por otro lado, debería tener constancia de qué *plugins* hay conectados a él a través del punto de extensión *NuevoFormato*. De ello se encargaría la misma arquitectura de *plugins*. La arquitectura de *plugins* permite a un *plugin* conocer qué otros *plugins* hay conectados a los puntos de extensión que define. Volviendo al ejemplo del reproductor de audio, el *PP* sería capaz

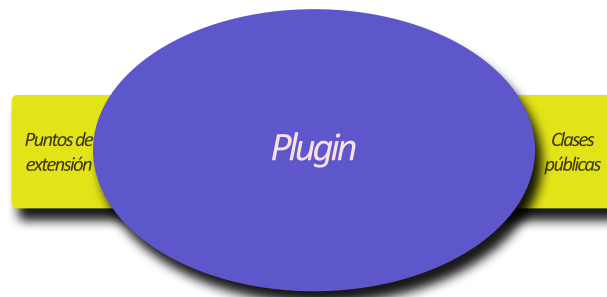

Figura 4.2: Un *plugin*

de saber que hay un plugin, *PE*, conectado a él a través del punto de extensión *NuevoFormato*. Es más, la arquitectura de *plugins* permite crear instancias de las clases que implementan (o extienden) la interfaz (o clase abstracta) definida en el punto de extensión. Esto significa que, por ejemplo, nuestro *PP* podría crear una instancia de la clase *MiFormato*, que originalmente estaba definida en el *PE*. El *PP* usaría dicha instancia para la reproducción de ficheros de audio en el formato soportado por la clase *MiFormato*.

Es importante notar que el *PP* no tendría conocimiento explícito de la clase *MiFormato*. Realmente, cuando la arquitectura subyacente de *plugins* crea el objeto de tipo *MiFormato*, el *PP* sólo puede manejarlo a través de la interfaz *IReproductorFormato* (realizando un *casting*). A través de dicha interfaz, el *PP* podría manejar el objeto *MiFormato* creado, y, con él, reproducir ficheros de audio en un nuevo formato[3].

La figura 4.2 representa un *plugin* aislado, cuya interfaz pública define una serie de puntos de extensión así como clases (y/o interfaces) accesibles desde el exterior.

La figura 4.3, por otro lado, representa el esquema de conexión básico entre dos *plugins*, el *Plugin A* y el *Plugin B*. El *Plugin A* define una serie de puntos de extensión así como clases (y/o interfaces) públicas. El *Plugin B*, por otro lado, hace uso de los puntos de extensión del *Plugin A*, así como de sus clases públicas. Por simplicidad, el *Plugin B* ha sido representado sin puntos de extensión y sin interfaz pública.

Un sistema basado en *plugins* real tiene una estructura bastante más compleja: decenas o incluso centenas de *plugins* se interconectan entre sí. Parte de esos *plugins* definen, simplemente, interfaces o clases públicas de utilidad para otros *plugins*. Otra parte, por otro lado, también define puntos de extensión a través de los cuales otros *plugins* pueden extender su funcionalidad. En el centro de todo el sistema, un *plugin* principal es el encargado de iniciar el sistema. La estructura de un sistema basado en plugins está representada en la figura 4.4.

4.2.2. Patrones de diseño y arquitectura del sistema

Dentro del proceso de desarrollo del software es frecuente encontrar situaciones en las que un problema a resolver ya ha sido resuelto con anterioridad. En vez de resolver el problema desde el principio, se intenta hacer uso de las soluciones propuestas en el pasado, eligiendo la que más se amolde a los requerimientos del problema.

Es en este contexto donde surgen los patrones diseño de software. Un patrón de diseño de software no es más que una solución estándar que se da a un problema que se suele presentar durante el diseño de software. Los patrones de diseño, en muchas circunstancias, fuerzan a que todo o parte del sistema siga una determinada arquitectura. A continuación se explicarán los

[3]Es necesario hacer hincapié en el hecho de que el mecanismo de los puntos de extensión es bastante más potente de lo que aquí se recoge. Sin embargo, en el contexto de Cool Imaging, los requerimientos de extensibilidad son ampliamente satisfechos por puntos de extensión que definen clases o interfaces que otros *plugins* deben extender o implementar.

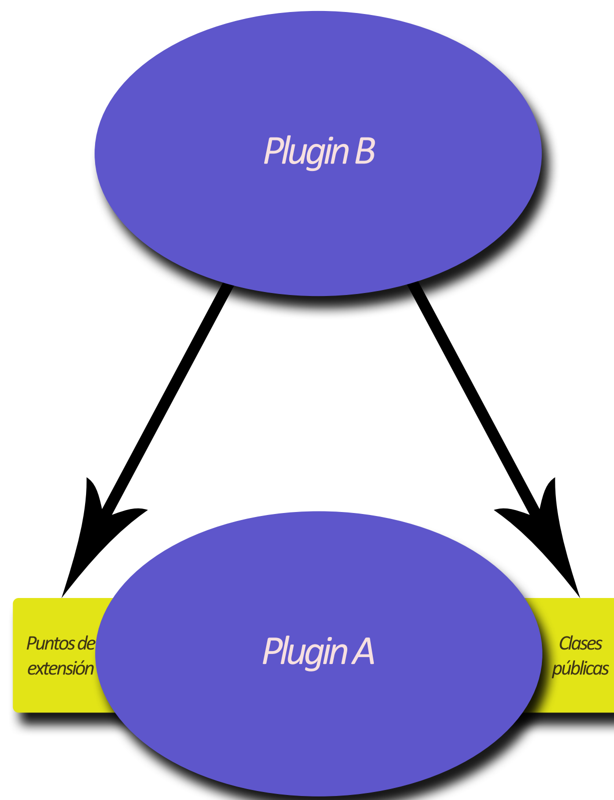

Figura 4.3: Esquema de conexión básico entre dos *plugins*

dos patrones de software más importantes y recurrentes a lo largo del diseño de la aplicación.

4.2.2.1. Patrón Modelo-Vista-Controlador

El patrón ***Modelo-Vista-Controlador*** (MVC) es un patrón software que se usa en multitud de ocasiones. Este patrón da solución al problema de tener un modelo de datos que puede ser visualizado de diferentes maneras, y donde es fundamental la separación entre el modelo de datos y su representación gráfica. Dentro del patrón MVC se distinguen tres componentes fundamentales, a saber, el *modelo*, la *vista* y el *controlador*.

El modelo representa aquellos datos que quieren ser visualizados, a ser posible, de diversas maneras. Una vista representa **una** posible representación gráfica del modelo. Se dice *una*, porque un modelo puede tener asociadas varias vistas, cada una representando al modelo de forma distinta. El controlador es aquella clase encargada de dirigir el flujo de información entre la vista y el modelo. El controlador es, en general, el componente que crea una mayor confusión, ya que su funcionalidad suele ser un tanto difusa. Es más, existen distintas implementaciones del patrón MVC en las que el controlador desempeña una tarea distinta.

De forma detallada, cada uno de los elementos constituyentes del patrón MVC desempeña la siguiente función:

- **Modelo**: el modelo representa el conocimiento, es decir, los datos de la aplicación. El propósito final del modelo es, no sólo ser capaz de almacenar dichos datos de forma consistente, sino además controlar todas las posibles transformaciones que puedan sufrir. El modelo nunca conoce a los controladores o a las vistas. En este sentido, es la componente del patrón más aislada. Es responsabilidad del sistema mantener los enlaces necesarios entre el modelo y sus respectivas vista, así como notificar a las vistas en el caso de que se produzca un cambio en el modelo (para que así éstas se actualicen y reflejen correctamente los cambios producidos en el modelo). En general, las vistas pedirán información al modelo

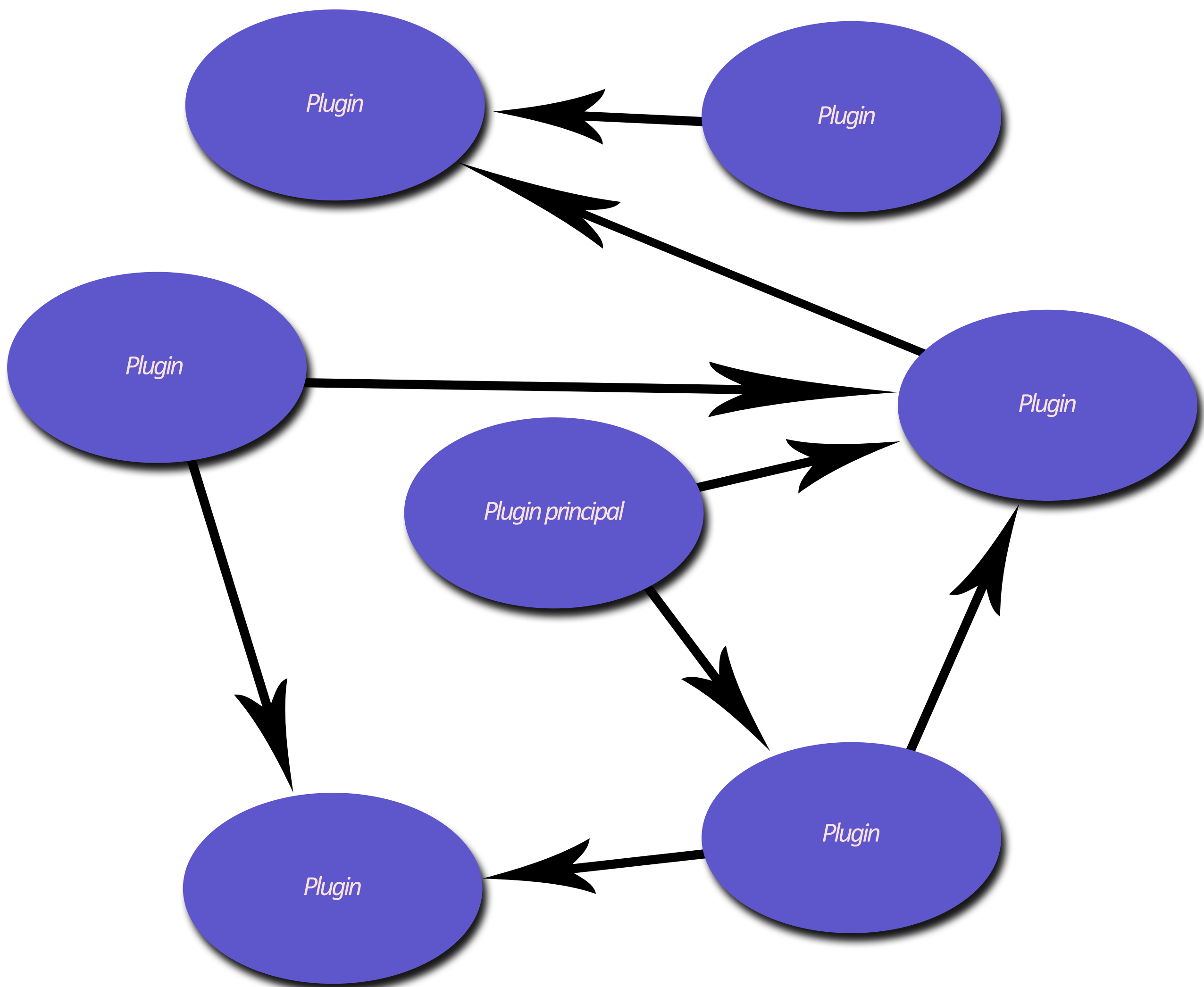

Figura 4.4: Arquitectura de un sistema basado en *plugins*

(información que visualizarán), y el controlador llevará a cabo modificaciones en el estado del modelo (a través de la interfaz pública del modelo).

- **Vista**: la vista es el objeto que se encarga de la representación visual de los datos almacenados en el modelo. Su objetivo fundamental es el de producir una representación visual de los datos del modelo. Dependiendo de la implementación del patrón, la vista puede tener o no una referencia al modelo. Si la tiene, el acceso a la información del modelo de datos se realiza a través de esta referencia. En caso de que no la tenga, la vista accederá a los datos del modelo a través del controlador, del cual deberá tener una referencia forzosamente.

- **Controlador**: el controlador es el encargado de dirigir la interacción entre el modelo y la vista. Existen, básicamente, tres tipos de controlador. En cualquier caso, el controlador siempre tiene referencias tanto al modelo de datos como a todas sus vistas:

  - **Controlador de instanciación**: este tipo de controlador es el más sencillo de todos, y el que tiene menos responsabilidades. Se encarga de iniciar el flujo de la aplicación, creando tanto el modelo como las vistas, y asociando el modelo con sus respectivas vistas (es decir, dando, a cada vista, una referencia del modelo). El controlador de instanciación no se encarga de la comunicación entre las vistas y el modelo. Cada vista podrá acceder al modelo de datos a través de la referencia que el controlador le ha proporcionado. Además, cualquier tipo de evento generado en la vista, y que implique modificación del modelo, deberá ser gestionado por la vista, que se encargará de modificar directamente el modelo, a través de la referencia que almacena.
  - **Controlador puro**: el controlador puro se encarga de actuar sobre los datos del modelo. Cualquier cambio realizado en el modelo de datos debe hacerse a través del controlador. Así, si en la vista se genera un evento por parte del usuario que implica un cambio en el modelo, la vista deberá delegar en el controlador dicha responsabilidad. Cuando se hace uso de este tipo de controlador, la vista mantiene la referencia al modelo.
  - **Controlador de salida**: el controlador de salida es una versión más restrictiva del controlador puro. Cuando se hace uso del controlador de salida, éste se encarga de hacer de pasarela entre el modelo y las vistas. En este caso, la vista no tiene una referencia al modelo, sino al controlador. De este modo, para consultar la información del modelo, debe hacer uso del controlador, el cual le proporciona los datos del modelo.

El flujo de ejecución de una aplicación que hace uso del patrón MVC se resume en los siguientes pasos:

1. El usuario interacciona con la vista de algún modo.

2. Si la implementación del controlador es puro (o de salida), éste recibe, por parte de la vista, la notificación de una acción solicitada por parte del usuario. El controlador se encarga de gestionar dicho evento. Si, por contra, el controlador es de instanciación, es la vista la encargada de gestionar el evento.

3. El controlador o la vista (dependiendo del tipo de controlador implementado), accede al modelo, realizando las transformaciones necesarias.

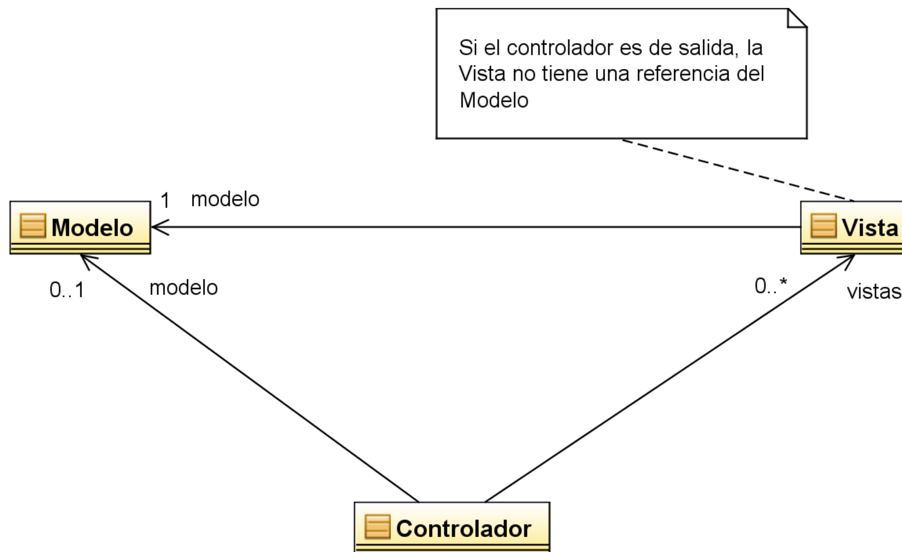

Figura 4.5: Patrón Modelo-Vista-Controlador

4. La vista obtiene el nuevo estado del modelo, y se actualiza. En este sentido, el uso de patrón *Observer* es especialmente útil, ya que permite que las vistas se registren con el modelo, de modo que, ante cualquier cambio en el modelo, las vistas son notificadas. En caso de que se esté haciendo uso de un controlador de salida, el nuevo estado del modelo deberá ser proporcionado a través del controlador

5. La aplicación espera a nuevas interacciones por parte del usuario.

El diagrama de clases del patrón MVC es el representado en la figura 4.5.

4.2.2.2. Patrón *Observer*

Otro patrón de gran utilidad dentro del desarrollo de software es el patrón *Observer*. El patrón *Observer* es utilizado cuando se desea observar el cambio de estado de un objeto de la aplicación. La idea es que todo cambio producido en el modelo de datos sea reflejado de forma inmediata por todas las vistas que estén actualmente visualizando el objeto recién modificado. Este patrón redunda, al igual que el patrón MVC, en una gran independencia entre los datos de la aplicación y su representación gráfica. De hecho, es frecuente que ambos patrones sean usados de forma simultanea.

Java da soporte al patrón *Observer*. El patrón *Observer* de Java está implementado mediante una clase, *Observable*, y una interfaz, *Observer*.

La clase *Observable* es una clase abstracta que representa el modelo de datos que puede ser visualizado de diferentes maneras. Dentro del dominio del problema, el modelo de datos debe extender la clase *Observable*.

La interfaz *Observer* es la interfaz que deben implementar todas aquellas entidades que quieran visualizar gráficamente a un modelo de datos (clase que hereda de *Observable*). Esta interfaz define un método, *update()*, que es invocado cada vez que se notifica un cambio en el modelo de datos (*Observable*).

Así pues, cada vez que se produce un cambio en el estado del *Observable*, éste debe llamar a los métodos *setChanged()* y *notifyObservers()*, para notificar, a todas las vistas registradas en el *Observable*, de que se ha producido un cambio en el objeto observado. Es justo entonces cuando se llama al método *update()* de cada una de las vistas registradas, las cuales, en dicho método, deberán encargarse de actualizar la representación gráfica del modelo de datos acorde a los cambios producidos en él.

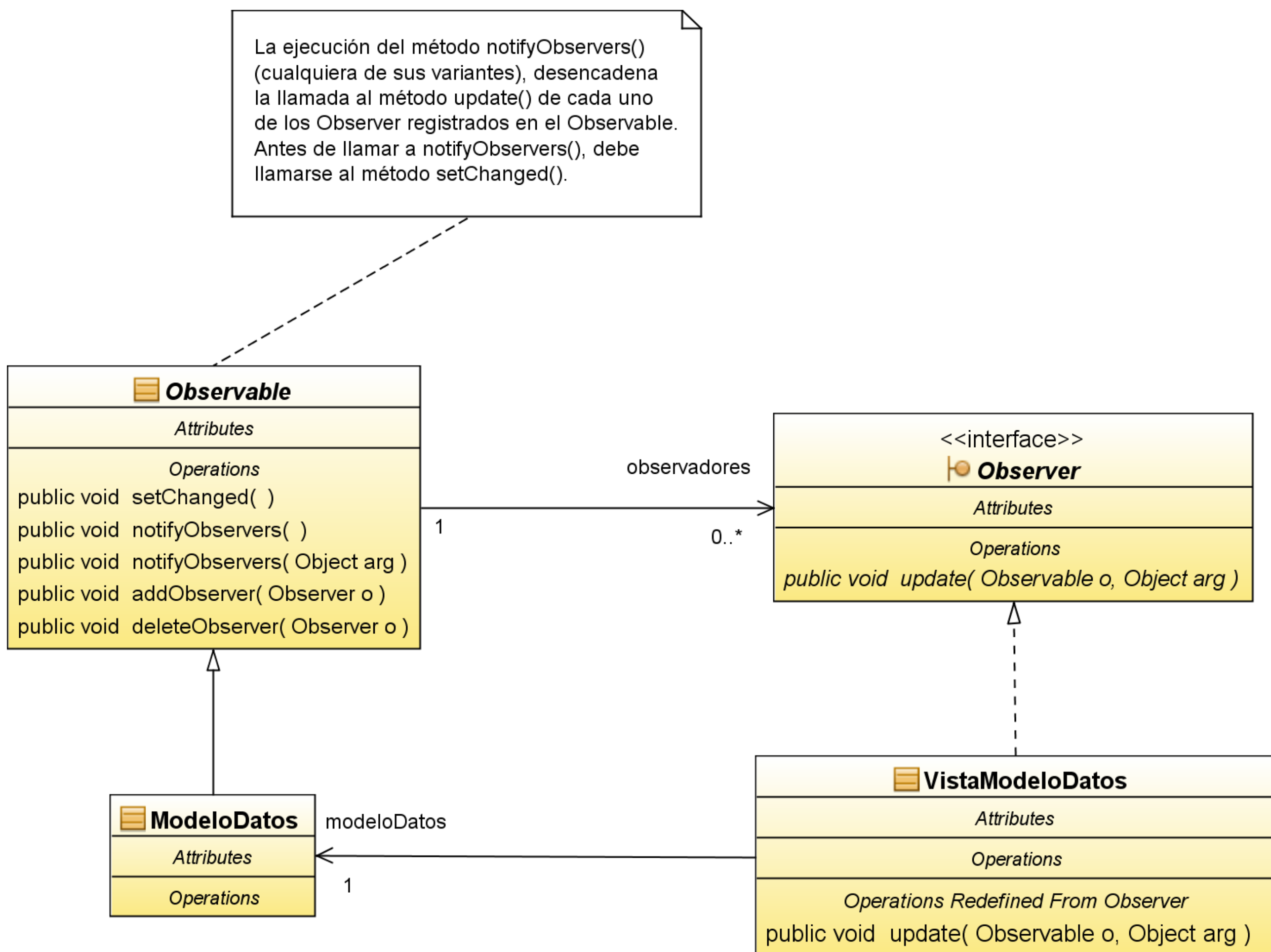

Figura 4.6: Patrón *Observer*

La figura 4.6 representa el diagrama de clases del patrón *Observer* en Java. Se muestra, además, la implementación de dicho patrón a través de dos clases concretas: *ModeloDatos*, siendo el modelo de datos a visualizar, y *Vista*, siendo la clase que representa visualmente el modelo de datos, y que espera ser notificada ante los cambios producidos en éste.

Es importante fijarse en el detalle de que la clase que implementa la interfaz *Observer* consta de una referencia al objeto *Observable* que observa. Realmente, dicha referencia no debería ser necesaria: cada vez que se llama al método *update()*, uno de los argumentos de entrada a dicho método es, justamente, el objeto *Observable*. De este modo, dentro del método *update()*, el *Observer* sería capaz de consultar el estado del objeto observado, para así ver cómo ha cambiado, y poder reflejar acordemente los cambios en su representación gráfica.

Sin embargo, es conveniente que el *Observer* disponga de una referencia al objeto *Observable*. Esto se hace así fundamentalmente para que, en cuanto el *Observer* no necesite reflejar más los cambios del *Observable*, pueda desregistrarse de éste. Una vez desregistrado, el *Observable* ya no notificará al *Observer*.

4.2.2.3. Patrón *Singleton*

El patrón *Singleton* es usado en circunstancias en las que se requiere que exista una única instancia de una clase. Este patrón debe satisfacer dos requerimientos: ofrecer una única instancia de la clase, y permitir el acceso global a dicha instancia. Mucha gente considera que el patrón *Singleton* no es realmente un buen patrón software, ya que introduce variables globales en el sistema. Sin embargo, el patrón *Singleton* es más elegante que la mera declaración de variables globales.

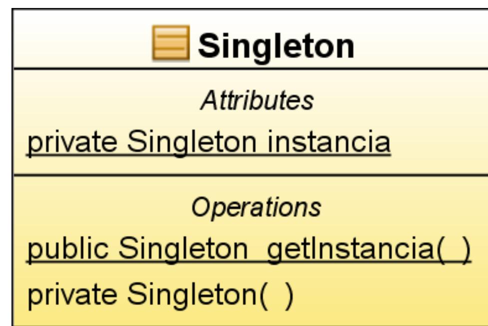

Figura 4.7: Diagrama de clases del patrón *Singleton*

En circunstancias donde realmente es necesaria la existencia de un recurso global compartido por diversas partes del sistema, el patrón *Singleton* desempeña un buen papel.

La implementación del patrón *Singleton* debe por tanto satisfacer dos requerimientos: debe proporcionar una única instancia de la clase, y dicha instacia debe ser globalmente accesible.

El diagrama de clases de la figura 4.7 muestra una posible implementación de este patrón.

La clase de la que existe una única instancia, la clase *Singleton*, almacena una variable de instancia (*instancia*) de tipo *Singleton*, estática y privada. Esta variable estática representa la **única** instancia existente de la clase. El método estático de clase *getInstancia()* se encarga de:

- Inicializar la variable de instancia *instancia* en caso de que no haya sido inicializada. Para ello se hace uso del constructor privado de la clase.

- Devolver la variable de instancia *instancia*.

El constructor de la clase *Singleton* es marcado como privado para evitar que se puedan crear varias instancias de la clase. La única instancia de la clase es creada por el método estático *getInstancia()*, a través del cual, además, se ofrece el acceso **global** a la única instancia de la clase *Singleton*.

Capítulo 5

Storyboard

El *storyboard* de una aplicación es un primer esbozo de su interfaz gráfica. Antes de proceder al desarrollo del aspecto visual de la aplicación, es conveniente hacer un boceto, para así guiar los posteriores desarrollo e implementación.

El *storyboard* (interfaz gráfica) está guiado por los casos de uso del sistema. Debe tenerse en cuenta que, en última instancia, se pretende desarrollar un software capaz de llevar a cabo toda la funcionalidad especificada en los casos de uso. Es por ello que se dice que la interfaz está guiada por los casos de uso, ya que la interfaz ha de ser para el usuario la puerta de acceso a todo aquello que la aplicación es capaz de hacer.

En nuestro caso, la interfaz de usuario del sistema está fuertemente determinada por la arquitectura RCP que hemos decido adoptar. Dentro de un sistema RCP existen *editores* y *vistas*[1], que son los principales componentes de la interfaz gráfica.

De forma general, se puede decir que un editor es el área principal de la aplicación, es decir, la zona donde se focaliza el interés del usuario. Así por ejemplo, en un entorno de programación, el área donde se muestran los ficheros de código fuente es candidata a ser implementada mediante un editor. En una aplicación donde se manipulan imágenes, la zona donde se muestran las imágenes también es candidata a ser implementada mediante un editor. Toda aplicación RCP contiene un único área de editores, en el cual se pueden situar tantos editores como permita el mismo sistema. Dicho área puede ser ocultada (minimizada), pero no cerrada.

Una vista, por otro lado, puede verse como un componente adicional, de apoyo, para realizar ciertas tareas. Por ejemplo, en un entorno de programación, el menú en forma de árbol que suele mostrar la estructura del proyecto actualmente abierto, sería candidato a ser implementado como vista. Las vistas, a diferencia de los editores, permiten situarse en cualquier parte de la ventana de trabajo de la aplicación, y también pueden cerrarse. En este sentido, son muy versátiles, pues permiten que el usuario personalice la interfaz de la aplicación según sus necesidades.

Por tanto, se puede decir que el aspecto gráfico de una aplicación basada en RCP se reduce a un área de editores junto con un repertorio de vistas. La disposición de los elementos gráficos en la ventana es controlada mediante *perspectivas*. Una perspectiva no es más que una descripción de cómo se disponen gráficamente las vistas y el área de editores. Una de las principales utilidades de las perspectivas es poder definir perspectivas de trabajo distintas, orientadas a funciones distintas. Por ejemplo, se podría proporcionar una perspectiva (llámese *perspectiva de tratamiento de imágenes*) que mostrara solamente las vistas relacionadas con el tratamiento digital de imágenes. Además, dicha perspectiva podría reordenar las vistas para que éstas se mostraran de una forma intuitiva al usuario. Si el usuario quisiera tratar digitalmente una serie de imágenes, podría seleccionar la perspectiva de tratamiento de imágenes, la cual se encargaría

[1]Para más información acerca de los editores y vistas de la arquitectura RCP, visitar http://wiki.eclipse.org/FAQ_What_is_the_difference_between_a_view_and_an_editor%3F.

de mostrar todas las vistas relacionadas con la manipulación de imágenes.

Así pues, a la hora de describir la interfaz de usuario de la aplicación, en general será suficiente con describir cada una de las vistas que ofrece. Esta forma de diseñar la interfaz es, cabe decir, muy legible, ya que permite descomponer su diseño en vistas que representan bloques semánticos independientes.

La figura 5.1 representa el aspecto general de la aplicación. Decimos *general* pues, como se ha explicado anteriormente, la posibilidad de reposicionar las vistas dentro de la ventana principal hace que no tenga sentido dibujar una configuración estática de la interfaz.

En dicha figura se puede apreciar que la interfaz está claramente dividida en una serie de componentes básicos:

- Barra de menú: la clásica barra de menú, que ofrece opciones como las de abrir imágenes, guardar imágenes, etc.
- Cool bar: barra de herramientas que muestra una serie de botones con iconos, cuya pulsación desencadena la ejecución de ciertas acciones.
- Selección de perspectivas: un pequeño menú que permite seleccionar las diferentes perspectivas definidas en la aplicación.
- Área de editores: el área central de la ventana, donde se visualizan imágenes e informes de caracterización.
- Espacio reservado para vistas: representa el hecho de que la ventana puede incorporar una gran variedad de vistas, en muy diversas posiciones. De hecho, las vistas de la aplicación podrían distribuirse de formas más complejas que las especificadas en la figura, pero, por brevedad, se ha decidido representarlas así.

5.1. Barra de menú

La barra de menú de la aplicación contiene los siguientes menús:

- Menú *archivo*: contiene acciones genéricas de la aplicación:
    - Abrir imagen: permite cargar una o varias imágenes en la aplicación.
    - Abrir imagen con ROI: permite cargar una o varias imágenes en la aplicación, y si éstas tienen ficheros de región de interés asociados, carga dichas regiones de interés en cada imagen.
    - Abrir informe de caracterización: permite cargar uno o varios informes de caracterización en la aplicación.
    - Guardar: permite guardar el contenido del editor actualmente seleccionado, en el archivo asociado a dicho editor. Si el editor es, por ejemplo, el editor de una imagen, esta acción permite guardar la imagen en su fichero asociado.
    - Guardar como: permite guardar el contenido del editor actualmente seleccionado en un archivo seleccionado por el usuario. Si el editor es, por ejemplo, el editor de una imagen, esta acción permite especificar un archivo donde guardar la imagen del editor.
    - Salir: cierra la aplicación.

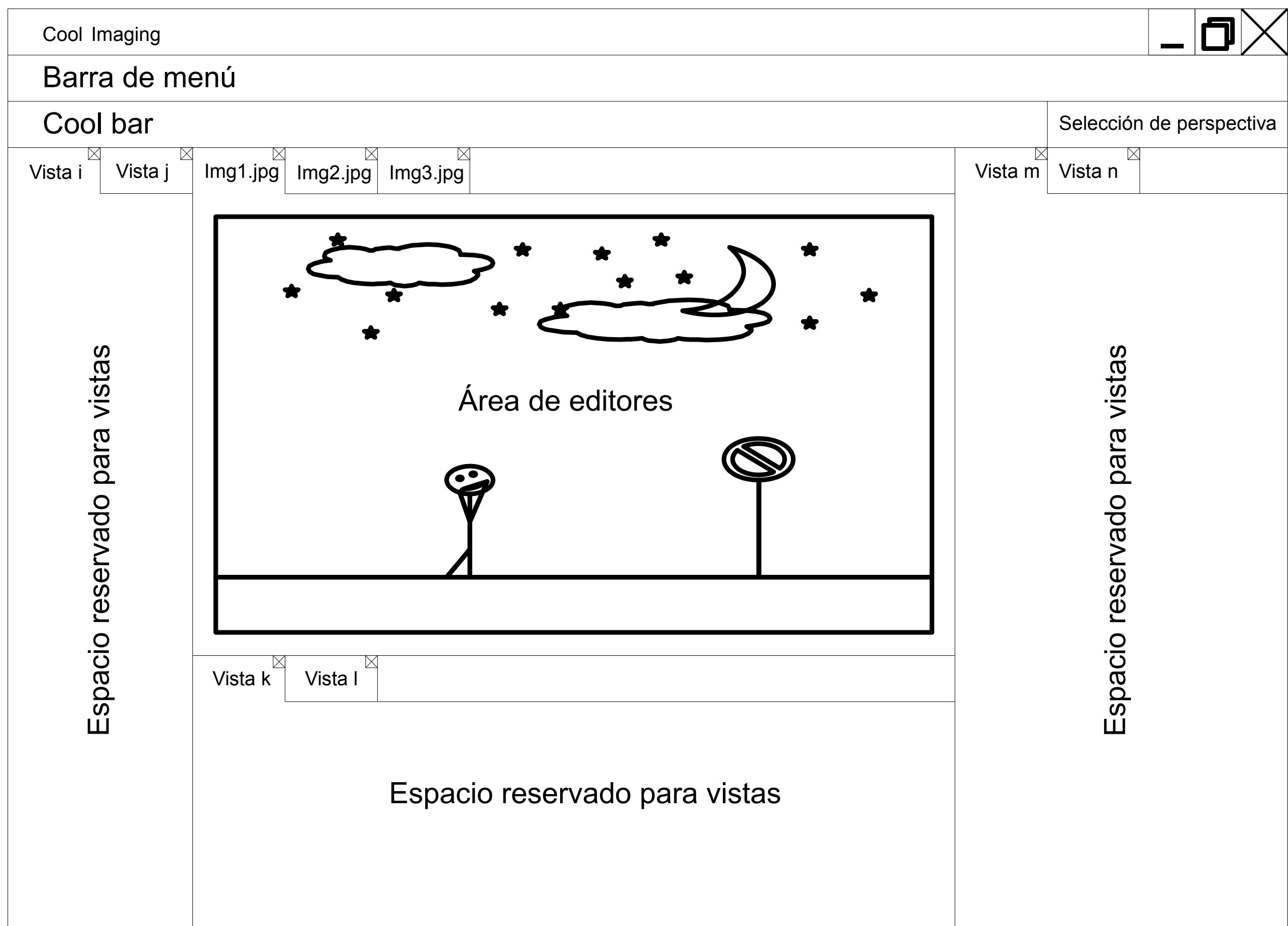

Figura 5.1: Aspecto general de la aplicación

- Menú *operaciones tratamiento imágenes*: es un menú que permite seleccionar las operaciones de tratamiento de imágenes definidas en la aplicación. Este menú está a su vez dividido en submenús que agrupan conjuntos de operaciones de semántica parecida.

- Menú *herramientas*: contiene un único elemento, el cual permite acceder a la página de preferencias de la aplicación.

- Menú *ver*: contiene dos elementos:

  - Mostrar vista: permite abrir cualquier vista definida en la aplicación.
  - Cambiar perspectiva: permite cambiar de perspectiva.

- Menu *Ayuda*: define tres elementos:

  - Mostrar ayuda: muestra la ayuda integrada en la aplicación, en forma de diálogo emergente.
  - Buscar ayuda: muestra una vista que permite realizar búsquedas sobre la ayuda integrada en la aplicación.
  - Ayuda dinámica: activa la vista de la ayuda dinámica de Eclipse.
  - Acerca de: muestra el diálogo *acerca de* de la aplicación.

Un aspecto importante de una aplicación RCP es la capacidad de modificar la estructura de la barra de menú (insertar nuevos menús de alto nivel o añadir submenús a otros menús ya existentes) de forma dinámica, dependiendo de los *plugins* instalados en la aplicación. Además, la arquitectura RCP ofrece facilidades para modificar la estructura de los menús dependiendo del editor o perspectiva seleccionados actualmente. Por ejemplo, si el editor seleccionado fuera un editor de imágenes, se podrían añadir a la barra de menú acciones asociadas a la manipulación de imágenes, como acciones que permitieran definir la región de interés de la imagen.

5.2. Cool bar

La cool bar contiene una serie de botones con iconos, que desencadenan la ejecución de ciertas acciones en la aplicación. Los botones que contiene son los siguientes:

- Abrir imagen: permite cargar una o varias imágenes dentro de la aplicación.

- Abrir imagen con ROI: permite cargar una o varias imágenes dentro de la aplicación. Además, si dichas imágenes tienen ficheros de región de interés (ROI) asociados, las regiones de interés también serán cargadas en la aplicación.

- Guardar: permite guardar el contenido del editor actualmente seleccionado, en el archivo asociado a dicho editor. Si el editor es, por ejemplo, el editor de una imagen, esta acción permite guardar la imagen en su fichero asociado.

La cool bar de una aplicación RCP muestra el mismo comportamiento dinámico que la barra de menú: puede ser modificada dinámicamente según los *plugins* que hay instalados en la aplicación. Además, dependiendo del editor y de la perspectiva seleccionados, puede también modificarse las acciones que se muestran el la cool bar. Por ejemplo, si el editor seleccionado es un editor de imágenes, la cool bar podría mostrar acciones relacionadas con la manipulación de la ROI de la imagen.

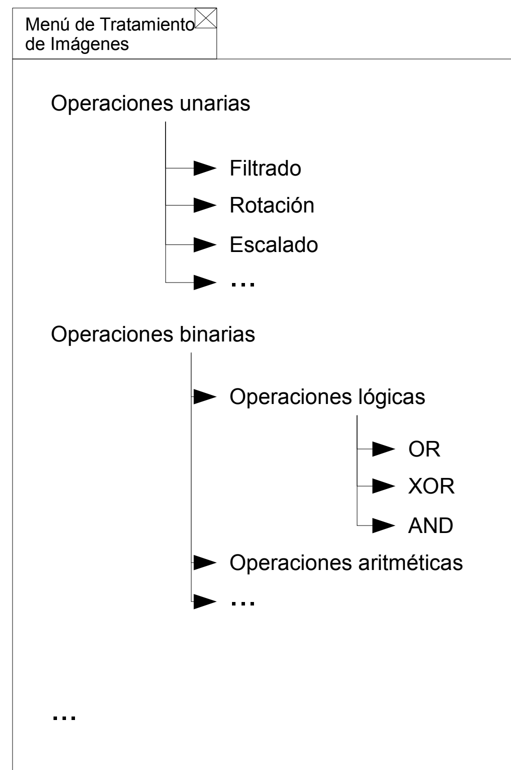

Figura 5.2: Vista Menú de Tratamiento de Imagenes

5.3. Vistas

5.3.1. Menú de Tratamiento de Imágenes

La vista Menú de Tratamiento de Imágenes representa un panel gráfico que muestra todas las operaciones de tratamiento de imágenes de la aplicación.

Dichas operaciones se muestran en una estructura de árbol desplegable, en el que las operaciones se encuentran agrupadas por categorías, y cuyas hojas representan las operaciones en sí. Su aspecto es el mostrado en la figura 5.2.

Cuando el usuario hace doble click en una de las operaciones mostradas, se muestra un diálogo emergente (PanelOperacionTratamiento) en el que el usuario puede introducir los datos de la operación a aplicar y aplicarla de forma directa a una o varias imágenes de entrada, o bien insertarla en una determinada cadena de operaciones. Su aspecto es el mostrado en la figura 5.3. El TabularPanelOperacionTratamiento representa un panel con dos pestañas.

La pestaña *Uso directo* permite al usuario usar de forma inmediata la operación, es decir, seleccionar las imágenes de entrada de la operación, y aplicar la operación sobre dichas imágenes de entrada. La imagen resultado puede bien sobreescribir la imagen actualmente seleccionada en la aplicación, o bien crear una nueva imagen.

La pestaña *Insertar en cadena* permite al usuario seleccionar una o varias cadenas de operaciones e insertar la operación en dichas cadenas de operaciones.

El aspecto del TabularPanelOperacionTratamiento es el mostrado en la figura 5.4.

Dentro de la pestaña de *Uso directo*, la tabla de selección de imágenes (TablaSeleccionImagenes) es la tabla que muestra las imágenes que requiere la operación para ejecutarse, y que permite al usuario elegirlas. Su aspecto es el de la figura 5.5.

El panel de selección del destino de la imagen resultado permite especificar si la imagen resultado sobreescribirá a la imagen actualmente seleccionada en la aplicación o si bien se creará una nueva.

Figura 5.3: PanelOperacionTratamiento

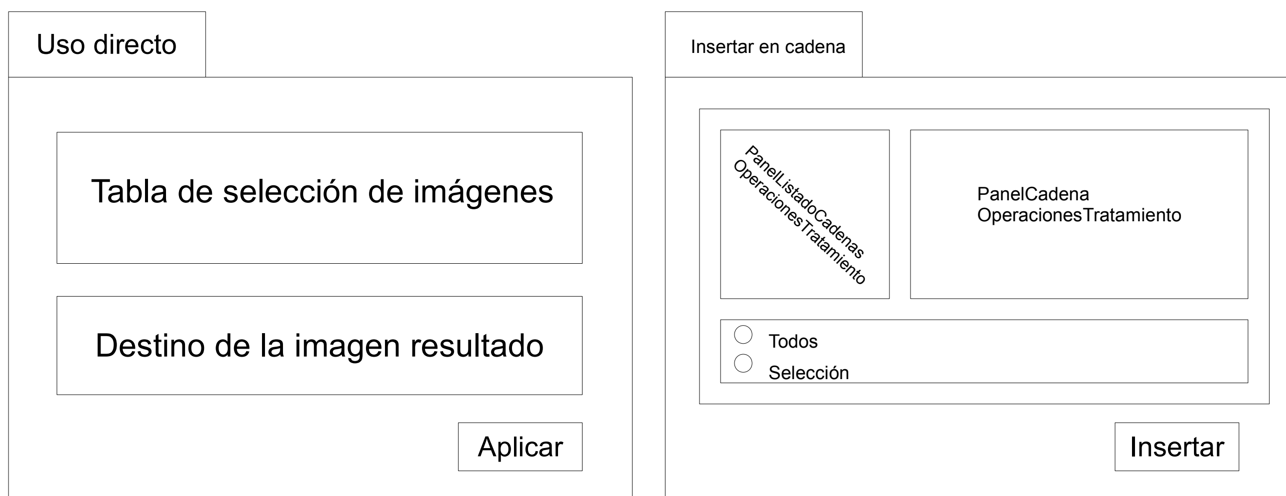

Figura 5.4: TabularPanelOperacionTratamiento

Descripción	Nombre de imagen
Imagen a rotar	Img1.jpg

Figura 5.5: TablaSeleccionImagenes

Cadena de operaciones 1
Cadena de operaciones 2
Cadena de operaciones 3

Crear cadena de operaciones
Renombrar cadena de operaciones
Seleccionar todos
Deseleccionar todos
Eliminar cadenas seleccionadas
Guardar cadena de operaciones

Figura 5.6: PanelListadoCadenasOperacionesTratamiento (izquierda), y su menú contextual (derecha)

Nombre	Descripción
Operación 1	Descripción 1
Operación 2	Descripción 2
Operación 3	Descripción 3
Operación 4	Descripción 4
Operación 5	Descripción 5

Seleccionar todos
Deseleccionar todos
Eliminar operaciónes seleccionadas

Figura 5.7: PanelCadenaOperacionesTratamiento (izquierda), y su menú contextual (derecha)

Dentro de la pestaña *Insertar en cadena*, el PanelListadoCadenasOperacionesTratamiento es un panel que muestra un listado de todas las cadenas de operaciones de tratamiento de imágenes de la aplicación. El listado muestra el nombre de las cadenas de operaciones. Este panel, además, proporciona un menú contextual que permite al usuario crear, eliminar, renombrar y almacenar en disco cadenas de operaciones. Su aspecto es el de la figura 5.6.

La opción del menú contextual que permite crear una cadena de operaciones muestra un diálogo emergente donde el usuario introduce el nombre de la cadena de operaciones. El nombre no puede ser vacío y además no debe coincidir con el nombre de ninguna otra cadena de operaciones existente en la aplicación. La opción que permite renombrar una cadena de operaciones es análoga, mostrando un diálogo donde el usuario introduce en nuevo nombre, que no puede ser vacío y que no debe coincidir con el de otra cadena de operaciones existente. La acción que permite guardar una cadena de operaciones muestra un diálogo donde seleccionar el nombre del archivo en el que guardar la cadena de operaciones.

El PanelCadenaOperacionesTratamiento es un panel que muestra un listado de todas las operaciones de una cadena de operaciones. Este panel, además, ofrece un menú contextual que permite al usuario eliminar operaciones de la cadena de operaciones. En el caso de la pestaña *Insertar en cadena*, además, en este panel se muestra las operaciones de la cadena seleccionada en el PanelListadoCadenasOperacionesTratamiento de la izquierda. El aspecto visual del PanelCadenaOperacionesTratamiento es el mostrado en la figura 5.7.

La vista Menú de Tratamiento de Imágenes está estrechamente relacionada con la vista Panel de Operación de Tratamiento. Como hemos explicado, cuando el usuario hace doble click en una de las operaciones del menú, se muestra, en un diálogo emergente que contiene el PanelOperacionTratamiento. Por otro lado, además, cuando el usuario selecciona una de las operaciones en el árbol, el PanelOperacionTratamiento se muestra en la vista Panel de Operación de Tratamiento (ver sección 5.3.2).

5.3.2. Panel de Operación de Tratamiento

La vista Panel de Operación de Tratamiento es una simple vista que muestra al usuario el PanelOperacionTratamiento de la operación seleccionada actualmente en la vista Menú de Tratamiento de Imágenes. El Panel de Operación de Tratamiento *tiene memoria*, es decir, es capaz de recordar los parámetros introducidos en cada uno de los PanelOperacionTratamiento que ha ido almacenando. Así, si el usuario selecciona distintas operaciones, aunque los paneles

Figura 5.8: Vista Paquetes de Imagenes

mostrados vayan cambiando, van conservando los valores que el usuario hubiera introducido.

5.3.3. Paquetes de Imágenes

La vista Paquetes de Imágenes permite gestionar la creación de paquetes de imágenes en la aplicación. Un paquete de imágenes, recuérdese, no es más que un listado de ficheros de imágenes, identificado por un nombre.

El aspecto gráfico de esta vista es el mostrado en la figura 5.8.

La vista se divide principalmente en dos regiones: la región llamada PanelPaqueteGlobal, y el resto. En este PanelPaqueteGlobal el usuario puede insertar imágenes. Digamos que este panel define un *paquete global* dentro de la vista. Este panel permite visualizar dicho paquete de imágenes, además, de dos maneras: de forma textual o mediante imágenes. Cuando se visualiza de forma textual, el panel muestra simplemente un panel igual al PanelPaqueteImagenes de la misma vista. Un PanelPaqueteImagenes muestra un listado de ficheros de tipo imagen, y proporciona al usuario un menú contextual que le permite eliminar las imágenes que haya seleccionadas en el paquete. Este panel, además, permite *arrastrar*[2] imágenes desde él mismo y soltarlas en otra zona de la aplicación. Además, permite que sobre él suelten imágenes, las cuales son añadidas al mismo paquete de imágenes. Su aspecto es el que se muestra en la figura

[2]Mediante el mecanismo de *drag and drop*.

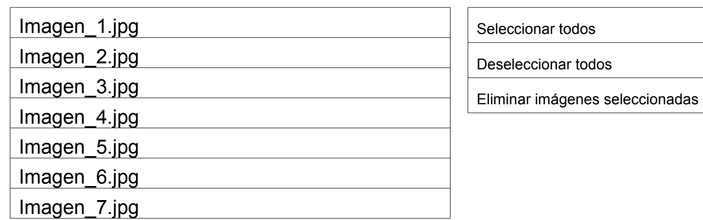

Figura 5.9: PanelPaqueteImagenes (izquierda), y su menú contextual (derecha)

Figura 5.10: PanelThumbnails (izquierda), y su menú contextual (derecha)

5.9.

El PanelPaqueteGlobal, además, permite visualizar sus imágenes en forma de *thumbnail*. Cuando éste es el caso, se muestra un panel gráfico (PanelThumbnails) con los thumbnails de las imágenes, organizados en forma de matriz. El panel permite al usuario, además, establecer tanto la disposición de los thumbnails del paquete como su tamaño. Su aspecto es el que se muestra en la figura 5.10.

El usuario puede insertar imágenes en el PanelPaqueteGlobal de diversas maneras. Si el panel está mostrando las imágenes como una lista de texto (PanelPaqueteImagenes), se puede hacer simplemente uso del mecanismo de *drag and drop* para insertar en él imágenes. En cualquier caso, la cool bar de la vista así como el menú desplegable de la vista permiten, mediante diversas opciones, insertar y eliminar imágenes de la zona del PanelPaqueteGlobal. Posteriormente pasaremos a describir dichos menús.

La región que no forma parte del PanelPaqueteGlobal está encargada de la creación, eliminación y modificación de los paquetes de imágenes globales a la aplicación. El PanelListadoPaquetesImagenes muestra un listado de todos los paquetes de imágenes globales a la aplicación. El PanelPaqueteImagenes situado a su derecha muestra las imágenes del paquete que se seleccione en el PanelListadoPaquetesImagenes. El PanelListadoPaquetesImagenes no es más que un simple listado que, también, dispone de un menú contextual que permite interaccionar con los paquetes de imágenes. Este menú permite, entre otras cosas, crear paquetes, eliminar paquetes, renombrar paquetes y guardar paquetes en ficheros de disco. Además, permite el mecanismo de *drag and drop* de paquetes de imágenes. Su aspecto es el de la figura 5.11.

La opción que permite crear un nuevo paquete de imágenes muestra un diálogo emergente que pide al usuario el nombre del paquete de imágenes a crear. El usuario debe introducir un nombre, y además éste no puede coincidir con el de ningún otro paquete ya existente.

PaqueteImagenes1
PaqueteImagenes2
PaqueteImagenes3

Crear paquete de imágenes
Renombrar paquete
Seleccionar todos
Deseleccionar todos
Eliminar paquetes seleccionados
Guardar paquete de imágenes

Figura 5.11: PanelListadoPaqueteImagenes (izquierda), y su menú contextual (derecha)

Igualmente, la opción de renombrar un paquete de imágenes muestra otro diálogo al usuario, el cual lo obliga a introducir un nombre y que, además, dicho nombre no coincida con el de ningún otro paquete existente. La opción de guardar el paquete de imágenes muestra al usuario un diálogo de archivo donde introducir el nombre del fichero en el que almacenar el paquete de imágenes.

Los demás controles gráficos de la vista Paquetes de Imágenes se encargan también de la gestión de los paquetes de imágenes.

El subpanel de creación de paquetes permite crear paquetes de imágenes. Proporciona tres opciones de creación de paquetes: vacío, que permite crear un paquete de imágenes inicialmente vacío; completo, que crea un paquete de imágenes con todas las imágenes cargadas en el PanelPaqueteGlobal; selección, que crea un paquete con todas las imágenes que están seleccionadas en el PanelPaqueteGlobal. En cualquier caso, la aplicación muestra un diálogo en el que el usuario debe introducir el nombre del paquete. El nombre no puede coincidir el de ningún otro paquete de imágenes ya creado.

El subpanel de eliminación de paquetes permite eliminar paquetes de imágenes. Ofrece dos opciones: todos, que elimina todos los paquetes de imágenes de la aplicación; selección, que permite eliminar los paquetes que están seleccionados en el PanelListadoPaquetesImagenes.

El subpanel de transferencia de imágenes permite transferir imágenes del PanelPaqueteGlobal a los paquetes de imágenes creados. Este subpanel permite especificar qué imágenes se transfieren, y a qué paquetes son transferidas. El panel de imágenes de origen permite seleccionar, como imágenes a transferir, tanto todas las que haya en el PanelPaqueteGlobal, como sólo las que hubiera seleccionadas. El panel de paquetes de destino permite indicar a qué paquetes se transferirán las imágenes de origen, que pueden ser, o bien todos, o bien los seleccionados en el PanelListadoPaquetesImagenes.

La cool bar de la vista Paquetes de Imágenes permite llevar a cabo diversas acciones. Este cool bar muestra una serie de botones que desencadenan la ejecución de dichas acciones:

- Activar modo thumbnail: hace que el PanelPaqueteGlobal visualice las imágenes mediante thumbnails.
- Activar modo texto: hace que el PanelPaqueteGlobal visualice las imágenes mediante un PanelPaqueteImagenes.
- Insertar imágenes del directorio de trabajo: inserta, en el PanelPaqueteGlobal, las imágenes existentes en el directorio de trabajo.
- Insertar imágenes abiertas: inserta, en el PanelPaqueteGlobal, las imágenes abiertas actualmente en la aplicación.
- Insertar imágenes desde directorio: abre un diálogo emergente que permite al usuario seleccionar imágenes desde el almacenamiento local. Las imágenes seleccionadas serán insertadas en el PanelPaqueteGlobal.
- Seleccionar todas las imágenes: selecciona todas las imágenes del PanelPaqueteGlobal.

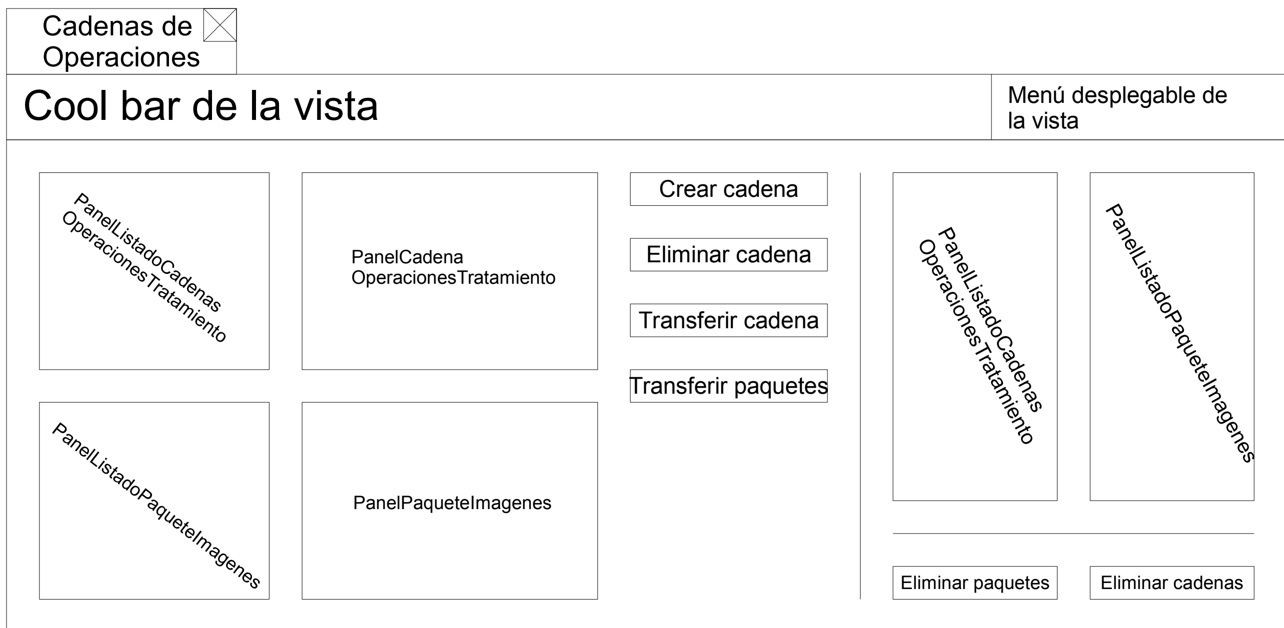

Figura 5.12: Vista Cadenas de Operaciones

- Deseleccionar todas las imágenes: deselecciona todas las imágenes del PanelPaqueteGlobal.
- Maximizar/minimizar región de imágenes: maximiza la región del PanelPaqueteGlobal si no está maximizada. Si está maximizada, restaura su tamaño.
- Maximizar/minimizar región de funciones de paquetes de imágenes: maximiza la región ajena al PanelPaqueteGlobal si no está maximizada. Si está maximizada, restaura su tamaño.
- Eliminar todas las imágenes: elimina todas las imágenes del PanelPaqueteGlobal.
- Eliminar imágenes seleccionadas: elimina las imágenes seleccionadas del PanelPaquete-Global.

El menú desplegable de la vista contiene todas las opciones de la cool bar.

5.3.4. Cadenas de Operaciones

La vista Cadenas de Operaciones permite al usuario definir cadenas de operaciones y aplicarlas sobre paquetes de imágenes. Recuérdese que una cadena de operaciones no es más que una secuencia ordenada de operaciones, agrupadas bajo un nombre (identificador). Esta vista permite visualizar tanto las cadenas de operaciones definidas en la aplicación como los paquetes de imágenes que el usuario haya podido crear. Además, el usuario puede seleccionar cadenas de operaciones y paquetes de imágenes, de modo que los paquetes de imágenes podrán ser procesados por las cadenas de operaciones. El aspecto de esta vista es el mostrado en la figura 5.12.

La vista se divide en dos regiones, a saber, la situada a la izquierda y la situada a la derecha.

La región situada a la izquierda muestra un listado de todas las cadenas de operaciones que hay actualmente creadas en la aplicación (el PanelListadoCadenasOperacionesTratamiento muestra el listado en cuestión, y el PanelCadenaOperacionesTratamiento muestra las operaciones de la cadena seleccionada). Además, se muestra un listado de todos los paquetes de

Figura 5.13: PanelConfiguracionProcesamientoPaquetes

imágenes definidos en la aplicación (el PanelListadoPaqueteImagenes muestra el listado de los paquetes, y el PanelPaqueteImagenes muestra las imágenes del paquete seleccionado). El botón *Crear cadena* permite crear una nueva cadena de operaciones. Para ello, se le muestra al usuario un diálogo emergente en el que simplemente introduce el nombre de la cadena a crear. El botón *Eliminar cadena* permite eliminar las cadenas de operaciones seleccionadas. El botón *Transferir cadenas* transfiere las cadenas de operaciones seleccionadas a la región de la derecha de la vista. El botón *Transferir paquetes* transfiere los paquetes de imágenes seleccionados a la región de la derecha de la vista.

La región de la vista situada a la derecha representa los paquetes de imágenes a procesar así como las cadenas de operaciones con las que procesar dichos paquetes. Cuando el usuario añade paquetes o cadenas de operaciones a esta región, el usuario podrá procesar los paquetes seleccionados con las cadenas de operaciones seleccionadas. Dentro de esta región, el los botones *Eliminar cadenas* y *Eliminar paquetes* permiten justamente eliminar cadenas de operaciones y paquetes de imágenes de esta región.

La cool bar de la vista contiene dos acciones. Una de ellas permite cargar cadenas de operaciones previamente guardadas en ficheros de disco. La otra permite procesar los paquetes de imágenes con las cadenas de operaciones seleccionadas. El menú desplegable contiene ambas acciones.

Cuando se hace uso de la acción que permite procesar paquetes de imágenes, al usuario se le muestra un diálogo como el de la figura 5.13 (PanelConfiguracionProcesamientoPaquetes).

Este diálogo permite al usuario especificar los últimos detalles relativos al procesamiento de los paquetes de imágenes, a saber:

- *Patrón*: representa una cadena que se le añadirá al principio de los nombres de todos los ficheros generados. Por ejemplo, si el patrón es *miPatron*, y una de las imágenes procesadas se llama *imagen1.jpg*, la correspondiente salida sería *miPatronimagen1.jpg*.

- Formato de los ficheros de salida: el usuario puede elegir mantener el formato original de las imágenes procesadas, o bien especificar un formato común para todas las imágenes generadas.

- Directorio de salida: el usuario especifica el directorio donde se almacenarán las imágenes procesadas. Dentro de este directorio se almacenan dichas imágenes, organizadas a su vez en carpetas que las distinguen por nombre de paquete de imágenes y por nombre de cadena de operaciones aplicada.

- Especificar región de interés (ROI): permite al usuario especificar una ROI a considerar a la hora de procesar las imágenes. Al usuario se le dan tres opciones:

  - Especificar un fichero de ROI: el usuario selecciona un fichero de ROI que será común a todas las imágenes procesadas.
  - Usar la ROI por defecto: para cada imagen procesada, se usará la ROI por defecto. La ROI por defecto es la situada en un archivo de mismo nombre que el de la imagen (sin extensión), y acabado en «.roi».
  - No usar ROI: no se usará ROI a la hora de procesar las imágenes.

Cuando el usuario pulsa el botón *Procesar*, todos los paquetes de imágenes que se situaron en la región derecha de la vista son procesados por todas las cadenas de operaciones situadas en la región derecha de la vista.

5.3.5. Menú de Caracterización de Imágenes

La vista Menú de Caracterización de Imágenes representa un panel gráfico que muestra todas las operaciones de caracterización de imágenes de la aplicación.

Dichas operaciones se muestran en una estructura de árbol desplegable, en el que las operaciones se encuentran agrupadas por categorías, y cuyas hojas representan las operaciones en sí. Su aspecto es el mostrado en la figura 5.14.

Cuando el usuario hace doble click en una de las operaciones mostradas, se muestra un diálogo emergente (PanelOperacionCaracterizacion) en el que el usuario puede introducir los datos de la operación de caracterización a aplicar. Además. el usuario puede determinar si quiere aplicar dicha operación de caracterización de forma directa sobre una imagen, o si bien quiere insertar dicha operación de caracterización en un generador de vector de caracterización. Su aspecto es el mostrado en la figura 5.15.

El panel *TabularPanelOperacionCaracterizacion* es el elemento tabular que permite al usuario, bien aplicar de de forma directa la operación de caracterización, o bien insertarla en un generador de vector de caracterización. Su aspecto es el de la figura 5.16.

La pestaña *Uso directo* permite al usuario aplicar de forma directa la operación de caracterización sobre una imagen abierta en la aplicación. Para ello, al usuario se le muestra una tabla donde puede elegir la imagen a la que aplicar la operación. El botón *Aplicar* desencadena la operación, y como resultado, se genera un informe de caracterización que contiene el resultado de caracterizar la imagen seleccionada mediante la operación aplicada.

La pestaña *Insertar en generador* permite al usuario insertar la operación en uno o varios generadores de vector de caracterización. Dicho panel permite seleccionar aquellos generadores de vector de caracterización en los que insertar la operación de caracterización elegida. El panel, como se aprecia, permite insertar dicha operación tanto en todos los generadores de vector de caracterización de la aplicación como sólo en los que haya seleccionados actualmente en el PanelListadoGeneradoresVC.

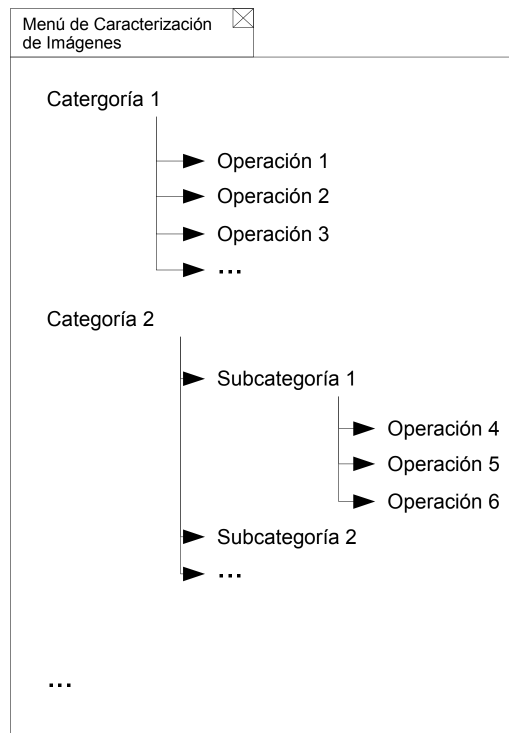

Figura 5.14: Vista Menú de Caracterización de Imagenes

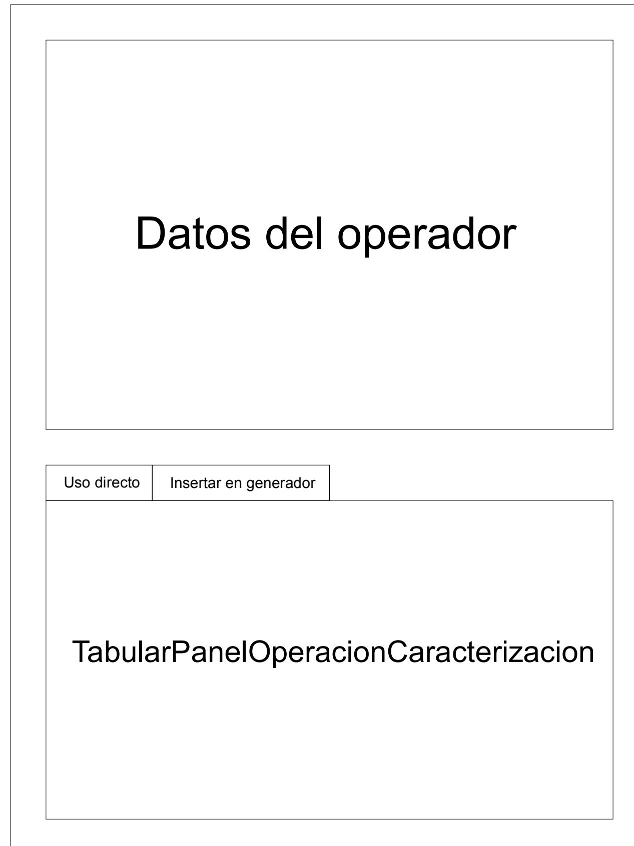

Figura 5.15: PanelOperacionCaracterizacion

Figura 5.16: TabularPanelOperacionCaracterizacion

Figura 5.17: PanelListadoGeneradoresVC (izquierda), y su menú contextual (derecha)

El PanelListadoGeneradoresVC es un panel gráfico que muestra el listado de los generadores de VC de la aplicación. Dicho panel dispone de un menú contextual que permite crear generadores de VC, renombrar los existentes y eliminar los seleccionados. Además, implementa el mecanismo de *drag and drop*. Su aspecto es el reflejado en la figura 5.17.

Por otro lado, el PanelGeneradorVC es un panel que permite visualizar las operaciones de caracterización que hay almacenadas dentro del GeneradorVC seleccionado en el PanelListado-GeneradorVC. Dichas operaciones son visualizadas en forma de lista. Además, el PanelGeneradorVC proporciona un menú contextual que permite eliminar operaciones del generador de VC. Su aspecto es el representado en la figura 5.18.

La vista Menú de Caracterización de Imágenes está estrechamente relacionada con la vista Panel de Operación de Caracterización. Como hemos explicado, cuando el usuario hace doble click en una de las operaciones del menú, se muestra, en un diálogo emergente que contiene el PanelOperacionCaracterización. Por otro lado, además, cuando el usuario selecciona una de las operaciones en el árbol, el PanelOperacionCaracterización se muestra en la vista Panel de Operación de Caracterización (ver sección 5.3.6).

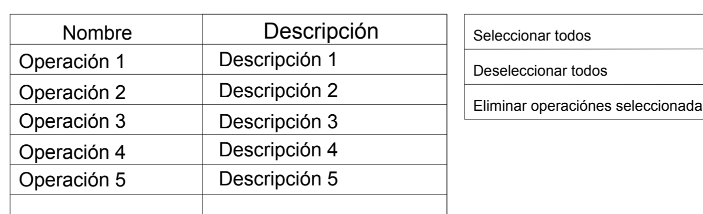

Figura 5.18: PanelGeneradorVC (izquieda), y su menú contextual (derecha)

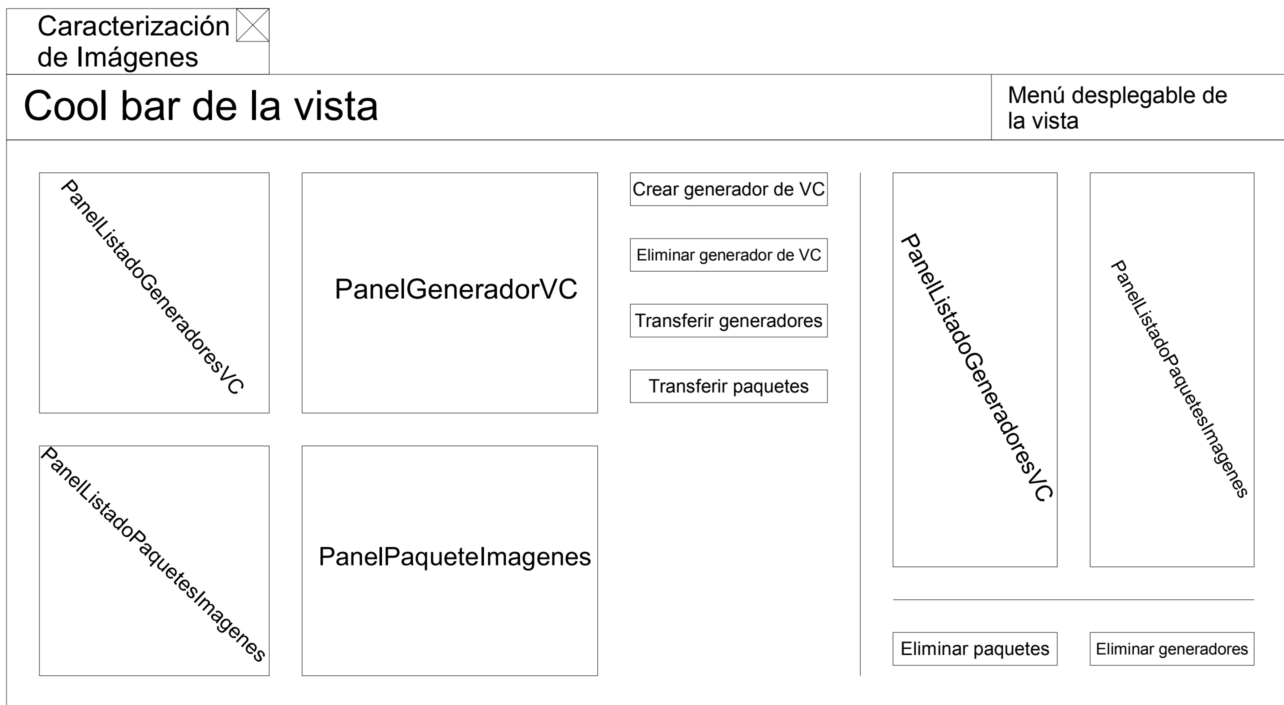

Figura 5.19: Vista Caracterización de Imágenes en modo de procesamiento de paquetes de imágenes

5.3.6. Panel de Operación de Caracterización

La vista Panel de Operación de Caracterización es una simple vista que muestra al usuario el PanelOperacionCaracterización de la operación seleccionada actualmente en la vista Menú de Caracterización de Imágenes. El Panel de Operación de Caracterización *tiene memoria*, es decir, es capaz de recordar los parámetros introducidos en cada uno de los PanelOperacionCaracterización que ha ido almacenando. Así, si el usuario selecciona distintas operaciones, aunque los paneles mostrados vayan cambiando, van conservando los valores que el usuario hubiera introducido.

5.3.7. Caracterización de Imágenes

La vista Caracterización de Imágenes es la vista que permite al usuario planificar y ejecutar la caracterización de imágenes.

Esta vista permite al usuario definir los *generadores de vectores de caracterización* de la aplicación, es decir, crearlos y eliminarlos. Recuérdese que, para caracterizar imágenes, el usuario debe definir objetos de tipo *generador de vector de caracterización*. Un *generador de vector de caracterización* no es más que un repositorio de operaciones de caracterización, las cuales podrán ser usadas para caracterizar una o varias imágenes (y, por tanto, generar vectores de caracterización asociados a las imágenes).

Esta vista dispone de dos métodos de funcionamiento: uno permite caracterizar paquetes de imágenes, mientras que el otro permite caracterizar imágenes de forma directa. Ambos modos de funcionamiento pueden establecerse a través de las acciones disponibles en la cool bar de la vista, y su aspecto es el mostrado en las figuras 5.19 y 5.20.

Cuando la vista está funcionando en el modo de procesamiento de paquetes de imágenes, muestra el aspecto de la figura 5.19.

La vista se divide en dos partes bien diferenciadas. La parte de la izquierda muestra un listado de los generadores de vector de caracterización globales de la aplicación, así como un

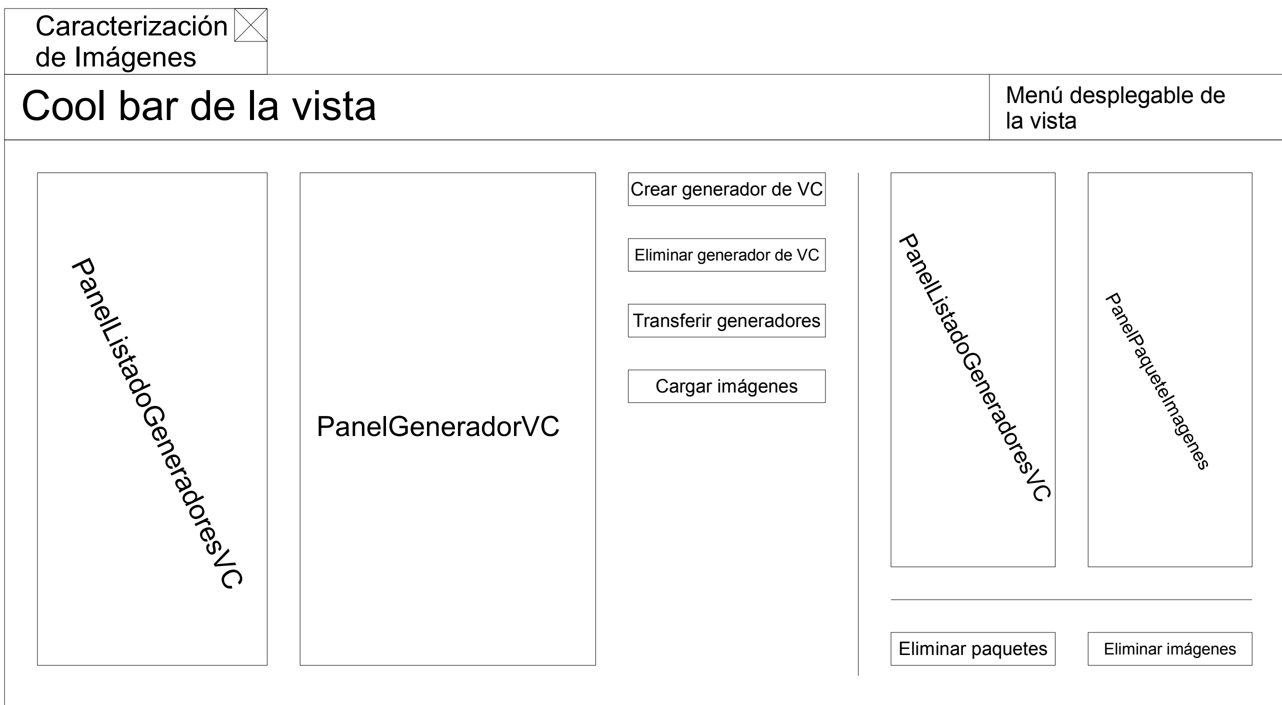

Figura 5.20: Vista Caracterización de Imágenes en modo de procesamiento de imágenes

listado de los paquetes de imágenes globales del aplicación. La parte de la derecha muestra un listado de los generadores a ser usados en la caracterización, así como los paquetes a ser caracterizados.

En la parte de la izquierda el botón *Crear generador de VC* permite crear un nuevo generador de vector de caracterización. Dicho botón lanza un diálogo en el que el usuario puede introducir el nombre del nuevo generador a crear. El nombre no puede ser vacío, y no puede coincidir con el nombre de otro generador existente en la aplicación. El botón *Eliminar generador de VC* permite eliminar los generadores seleccionados en el PanelListadoGeneradoresVC de la izquierda. Dicho PanelListadoGeneradoresVC muestra todos los generadores presentes en la aplicación. El botón *Transferir generadores* transfiere los generadores seleccionados en el listado de los generadores globales de la aplicación, al listado de generadores a usar en la caracterización. El botón *Transferir paquetes* transfiere los paquetes de imágenes del listado de paquetes de la izquierda (listado de paquetes de imágenes globales de la aplicación) al listado de la derecha, es decir, al listado de paquetes a usar en la caracterización.

La región de la derecha, aparte de los listados de los generadores y los paquetes a usar en la caracterización, muestra dos botones, el botón *Eliminar generadores*, que permite eliminar los generadores seleccionados en la región de la derecha, y el botón *Eliminar paquetees*, que permite eliminar los paquetes de imágenes seleccionados en la región de la derecha.

Cuando la vista está funcionando en el modo de procesamiento de imágenes muestra el aspecto de la figura 5.20. Su aspecto es bastante similar al que muestra cuando funciona en el modo de procesamiento de imágenes, con la diferencia de que no permite operar con paquetes de imágenes, sino con imágenes de forma directa. El botón *Cargar imágenes* abre un diálogo donde el usuario puede seleccionar imágenes a cargar en el listado de imágenes a caracterizar.

Tanto el cool bar de la vista como su menú desplegable muestran las siguientes acciones:

- Activar modo caracterización imágenes: activa el modo de operación que permite caracterizar imágenes.

- Activar modo caracterización paquetes imágenes: activa el modo de operación que permite caracterizar paquetes de imágenes.

Figura 5.21: PanelConfiguracionCaracterizacion

- Cargar generadores de VC: permite cargar de disco generadores de vector de caracterización. Esta acción lanza un diálogo en el que el usuario selecciona los ficheros que contienen los generadores a cargar.
- Procesar: lanza un diálogo de configuración de la caracterización, a través del cual se puede desencadenar la caracterización de imágenes.

La acción Procesar lanza un diálogo como el de la figura 5.21 (PanelConfiguracionCaracterizacion), en el que el usuario puede especificar las opciones de la caracterización a llevar a cabo, a saber:

- Especificar región de interés (ROI): permite al usuario especificar una ROI a considerar a la hora de caracterizar las imágenes. Al usuario se le dan tres opciones:
    - Especificar un fichero de ROI: el usuario selecciona un fichero de ROI que será común a todas las imágenes caracterizadas.
    - Usar la ROI por defecto: para cada imagen caracterizada, se usará la ROI por defecto. La ROI por defecto es la situada en un archivo de mismo nombre que el de la imagen (sin extensión), y acabado en «.roi».
    - No usar ROI: no se usará ROI a la hora de caracterizar las imágenes.
- Especificar si calcular un resumen del informe: cuando se activa esta opción, se crearán medidas de resumen del informe de caracterización. Como consecuencia, se calcularán medias y desviaciones típicas de los vectores de caracterización creados a partir de las imágenes.

5.4. Editores

En esta sección describiremos los editores que pueden ser usados en Cool Imaging. Recuérdese que, dentro del contexto de una aplicación RCP, un *editor* muestra información principal de la aplicación, con la característica de poder ser salvada. En nuestro caso, existen dos tipos de editores: el que permite mostrar imágenes y el que permite mostrar informes de caracterización.

El área de editores, es decir, la región donde se muestran los editores de la aplicación, implementa el mecanismo de *drop* con recursos de tipo fichero. Esto quiere decir que, sobre el área de editores, el usuario puede soltar ficheros previamente arrastrados. Si el fichero soltado sobre el área de editores almacena una imagen que Cool Imaging puede reconocer, o bien un informe de caracterización, inmediatamente se crea un nuevo editor que visualiza el contenido de dicho fichero.

5.4.1. Editor Imagen

El Editor Imagen es el Editor encargado de visualizar imágenes dentro de la aplicación. Cada vez que el usuario abre una imagen, o la crea, se crea un Editor Imagen encargado de visualizarla. Al usuario, el Editor Imagen le da la posibilidad no sólo de visualizar la imagen, sino también de realizar ciertas acciones sobre ella. Para ello, al Editor Imagen se le asocian acciones dinámicas que se mostrarán solamente cuando haya un Editor Imagen activo. En otro caso, dichas acciones no serán mostradas.

Las respectivas acciones son mostradas tanto en la cool bar de la aplicación como en la barra de menú. Las acciones, su respectiva función y su localización, se resumen en el siguiente listado.

- Ampliar imagen: permite ampliar la imagen seleccionada. Cada vez que se hace uso de esta acción, al imagen es aplicada una cierta cantidad. Esta acción es situada tanto en el cool bar como en el menú *Ver* de la barra de menú.

- Reducir imagen: permite reducir la imagen visualizada. Cada vez que se hace uso de esta acción, al imagen es reducida una cierta cantidad. Esta acción es situada tanto en el cool bar como en el menú *Ver* de la barra de menú.

- Ajustar imagen: permite seleccionar un área rectangular de la imagen; la imagen se centrará en dicho área rectangular, y será ampliada para abarcarla todo lo que sea posible. Esta acción es situada tanto en el cool bar como en el menú *Ver* de la barra de menú.

- Reestablecer tamaño de imagen: hace que la imagen visualizada pase a su tamaño normal. Esta acción es situada tanto en el cool bar como en el menú *Ver* de la barra de menú.

- Arrastrar imagen: permite, mediante el clásico mecanismo de *pulsar y arrastrar* un botón del ratón, desplazarse libremente por la imagen. Esta acción es situada tanto en el cool bar como en el menú *Ver* de la barra de menú.

- Extraer ROI: permite seleccionar una región arbitraria de la imagen, en forma de polígono, la cual será extraída a una nueva imagen. Esta acción es situada tanto en el cool bar como en un nuevo menú de la barra de menú, el menú *ROI*.

- Definir ROI: permite definir una región arbitraria de la imagen, en forma de polígono, que será marcada como ROI de la imagen. Esta acción es situada tanto en el cool bar como en un nuevo menú de la barra de menú, el menú *ROI*.

- Substraer ROI: permite substraer, de una ROI definida en una imagen, una región arbitraria, definida en forma de polígono. Esta acción es situada tanto en el cool bar como en un nuevo menú de la barra de menú, el menú *ROI*.

- Eliminar ROI: permite eliminar la ROI asociada a una imagen. Esta acción es situada tanto en el cool bar como en un nuevo menú de la barra de menú, el menú *ROI*.

- Eliminar punto polígono: permite eliminar el último punto de la región arbitraria que el usuario puede definir. Esta acción es situada tanto en el cool bar como en un nuevo menú de la barra de menú, el menú *ROI*.

- Limpiar polígono: permite limpiar la región arbitraria que el usuario puede definir en forma de polígono. Esta acción es situada tanto en el cool bar como en un nuevo menú de la barra de menú, el menú *ROI*.

Ciertas acciones realizadas sobre la imagen cambian el estado de *suciedad* del Editor Imagen. Concretamente, todas las acciones que tengan como resultado la modificación de la región de interés definida sobre la imagen, marcan a la imagen mostrada en el editor como sucia, es decir, pendiente de almacenar. Guardar una imagen a través de las acciones *Guardar* o *Guardar como...* de la barra de menú principal o de la cool bar de la aplicación, no sólo almacena la imagen, sino también la región de interés definida sobre ella, en un archivo cuyo nombre es igual al del archivo de la imagen, pero cuya extensión es *.roi*.

5.4.2. Editor Caracterización

El Editor Caracterización es el editor encargado de visualizar gráficamente un informe de caracterización. Cuando se crea un nuevo informe de caracterización o bien se abre alguno ya existente, se crea un Editor Caracterización para visualizar dicho informe.

Un informe de caracterización recoge los datos generados tras la caracterización de una o varias imágenes. El Editor Caracterización muestra principalmente los datos del informe de caracterización de forma tabular. Recuérdese que un informe de caracterización se organiza fundamentalmente en paquetes de imágenes. Cada paquete contiene imágenes caracterizadas, las cuales a su vez contienen los vectores de caracterización que representan los datos obtenidos a partir de ellas. Un informe de caracterización también puede contener imágenes caracterizadas que no están asociadas a ningún paquete de imágenes. Por último, un informe puede contener *resúmenes de vectores de caracterización*, es decir, vectores de caracterización que compactan, en uno sólo, los valores de varios vectores de caracterización. Estas medidas de resumen pueden estar asociadas a paquetes de imágenes (resumen de los vectores de las imágenes de un paquete de imágenes), o bien a nada (en cuyo caso representan resúmenes de los vectores de las imágenes que no están asociadas a ningún paquete de imágenes).

Los datos de caracterización del informe de caracterización son mostrados, dentro del Editor Caracterización, en formato tabular, en una tabla como la mostrada en la figura 5.22 (TablaEditorCaracterizacion):

Como se puede observar, la TablaEditorCaracterizacion muestra los datos del informe de caracterización de forma estructurada: para cada paquete de imágenes, muestra las imágenes caracterizadas, y para cada una de las imágenes, sus vectores de caracterización. Igualmente se muestran las imágenes que no están asociadas a ningún paquete del informe.

El Editor Caracterización, además de mostrar los datos de la caracterización contenidos en el informe de caracterización, muestra información adicional relativa al elemento seleccionado en la TablaEditorCaracterizacion. Dicha información se muestra en el PanelInfoElementoSeleccionado. De este modo, la estructura del Editor Caracterización es la mostrada en la figura 5.23:

El Editor Caracterización tiene asociadas acciones dinámicas, al igual que el Editor Imagen, que se muestran sólo cuando hay activo un Editor Caracterización. Dichas acciones permiten interactuar con el informe de caracterización. Estas acciones son las siguientes:

- Aumentar decimales: aumenta en 1 el número de decimales de los números mostrados en el informe de caracterización.

- Reducir decimales: redice en 1 el número de decimales de los números mostrados en el informe de caracterización.

- Ordenar alfabéticamente: activa la ordenación alfabética de los elementos mostrados en la TablaEditorCaracterizacion.

Elemento	Descripción	Valor
Paquete1	Paquete de imágenes	
Imagen1.jpg	Imagen caracterizada	
Vector1	Vector de caracterización	
Media	Medida de caracterización	
Media1	Media banda 1	66.2
Media2	Media banda 2	45.2
Media3	Media banda 3	194.5
Desviación típica	Medida de caracterización	
Desviación1	Desviación banda 1	77.97
Desviación2	Desviación banda 2	34.5
Desviación3	Desviación banda 3	10.4
Vector2	Vector de caracterización	
...		
Imagen2.jpg	Imagen caracterizada	
...		
VectorResumen1	Resumen de vectores	
VectorResumen2	Resumen de vectores	
...		
Paquete2	Paquete de imágenes	
...		
ImagenIndependiente1.jpg	Imagen caracterizada	
ImagenIndependiente2.jpg	Imagen caracterizada	
...		
VectorResumen1	Resumen de vectores	
VectorResumen2	Resumen de vectores	
...		

Figura 5.22: TablaEditorCaracterizacion

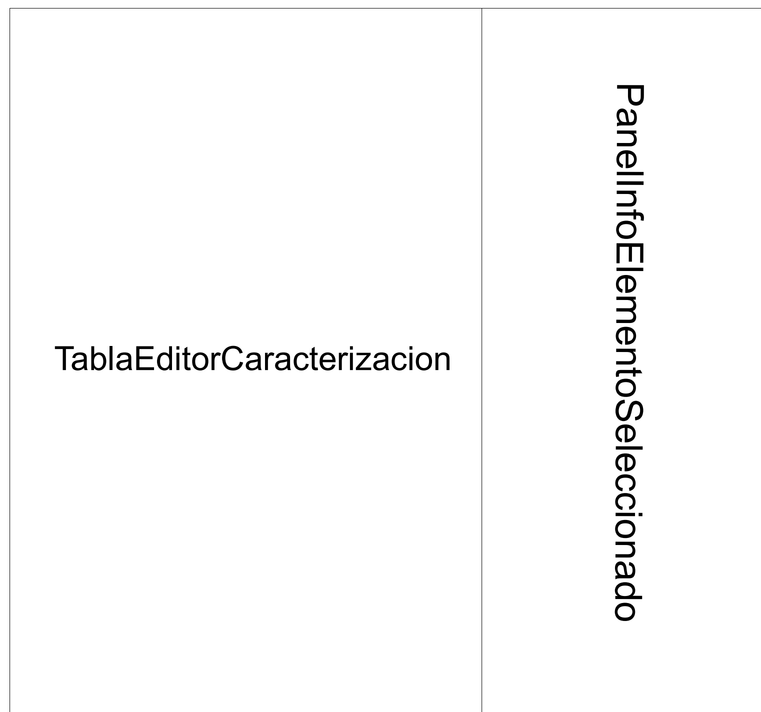

Figura 5.23: Editor Caracterización

- Mostrar área de información: hace que se muestre el PanelInfoElementoSeleccionado, el cual muestra información detallada del elemento seleccionado en la TablaInformeCaracterizacion.
- Mostrar medidas de resumen: hace que, en la TablaInformeCaracterizacion, sólo se muestren las medidas (vectores) de resumen.
- Resumir informe: permite calcular las medidas de resumen de los vectores de caracterización del informe, en caso de que no estén ya calculados.
- Exportar: lanza un asistente de exportación del informe de caracterización. Dicho asistente permite al usuario seleccionar vectores a exportar a formatos como ODS o XLS.

La acción que calcula las medidas de resumen del informe de caracterización cambia el estado de *suciedad* del Editor Caracterización. De este modo, el informe asociado es marcado como sucio, es decir, pendiente de almacenar. Las acciones *Guardar* y *Guardar como...* de la barra de menú principal y de la cool bar de la aplicación, permiten almacenar el un fichero de disco el informe de caracterización mostrado en el Editor Caracterización.

5.4.2.1. Asistente de exportación

La acción dinámica *Exportar* asociada al Editor Caracterización lanza un asistente de exportación que permite exportar, a distintos formatos, el informe de caracterización actual. Concretamente, el asistente permite seleccionar uno o varios vectores de caracterización a exportar, así como el formato al que exportar dichos vectores de caracterización.

El asistente muestra al usuario varias páginas a través de las cuales puede configurar la exportación del informe de caracterización.

La primera página es la que se muestra en la figura 5.24. En dicha página, el usuario puede seleccionar tanto los vectores de caracterización a exportar como el formato de exportación.

Si el formato elegido es el de un simple documento de texto (.dat), el usuario puede, además, introducir el separador de columnas a usar en el fichero creado.

Selección del formato de salida

Exportar a hoja de cálculo ODF

Exportar a Microsoft Excel 97/2000/XP

Exportar a Microsoft Excel 2007 XML

Exportar a documento de texto

Separador del documento de texto

Espacio en blanco

Tabulador

Otro separador

Selección del vector de caracterización a exportar

Vector1

Vector2

Vector3

Figura 5.24: Página primera del asistente de exportación

Un detalle cobra especial importancia en la exportación de vectores de caracterización. Aunque las imágenes de un informe de caracterización hayan sido caracterizadas por un mismo generador de vector de caracterización, es posible que los vectores de caracterización producidos tengan una estructura diferente.

Supongamos que, por ejemplo, se dispone de dos imágenes, una imagen A, en niveles de gris (tiene por tanto una sola banda), y otra imagen B, en sistema RGB (y que, por tanto, tiene tres bandas). Supongamos ahora que se dispone de un generador de vector de caracterización que calcula dos medidas de caracterización, a saber, la media de los píxeles de cada banda de la imagen, y la desviación típica de los píxeles de cada banda de la imagen.

Los vectores de caracterización obtenidos por este generador para la imagen A y la imagen B tendrían una estructura bastante diferente, problema que tiene su origen en el hecho de que ambas imágenes tienen un número de bandas distinto.

Así, el vector de caracterización de la imagen A podría ser $V_A = \{98{,}4; 13{,}77\}$, y el de la imagen B podría ser $V_B = \{\{23{,}9; 54{,}78; 119{,}4\}; \{21{,}22; 13; 7{,}6\}\}$. El vector V_A sería intrínsecamente más simple que el vector V_B, ya que sólo debería almacenar la media y la desviación típica de una única banda. Por contra, el vector V_B debería almacenar las medias de tres bandas, así como las desviaciones típicas de tres bandas. Ambos vectores tendrían una estructura distinta (incompatible).

La existencia de vectores con estructuras incompatibles supone un serio problema a la hora de llevar a cabo la exportación de vectores de caracterización. Supongamos que el usuario quiere exportar el vector de caracterización *Vector1* (debería seleccionar dicho vector en la primera página del asistente de exportación). La idea del asistente es exportar todos los vectores de identificador *Vector1* presentes en todas las imágenes caracterizadas del informe. Si todos esos vectores tienen la misma estructura, no habría ningún problema. Sin embargo, si la estructura de

Figura 5.25: Página segunda del asistente de exportación

Figura 5.26: Última página del asistente de exportación

esos vectores difiriese entre sí, no podrían exportarse todos a un mismo archivo. La naturaleza de los ficheros estadísticos con estructura de tabla obliga a que todas las filas de la tabla (muestras) tengan los mismos atributos (columnas). En este caso, los vectores de caracterización (filas de la tabla) podrían tener distintos atributos (estructura del vector), y por tanto no podrían compactarse en una única tabla.

La solución adoptada para resolver este problema es la de permitir exportar a un archivo solamente vectores de caracterización que tienen la misma estructura.

Si en la primera página del informe de caracterización se detecta que alguno de los vectores seleccionados tiene más de una estructura de vector asociada, la siguiente página del asistente le permite al usuario seleccionar qué estructura es la que desea exportar para dichos vectores. Esta página del asistente es la mostrada en la figura 5.25.

En el listado de la izquierda aparecen todas las estructuras asociadas a los vectores que tienen más de una estructura asociada. Así, por ejemplo, si el vector de caracterización *Vector1* tiene asociadas dos estructuras, nombradas como *(1) Vector1* y *(2) Vector1*. Cuando se selecciona (que no chequea), un elemento de este listado, como el *(2) Vector1* de la figura 5.25, en la tabla de arriba a la derecha se muestra la estructura asociada. En la tabla de abajo a la derecha se muestra un listado de las imágenes que tienen esa estructura de vector de caracterización.

Una vez que el usuario ha seleccionado los vectores a exportar, el asistente pasa a la última página, donde introduce los nombres de los ficheros de salida en los que almacenar los vectores (o estructuras de vectores) seleccionadas. Se trata de la página mostrada en la figura 5.26.

En dicha página el usuario introduce los nombres de los ficheros de salida, tanto para los vectores que tenían una sola estructura asociada (como el *Vector5* de la imagen), como para las estructuras seleccionadas de aquellos vectores con más de una estructura (el resto de las entradas del panel).

Capítulo 6

Diseño

El Diseño, dentro del proceso de desarrollo de software, es la etapa en la cual, mediante la aplicación de diferentes técnicas y principios, se consigue definir con suficiente detalle el sistema planteado en las etapas de modelado de requisitos y análisis como para permitir su construcción física. El diseño es, en suma, el paso del qué al cómo. El Lenguaje Unificado de Modelado, o UML, nos servirá, nuevamente, como principal herramienta de modelado. El diagrama de clases de diseño nos permitirá definir una estructura de clases para el lenguaje de programación Java que represente el conjunto de clases de diseño necesarias para una implementación física del sistema. Mediante los diagramas secuencia y de colaboración de diseño[1], además, se describirá de forma precisa la interacción entre los distintos objetos de diseño durante la ejecución de las operaciones más relevantes del sistema, con el propósito de que la posterior implementación sea más fácil y esté libre de errores.

No podemos olvidar que el sistema está ligado (principalmente) a la arquitectura basada en *plugins* analizada en la fase de análisis, y que será la que finalmente diseñaremos e implementaremos. Los patrones de diseño MVC y *Observer*, en menor grado, también guiarán el diseño del sistema.

La arquitectura basada en *plugins* obliga a que el proyecto deba estructurarse como un conjunto de plugins interconectados entre sí. Es por ello que será necesario llevar a cabo el diseño del sistema en lo que a *plugins* respecta: no sólo qué *plugins* intervienen, sino cuáles son los puntos de extensión más relevantes.

Este capítulo recoge principalmente el diseño de la aplicación a nivel de clases, es decir, las clases necesarias para implementar la funcionalidad recogida en las fases anteriores del proceso de desarrollo del software. El *storyboard*, dentro de la fase de diseño, se muestra especialmente útil, ya que da una idea bastante clara de cómo debe organizarse la interfaz gráfica de la aplicación. Con ello, el diseño de clases de la interfaz de usuario será más que evidente.

6.1. Arquitectura de *plugins* de Eclipse RCP

En esta sección se recoge una visión global y precisa de la estructura de *plugins* de Cool Imaging.

Cool Imaging es una aplicación basada en el RCP de Eclipse. Tal y como explicamos, RCP es un *framework* de trabajo que da un soporte muy efectivo en lo que se refiere a la creación

[1]Los diagramas de secuencia y de colaboración son semánticamente equivalentes, lo cual significa que ambos pueden representar exactamente lo mismo. Sin embargo, los diagramas de secuencia son más fáciles de comprender, ya que representan de una forma más intuitiva el flujo de mensajes entre objetos. Por contra, un diagrama de colaboración es más compacto, ocupando bastante menos espacio que el equivalente diagrama de secuencia, pero es bastante menos legible. En general, seguiremos la siguiente regla: usaremos diagramas de secuencia salvo cuando sea tan grande que se haga inmanejable. En ese caso, se hará uso de los diagramas de colaboración.

de proyectos basados en *plugins*[2].

Todo *plugin* tiene una serie de elementos asociados, de entre los que tienen especial relevancia los siguientes:

- **Identificador del *plugin***: es una cadena de texto que identifica el *plugin* de forma unívoca. El identificador de un *plugin* es universal, y por tanto no debería coincidir con el identificador de ningún otro *plugin* existente. Para fomentar esta premisa, se suele usar nombres de dominios invertidos, como por ejemplo *com.XXX.XXX.proyecto*.

- **Listado de clases e interfaces públicas**: un *plugin* define una serie de interfaces y clases que declara como públicas. Toda interfaz o clase marcada como pública puede ser accedida externamente desde otros *plugins*. Este listado constituye un punto clave dentro de la interfaz pública del *plugin* (tal y como se explicó en la fase de análisis), ya que permite la construcción de sistemas modulares en los que unos *plugins* hacen uso de las clases e interfaces definidas por otros.

- **Puntos de extensión**: tal y como se explicó en la fase de análisis, un punto de extensión define un modo a través del cual otros *plugins* externos pueden extender la funcionalidad de un *plugin* determinado. Los puntos de extensión de un *plugin* definen el resto de la interfaz pública del *plugin*. Un punto de extensión está descrito, fundamentalmente, por dos elementos:

  - Un identificador: el identificador es usado por aquellos *plugins* externos que quieren conectarse al *plugin* que define el punto de extensión. Si un *plugin* quiere conectarse a un punto de extensión determinado, debe hace uso del identificador del punto de extensión, para indicar a qué punto de extensión quiere conectarse.
  - Una clase o interfaz: todo *plugin* externo que quiera conectarse al punto de extensión, deberá proporcionar una clase que implemente la interfaz o que extienda la clase especificada. Tal y como se explicó en la fase de análisis, es dicha clase la encargada de extender de forma real la funcionalidad del *plugin*[3].

- **Un listado *plugins* de los que depende**: recuérdese que una parte central de la filosofía de una aplicación basada en *plugins* es la de descomponer la aplicación en *plugins* independientes que, interconectados entre sí, den forma al sistema. Así, es frecuente la situación en la que un *plugin* depende de otro, en el sentido de que hace uso de su interfaz pública (bien de las clases públicas definidas por el *plugin* o bien contribuyendo a sus puntos de extensión). La figura 4.4 recoge esta idea. Un *plugin*, por tanto, tiene asociado un listado de dependencias externas, a saber, otros *plugins* de los que hace uso.

6.1.1. *Core Runtime* y *Eclipse UI*

Toda aplicación basada en Eclipse RCP consta de dos *plugins* principales, encargados de arrancar la aplicación y visualizar la interfaz gráfica, a saber, *org.eclipse.core.runtime* y *org.eclipse.ui*. Estos dos *plugins* permiten la construcción de una aplicación RCP estándar, basada en editores y vistas, como lo es, por ejemplo, Eclipse SDK.

[2]Además de, por supuesto, una gran cantidad de componentes que agilizan el diseño de interfaces gráficas y del software en general.

[3]Un punto de extensión de Eclipse RCP ofrece una potencialidad bastante mayor que la aquí explicada. A pesar de ello, dado que los requerimientos de extensibilidad de Cool Imaging no son excesivamente complejos, nos centraremos en la definición de puntos de extensión bastante simples, cuyo principal componente sea la implementación o extensión de una interfaz o clase definida en el punto de extensión.

Estos dos *plugins* ofrecen una serie de puntos de extensión que, al ser usados, permiten la ejecución de una aplicación RCP. Sus principales puntos de extensión son:

- *org.eclipse.core.runtime.applications*: permite especificar una clase que, dentro de la aplicación, implemente la interfaz *IApplication*. Esta clase define el punto de arranque de la aplicación, a través de su método *start()*. En el contexto de una aplicación RCP, el método *start()* suele crear y ejecutar un *WorckbenchAdvisor*, clase encargada de la inicialización del aspecto gráfico de la aplicación. En suma, esta clase inicializa el *Workbench*, el cual ofrece al usuario editores y vistas con una distribución (*layout*) concreta. El *Workbench* define un potente paradigma de diseño de interfaces gráficas, basado en ventanas, perspectivas, vistas, editores y acciones. Además, ofrece mecanismos de extensibilidad mediante puntos de extensión del plugin *org.eclipse.ui*.

- *org.eclipse.ui.editors*: punto de extensión a través del cual se pueden añadir nuevos editores a la aplicación. Todo editor que el usuario desee añadir debe extender la clase abstracta *EditorPart*.

- *org.eclipse.ui.views*: punto de extensión a través del cual se pueden añadir nuevas vistas a la aplicación. Toda vista que el usuario desee añadir debe extender la clase abstracta *ViewPart*.

Un *plugin* puede *reexportar* toda o parte de la interfaz pública de otros *plugins* de los que hace uso. Se podría decir que existe un *plugin* padre que puede reexportar la interfaz pública de sus hijos. Este mecanismo de reexportación es de gran utilidad en el desarrollo de aplicaciones basadas en *plugins*, ya que permite que un *plugin*, el padre, muestre parte de la API de otro *plugin* (hijo o hijos) como si fuera su API propia. Así, por ejemplo, una aplicación RCP que hiciera uso del *plugin org.eclipse.ui* podría reexportar los puntos de extensión de dicho *plugin*. De este modo, el *plugin* principal de la aplicación RCP permitiría hacer uso de puntos de extensión como *org.eclipse.ui.editors*. Otros *plugins* externos, como consecuencia, podrían hacer uso de dicho punto de extensión, para añadir nuevos editores a la aplicación RCP.

6.1.2. Estructura de *plugins* de Cool Imaging

Cool Imaging es una aplicación RCP. Así, hace uso de los *plugins org.eclipse.core.runtime* y *org.eclipse.ui*.

Cool Imaging consta de un *plugin* principal, *com.coolimagingproject.coolimaging*, el cual se encarga de arrancar la aplicación e inicializar la interfaz gráfica. Por brevedad, este *plugin* será llamado simplemente *coolimaging*. El *plugin coolimaging* depende de forma directa tanto de *org.eclipse.core.runtime* (*plugin runtime* de ahora en adelante) como de *org.eclipse.ui* (*plugin ui* de ahora en adelante). Principalmente, hace uso de los puntos de extensión explicados en la sección 6.1.1, inicializando el *Workbench*, las vistas y las perspectivas. Cool Imaging reexporta los puntos de extensión tanto del *plugin runtime* como del *plugin ui*. Así, Cool Imaging puede ser extendida a su vez mediante estos puntos de extensión, permitiendo principalmente añadir nuevas vistas y editores a través de *plugins* externos.

El *plugin coolimaging*, además, define dos puntos de extensión adicionales. Uno de ellos, el más importante, permite añadir nuevas operaciones a la aplicación, tanto de caracterización como de tratamiento de imágenes. El otro permite registrar operaciones de la biblioteca JAI en la aplicación. A continuación se proporciona una descripción más detallada:

- *com.coolimagingproject.coolimaging.operationApplication*: este punto de extensión (*operationApplication* de ahora en adelante) permite añadir nuevas operaciones de caracterización o de tratamiento de imágenes a la aplicación. Todo *plugin* externo que desee añadir una nueva operación a Cool Imaging debe hacer uso de este punto de extensión.

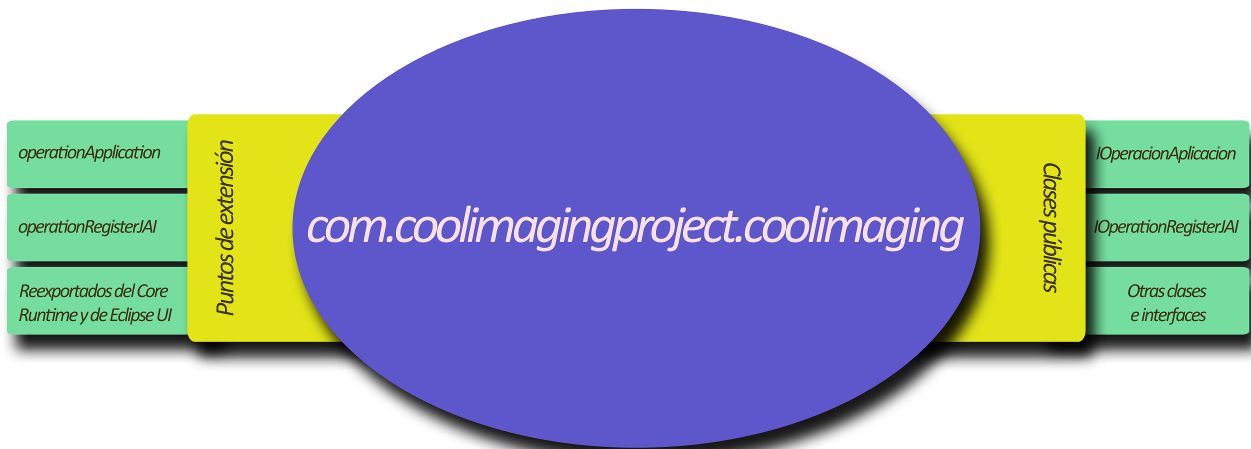

Figura 6.1: *Plugin coolimaging*

Este punto de extensión requiere que el *plugin* externo que hace uso de él proporcione una (o varias) clases que implementen la interfaz *IOperadorAplicacion*. La interfaz *IOperadorAplicacion*, accesible a través de las clases públicas definidas por el *plugin coolimaging*, define los métodos necesarios para poder hacer uso de una operación dentro de Cool Imaging. Un *plugin* puede proporcionar una o varias clases que implementen la interfaz *IOperadorAplicacion*. Así, un *plugin* externo puede añadir al sistema varias operaciones, y no sólo una.

- *com.coolimagingproject.coolimaging.operationRegisterJAI*: este punto de extensión (*operationRegisterJAI* de ahora en adelante) es usado para registrar nuevas operaciones de JAI en Cool Imaging. Recuérdese que en Cool Imaging el núcleo del tratamiento de imágenes digitales está construido sobre la bibliteca JAI. JAI ofrece al usuario un repertorio de operaciones de tratamiento de imágenes. Sin embargo, si el usuario quiere definir nuevas, y usarlas, éstas deben ser *registradas* en JAI.

  Este punto de extensión requiere que el *plugin* externo que hace uso de él proporcione una (o varias) clases que implementen la interfaz *IOperationRegisterJAI*. La interfaz *IOperationRegisterJAI*, accesible a través de las clases públicas definidas en el *plugin coolimaging*, define un método, *register()*, en el cual se debe llevar a cabo el registrado de la operación (u operaciones) a registrar en JAI.

La figura 6.1 representa el *plugin coolimaging*, con sus puntos de extensión y sus principales clases e interfaces públicas.

En las siguientes secciones se ofrecerá una descripción más detallada de las interfaces y clases implicadas en el uso de ambos puntos de extensión.

Como es de esperar un un *plugin* principal, el *plugin coolimaging* define, además de los puntos de extensión descritos, una serie de clases e interfaces públicas que pueden ser accedidas desde otros *plugins* externos. Dentro de este grupo de clases e interfaces destacan, sobretodo, las interfaces *IOperadorAplicacion* e *IOperationRegisterJAI*. Ambas deben ser accesibles por aquellos *plugins* que quieran contribuir a los puntos de extensión antes mencionados.

Aunque el núcleo de Cool Imaging sea el *plugin* principal, *coolimaging*, la aplicación creada viene acompañada de varios *plugins* que incrementan su funcionalidad:

- *com.coolimagingproject.basicImageProcessingOperations*: este *plugin* contiene operaciones básicas de tratamiento de imágenes: filtrado lineal, rotación, escalado, etc. Hace uso del punto de extensión *operationApplication* del *plugin coolimaging*, para añadir así cada una de las operaciones.

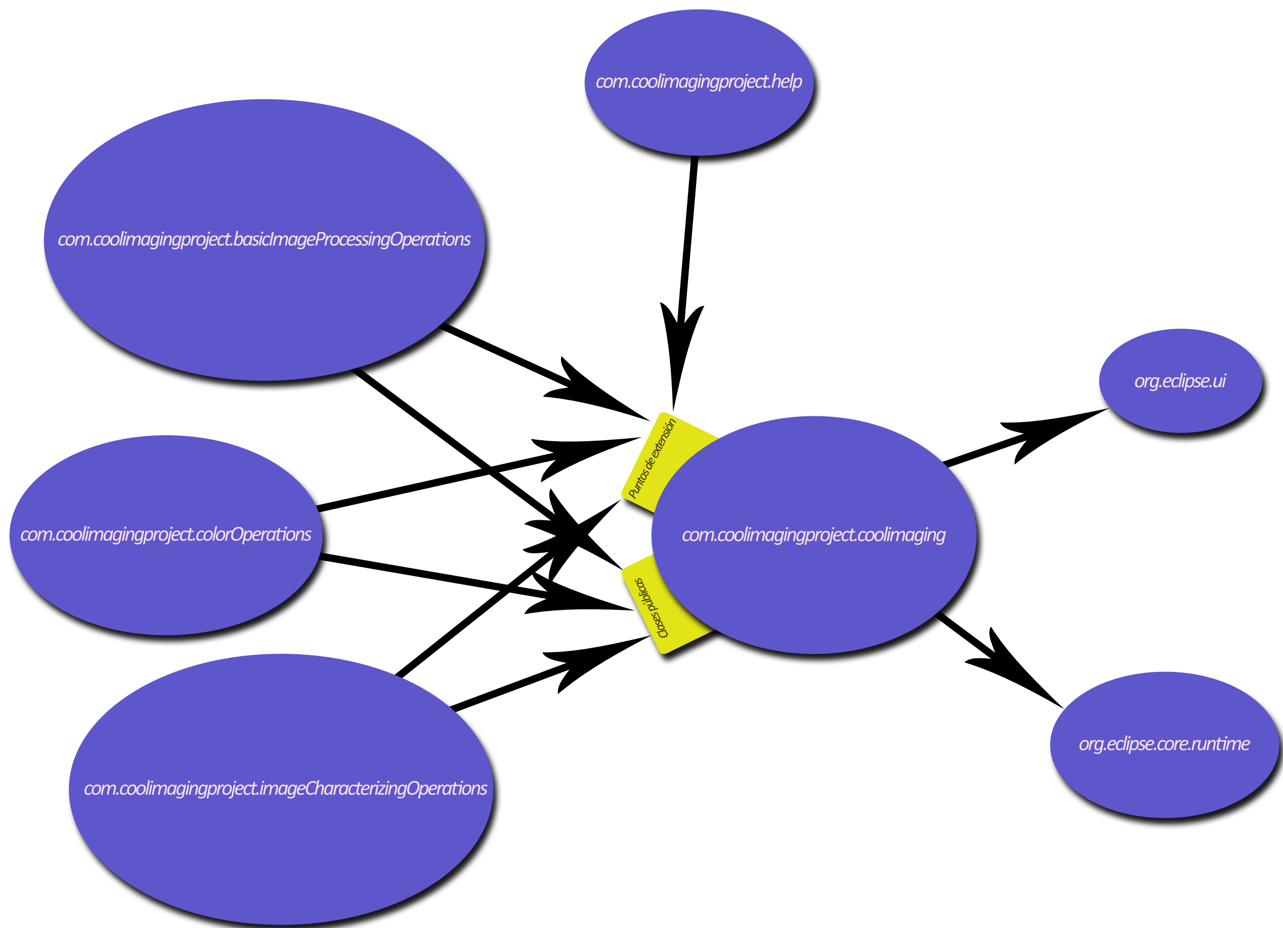

Figura 6.2: Arquitectura de Cool Imaging

- *com.coolimagingproject.colorOperationes*: este *plugin* contiene operaciones de conversión entre distintos espacios de color. Hace uso del punto de extensión *operationApplication* del *plugin coolimaging*, para añadir así cada una de las operaciones.

- *com.coolimagingproject.basicImageCharacterizingOperations*: este *plugin* contiene operaciones básicas de caracterización de imágenes, basadas en el histograma y en la matriz de co-ocurrencia. Hace uso del punto de extensión *operationApplication* del *plugin coolimaging*, para añadir así cada una de las operaciones. Este *plugin* también hace uso del punto de extensión *operationRegisterJAI*, ya que hace uso de nuevas operaciones de JAI que requieren ser registraras previamente a su uso.

- *com.coolimagingproject.help*: este *plugin* contiene el manual de ayuda en línea de Cool Imaging, el cual puede ser accedido desde la misma aplicación. Hace uso del punto de extensión *org.eclipse.help.toc*, el cual permite especificar los ficheros XML y HTML que componen el manual de ayuda[4]. En este sentido merece la pena señalar de que se trata de un punto de extensión distinto a los que hemos visto hasta ahora, ya que no especifica requiere que se implemente o extienda una interfaz o clase.

  La figura 6.2 representa la arquitectura de Cool Imaging como un conjunto de *plugins* interconectados entre sí.

[4]El sistema de ayuda de Eclipse obliga a especificar, por una lado, el contenido de la ayuda, y por otro su estructura. El contenido lo forman un conjunto de ficheros HTML. La estructura de la ayuda (índices, subíndices. etc.), se especifica a partir de ficheros XML

6.2. Manipulación básica de imágenes

Cool Imaging es una aplicación de procesamiento de imágenes digitales. Es necesario, por tanto, que la base sobre la que se construya toda la lógica de manipulación de imágenes sea lo suficientemente potente como para no comprometer la funcionalidad futura del sistema.

El bloque lógico que se encarga del tratamiento digital de imágenes dentro de la aplicación ha sido diseñado partiendo de la base de que se haría uso de la biblioteca JAI.

La biblioteca JAI ofrece una API suficientemente potente para la manipulación de imágenes. Dentro de la aplicación, sin embargo, nos hemos visto obligados a extender JAI, para amoldar dicha biblioteca a nuestros requerimientos.

Esta extensión se plasma, principalmente, en la representación gráfica de imágenes. Cool Imaging debe permitir representar imágenes en distintos factores de zoom, así como regiones de interés asociadas a las imágenes.

El diagrama de clases correspondiente es el de la figura 6.3.

6.2.1. Clase Imagen

6.2.1.1. Imagen

La clase *Imagen* representa una imagen digital que puede ser accedida en modo lectura y escritura. La clase *Imagen* hereda de la clase *TiledImage* de la biblioteca JAI. Esta clase ha sido construida con el propósito de brindar funcionalidad extra que se pretenda añadir a las imágenes de nuestra aplicación. Así por ejemplo, la clase *Imagen* define una función que permite almacenar la imagen en un determinado archivo.

6.2.2. Representación gráfica de imágenes

Un objeto de tipo *Imagen* debe ser visualizado gráficamente en la aplicación. Son varias las clases encargadas de ello.

6.2.2.1. DisplayJAIConZoom

La clase *DispayJAIConZoom* extiende la clase *DisplayJAI*. La clase *DisplayJAI* es una clase de la librería JAI, que permite la representación gráfica de objetos de tipo *RenderedImage*. La clase *Imagen*, en todo caso, implementa la interfaz *RenderedImage*, así que puede ser visualizada mediante un *DisplayJAI*.

La clase *DisplayJAIConZoom* ofrece funcionalidad extra. Al extender la clase *DisplayJAI*, puede representar imágenes. Además, permite visualizar las imágenes con un cierto factor de zoom, mostrándolas ampliadas o reducidas dependiendo del factor de zoom establecido.

6.2.2.2. DisplayJAIConGraficos

La clase *DisplayJAIConGraficos* extiende la clase *DisplayJAIConZoom*. La clase *DisplayJAIConGraficos* ofrece toda la funcionalidad de la clase *DisplayJAIConZoom*, y además permite la visualización de elementos gráficos adicionales sobre la imagen.

En el contexto de Cool Imaging hay circunstancias en las que no sólo se visualiza una imagen, sino una imagen y ciertos elementos adicionales encima de ella. Por ejemplo, si la imagen tiene asociada una región de interés (ROI), ésta se visualiza sobre la imagen. Cuando el usuario está seleccionando un área a escalar de la imagen, se debe mostrar el clásico rectángulo de ajuste que indica qué área va a ser escalada.

La clase *DisplayJAIConGraficos* ofrece métodos genéricos que permiten la visualización de este tipo de elementos gráficos sobre la imagen.

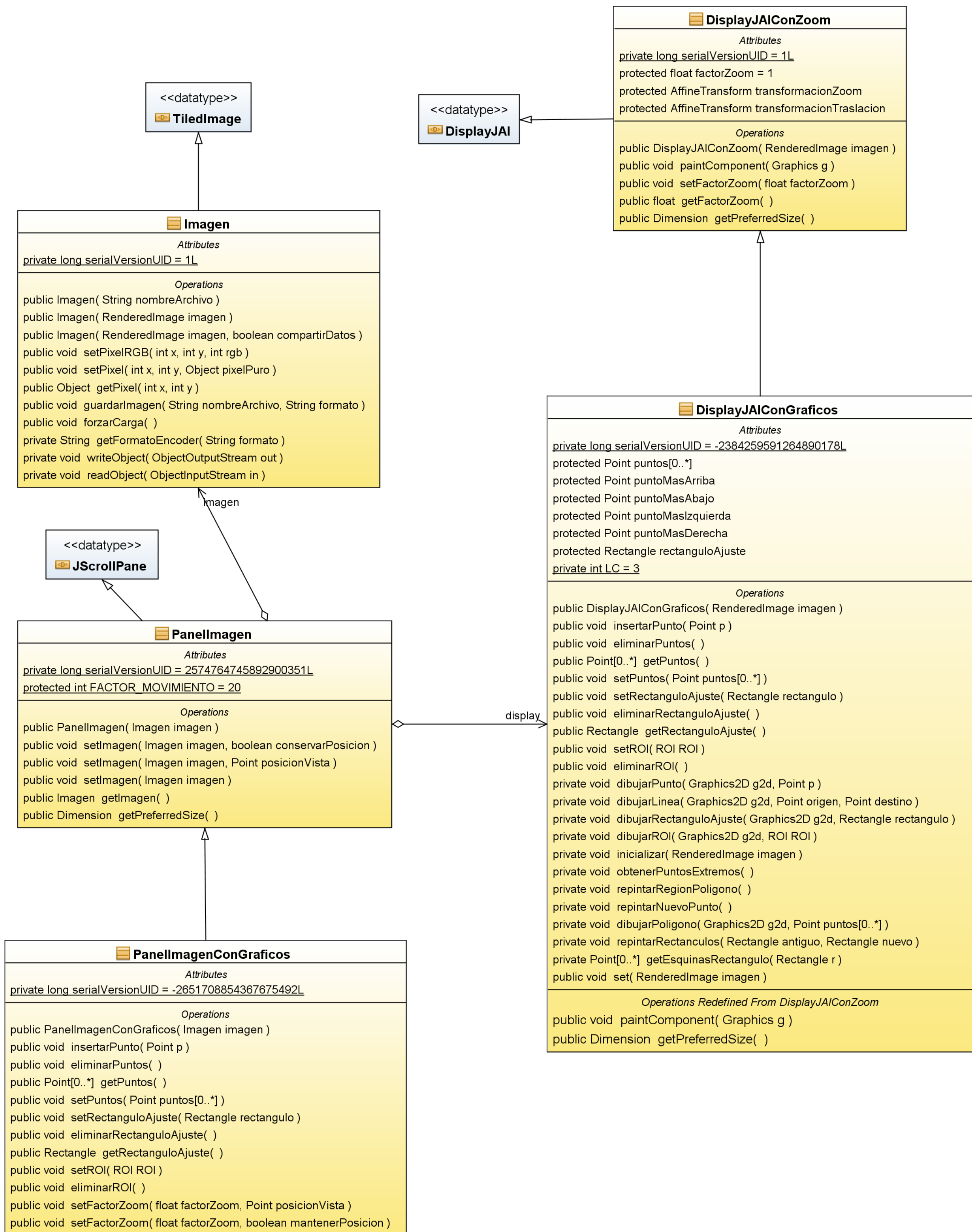

Figura 6.3: Diagrama de clases del módulo de manipulación básica de imágenes

6.2.2.3. PanelImagen

La clase *PanelImagen* hereda de *JScrollPane*, y permite representar un objeto de tipo *Imagen*, de modo que cuando el tamaño del panel no sea suficientemente grande como para visualizar toda la imagen, barras de desplazamiento tanto horizontales como verticales permitan desplazarse por toda la imagen.

Internamente, la clase *PanelImagen* almacena un objeto de tipo *DisplayJAIConGraficos*. Este objeto *DisplayJAIConGraficos*, que es el que se encarga realmente de la visualización del objeto tipo *Imagen*, es añadido al *PanelImagen*, de modo que se muestran barras de desplazamiento cuando es necesario.

Para representar gráficamente una imagen no se hace uso directo de las clases *DisplayJAIConZoom* o *DisplayJAIConGraficos*, ya que dichas clases no ofrecen soporte para mostrar las barras de desplazamiento en caso de que la imagen no quepa en el panel. Es por ello que es necesario construir una abstracción de mayor nivel, a saber, la clase *PanelImagen*.

6.2.2.4. PanelImagenConGraficos

La clase *PanelImagenConGraficos* es un panel gráfico que hereda de *PanelImagen*. La clase *PanelImagenConGraficos* permite, no sólo representar una *Imagen*, sino además ciertos elementos gráficos sobre ésta. El principal objetivo de esta clase es permitir representar otros elementos gráficos, sobre la imagen, que puedan resultar de interés. Por ejemplo, este panel permite la representación de un polígono arbitrario cualquiera: dicho polígono puede ser usado para la representación de la ROI conforme el usuario la va seleccionando. Este panel permite la representación, también, de un *rectángulo de ajuste*. Un *rectángulo de ajuste* es un rectángulo semitransparente que indica un área de la imagen a ser ampliada: cuando se hace uso de la herramienta de la selección de un área a escalar de la imagen, se debe representar gráficamente el área seleccionada. Dicho área puede ser representada gráficamente mediante esta clase. Esta clase también permite representar gráficamente una región de interés asociada a una imagen. Por otro lado, la clase *PanelImagenConGraficos* permite ampliar o reducir la imagen representada. La ampliación o reducción afecta sólo a la visualización, no a la imagen subyacente.

Internamente, la clase *PanelImagenConGraficos* almacena un objeto de tipo *DisplayJAIConGraficos* (clase que hereda de *DisplayJAIConZoom)*, el cual es, en última instancia, el que dibuja sobre la imagen todos los elementos gráficos adicionales, y el que se encarga de la representación de la imagen al factor de zoom adecuado. Realmente, la clase *PanelImagen* contiene un objeto de tipo *DisplayJAIConGraficos*, pero la clase *PanelImagen* no permite acceder a toda la funcionalidad adicional de dicha clase, mientras que la clase *PanelImagenConGraficos* sí.

6.3. Control de imágenes

Esta sección recoge la explicación del diseño detallado de las clases que gestionan la presentación de imágenes en la aplicación.

La aplicación puede contener, en todo momento, una serie de imágenes que se encuentran *abiertas*, es decir, que están siendo usadas por el sistema actualmente. Una imagen que se encuentra *abierta* debe poder ser visualizada en la aplicación. En caso de que haya varias imágenes abiertas, éstas pueden visualizarse de forma simultanea, aunque varias podrían mantenerse ocultas para ser visualizadas en cualquier otro momento.

Así pues, el sistema debe llevar a cabo un seguimiento de todas las imágenes que actualmente está usando. Ahora bien, ¿cómo se representa una imagen que está siendo usada por el sistema? La clase *ModeloImagen* representa toda la información que el sistema necesita para manipular una imagen. El conjunto de todas las imágenes que el sistema mantiene abiertas está repre-

sentado mediante la clase *ConjuntoModeloImagen*, la cual viene a almacenar, simplemente, un conjunto de objetos de tipo *ModeloImagen*.

Las clases *ModeloImagen* y *ConjuntoModeloImagen* están íntimamente relacionadas con las clases *VistaImagen* y *VistaConjuntoModeloImagen* respectivamente. La clase *VistaImagen* se encarga de la representación gráfica de un objeto de tipo *ModeloImagen*, de modo que ésta clase no representa más que un panel gráfico donde se visualiza la imagen asociada a un *ModeloImagen* determinado.

La clase *VistaImagen* está sincronizada con la clase *ModeloImagen* mediante el patrón *Observer*: el *ModeloImagen* se declara como un objeto *observado* (deriva de la clase *Observable*), mientras que la *VistaImagen* se declara como un objeto *observador* (implementa la interfaz *Observer*), de modo que en el momento de la creación de un objeto tipo *VistaImagen*, la *VistaImagen* es registrada como *observadora* de un *ModeloImagen* determinado. Así, cualquier cambio que se produce en el *ModeloImagen* es inmediatamente notificado a la *VistaImagen*, de modo que la representación gráfica de la imagen es inmediatamente actualizada. Por ejemplo, si se modifica el factor de zoom asociado a una imagen, el cambio será reflejado de manera automática por la *VistaImagen*.

Igualmente, la clase *VistaConjuntoModeloImagen* está sincronizada con la clase *ConjuntoModeloImagen* mediante el patrón *Observer*: el *ConjuntoModeloImagen* se declara como un objeto *observado* (deriva de la clase *Observable*), mientras que la *VistaConjuntoModeloImagen* se declara como un objeto *observador* (implementa la interfaz *Observer*), de modo que en el momento de la creación de un objeto tipo *VistaConjuntoModeloImagen*, la *VistaConjuntoModeloImagen* es registrada como *observadora* de un *ConjuntoModeloImagen* determinado. Así, cualquier cambio que se produce en el *ConjuntoModeloImagen* es inmediatamente notificado a la *VistaConjuntoModeloImagen*, de modo que la representación gráfica del *ConjuntoModeloImagen* es inmediatamente actualizada. Si bien la clase *VistaImagen* es la encargada de visualizar una imagen concreta, la clase *VistaConjuntoModeloImagen* es la encargada de crear dichos objetos *VistaImagen* y hacerlos visibles dentro de la aplicación.

El sistema no hace uso de estas clases de forma independiente, sino que emplea una clase mediadora entre ellas, el *ControladorImagenes*. La clase *ControladorImagenes* es la encargada de inicializar el *ConjuntoModeloImagen* que almacena las imágenes de la aplicación, la *VistaConjuntoModeloImagen* encargada de visualizarlas, y sincronizarlos entre sí. El *ControladorImagenes* permite obtener el *ConjuntoModeloImagen* de la aplicación. Obsérvese que este controlador sigue la filosofía de *controlador de instanciación*, es decir, se trata de un controlador que simplemente inicializa y sincroniza el modelo con la vista. Dentro de la aplicación, el controlador sólo debe preocuparse de proporcionar su instancia del modelo a quien la requiera, para que ésta sea modificada por quien fuera necesario: el patrón *Observer* será el encargado de actualizar la vista según los cambios producidos en el modelo.

A continuación se recoge una explicación detallada de cada una de las clases anteriormente nombradas. La figura 6.4 recoge el diagrama de clases de diseño del control de la presentación de las imágenes en la aplicación.

6.3.1. ModeloImagen

La clase *ModeloImagen* tiene como objetivo representar toda la información necesaria para trabajar, dentro de la aplicación, con una imagen. Se podría pensar que la clase *Imagen* satisface esa necesidad, pero no es cierto. Ciertos aspectos relacionados con una imagen al nivel de la aplicación no son recogidos por la clase *Imagen*, la cual sólo se encarga de almacenar los datos intrínsecos de una imagen, fundamentalmente la matriz de datos y el modelo de color de la imagen. Por ejemplo, dentro de la aplicación, una imagen tiene asociado un nivel de *zoom* que representa cuán ampliada se ve la imagen dentro de la aplicación, o también un identificador

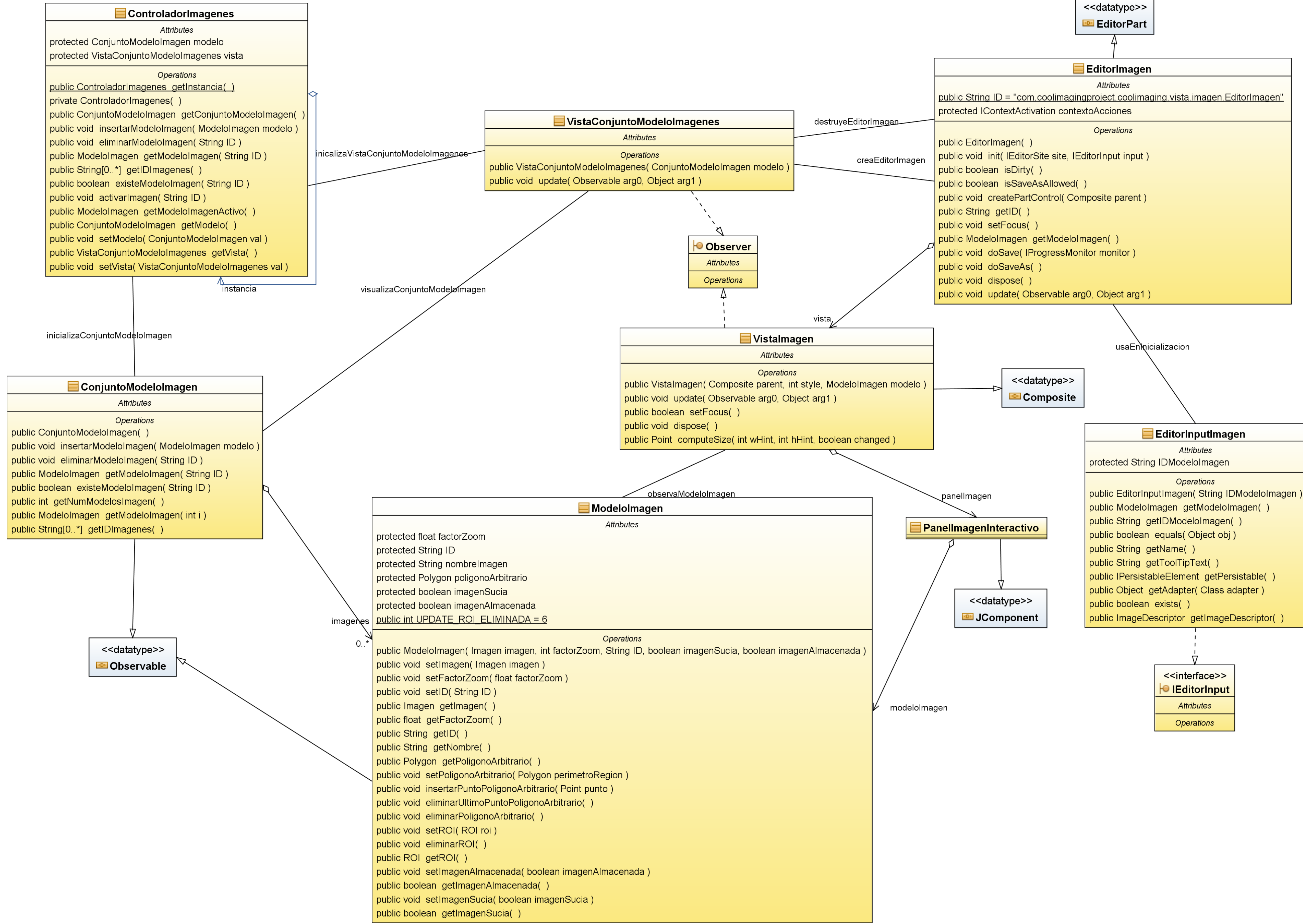

Figura 6.4: Diagrama de clases del control de imágenes

que la distingue de todas las demás imágenes abiertas. Dentro de la aplicación, además, una imagen también tiene asociados una región de interés (ROI), así como un polígono arbitrario definido sobre la imagen. Es la clase *ModeloImagen* la encargada de almacenar y gestionar estos detalles que no son innatos a la clase *Imagen* y que, sin embargo, son necesarios para la correcta gestión de las imágenes dentro de la aplicación. Es de especial interés distinguir entre la ROI de una imagen y el *polígono arbitrario*. Una ROI es una región de interés de una imagen, la cual es usada para cálculos numéricos con la imagen. El polígono arbitrario es un simple polígono definido sobre la imagen, el cual, aun representando una región de cierto interés en la imagen, no es usada en cálculos, como la ROI. Mientras que la ROI es usada a la hora de procesar o caracterizar una imagen, el polígono arbitrario no lo es. El polígono arbitrario es usado cuando se desea señalar una región de la imagen, sin ser ROI (por ejemplo, cuando se van seleccionando los puntos de una ROI de una imagen, antes de extraer en efecto la ROI, dichos puntos deben ser marcados en la imagen. El polígono arbitrario se encarga de almacenar dichos puntos). Es por ello que es necesario distinguir entre ambos conceptos.

La clase *ModeloImagen* hereda de la clase *Observable*, ya que representa un modelo de datos que puede ser potencialmente observado por otros objetos a la espera de cambios (objetos de tipo *observador*, según la filosofía del patrón *Observer*). Cuando se produce un cambio en un objeto de tipo *ModeloImagen*, los cambios son notificados a los objetos observadores (que implementan la interfaz *Observer*).

6.3.2. VistaImagen

La clase *VistaImagen* es la representación gráfica del *ModeloImagen*. Un objeto de la clase *ModeloImagen* es representado gráficamente en la aplicación mediante un objeto del tipo *VistaImagen*. Esta clase tiene especial interés cuando pensamos en aspectos como el grado de *zoom* de una imagen, que son fácilmente recogidos según la arquitectura dictada por los patrones *MVC* y *Observer*. Todo objeto de tipo *VistaImagen* es *observador* de un *ModeloImagen*, el cual es registrado en la *VistaImagen* en el momento de su creación. La clase *VistaImagen* implementa la interfaz *Observer*, de modo que su método *update()* se encarga de actualizar la representación gráfica cuando el *ModeloImagen* le notifica algún cambio.

La clase *VistaImagen* no sólo es capaz de recoger el factor de zoom asociado a una imagen, sino que también es capaz de representar la ROI del *ModeloImagen* así como el polígono arbitrario que contiene el *ModeloImagen*.

Internamente, para poder representar gráficamente tanto la imagen contenida en el *ModeloImagen* como elementos adicionales tales como la ROI, el polígono arbitrario o el factor de escalado, la clase *VistaImagen* hace uso de la clase *PanelImagenInteractivo*. La clase *PanelImagenInteractivo* no es más que una clase auxiliar que facilita la implementación de la clase *VistaImagen*. En última instancia, la clase *PanelImagenInteractivo* contiene un *PanelImagenConGraficos*, usado para representar todos los componentes gráficos adicionales requeridos por el *ModeloImagen*.

6.3.3. ConjuntoModeloImagen

La clase *ConjuntoModeloImagen* representa el conjunto de imágenes abiertas, actualmente, en la aplicación. Dado que el sistema es capaz de mantener abiertas varias imágenes de forma simultanea, se requiere de una estructura que sea capaz de coordinarlas. La clase *ConjuntoModeloImagen* representa un conjunto de objetos de tipo *ModeloImagen*. La clase *ConjuntoModeloImagen* permite, fundamentalmente, añadir o eliminar imágenes (objetos de tipo *ModeloImagen*) a la aplicación, así como acceder a las ya existentes, para que sean modificadas en caso de ser necesario.

6.3.4. VistaConjuntoModeloImagen

La clase *VistaConjuntoModeloImagen* es la representación gráfica del *ConjuntoModeloImagen*. Un objeto de la clase *ConjuntoModeloImagen* es representado gráficamente mediante un objeto del tipo *VistaConjuntoModeloImagen*. En nuestro caso, la clase *VistaConjuntoModeloImagen* no es una clase de tipo *vista* convencional: no representa un panel o contenedor gráfico donde se representa gráficamente un cierto modelo de datos. El problema principal a la hora de diseñar el cómo representar las imágenes de la aplicación es el de coordinar la visualización de imágenes con la arquitectura que provee el RCP. Toda imagen es abierta en un *EditorPart* del RCP. La clase *VistaConjuntoModeloImagen* coordina la apertura de un *EditorPart* por cada una de las imágenes que hay en el *ConjuntoModeloImagen*: si se añade una nueva imagen al *ConjuntoModeloImagen*, la *VistaConjuntoModeloImagen* abre un nuevo editor que contiene la imagen; si se elimina una imagen del *ConjuntoModeloImagen*, la *VistaConjuntoModeloImagen* cierra el editor asociado a dicha imagen.

La clase *VistaConjuntoModeloImagen* no abre exactamente editores de tipo *EditorPart*, sino editores de tipo *EditorImagen*, los cuales heredan de *EditorPart*. Un editor de tipo *EditorImagen* simplemente contiene un objeto de tipo *VistaImagen*, el cual visualiza un *ModeloImagen*. Concretamente, cuando se inserta un nuevo *ModeloImagen* en el *ConjuntoModeloImagen* (es decir, cuando se añade una nueva imagen a la aplicación), la *VistaConjuntoModeloImagen* abre un *EditorImagen*, el cual recibe, para su inicialización, el *ModeloImagen* insertado. Dicho *EditorImagen* crea un objeto de tipo *VistaImagen*, que muestra como contenido del editor.

Debe tenerse en cuenta que los objetos de tipo *EditorPart*, y por tanto también el *EditorImagen*, requieren ser inicializados mediante un objeto de tipo *IEditorInput* en su construcción, cuando son abiertos a través del método *IWorkbenchWindow.openPage(IEditorInput input, String editorId)*. Mediante el *IEditorInput* que se le pasa al *EditorImagen*, éste es capaz de obtener la referencia al *ModeloImagen* que necesita para la creación de la *VistaImagen* que contiene. Concretamente, la clase *EditorInputImagen*, la cual implementa la interfaz *IEditorInput*, contiene el identificador del *ModeloImagen*, a partir del cual es fácil obtener una referencia a éste. Cuando se abre un *EditorImagen* desde la *VistaConjuntoModeloImagen*, el método *openPage()* antes mencionado recibe como argumento un *EditorInputImagen* que permite obtener una referencia al *ModeloImagen* que se visualizará en el editor mediante la *VistaImagen*.

6.3.5. ControladorImagenes

La clase *ControladorImagenes* es la encargada de inicializar el *ConjuntoModeloImagen* global de la aplicación así como de la *VistaConjuntoModeloImagen* que se encarga de representar dicho *ConjuntoModeloImagen*. El *ControladorImagenes* es una clase que sigue el patrón *singleton*, es decir, sólo existe una instancia de dicha clase. La única instancia del *ControladorImagenes* es accesible mediante un método estático de dicha clase. A través de esa instancia, se puede acceder al *ConjuntoModeloImagen* global de la aplicación, aquél que contiene todas las imágenes actualmente abiertas. Recuérdese que este controlador es un *controlador de instanciación*, es decir, se trata de un controlador que simplemente inicializa y sincroniza el *ConjuntoModeloImagen* con la *VistaConjuntoModeloImagen*.

6.4. Opciones de interactividad de imágenes

La aplicación permite diversos tipos de interacciones con las imágenes que mantiene abiertas. Así por ejemplo, la aplicación permite aplicar todo tipo de operaciones (de tratamiento o de caracterización) a las imágenes.

Existe otro tipo de interacciones que el usuario puede llevar a cabo con las imágenes. Son interacciones más corrientes, como pueden ser la ampliación de una imagen (seleccionando, por ejemplo, una región rectangular a escalar) o el desplazamiento sobre ésta (como puede ser arrastrando el ratón para visualizar la región en la que estemos interesados).

Estas opciones de **interactividad** pueden dividirse en dos grupos: aquellas que no necesitan de la interacción del usuario con el panel gráfico de la imagen, y aquellas que sí lo requieren. Un ejemplo del primer grupo sería el clásico zoom de imagen, el cual, sin especificar la región a ampliar o reducir, amplia o reduce la imagen (mediante la simple pulsación de un botón, por ejemplo). Un ejemplo del segundo grupo podría ser el escalado de una región rectangular de la imagen, ya que el usuario debe definir, sobre el panel gráfico, la región rectangular a escalar.

El primer grupo, aquél que no requiere interacción del usuario sobre el panel gráfico, es sensiblemente más fácil de diseñar que el segundo. Cuando no se requiere interacción por parte del usuario, la estrategia a seguir es bien sencilla: la ejecución de esa opción de interactividad, por ejemplo mediante la pulsación de un botón de la interfaz gráfica, desencadenará una modificación del *ModeloImagen* asociado a la imagen actualmente seleccionada; esta modificación será inmediatamente recogida por la *VistaImagen* gracias al patrón *Observer*, reflejando dicho cambio. Por ejemplo, si se presiona un botón que aumenta el factor de zoom de la imagen, simplemente se cambiará el factor de zoom del *ModeloImagen*, y dicho cambio será visualizado por la *VistaImagen*, que mostrará la imagen ampliada.

El segundo grupo, el que requiere de la interacción del usuario sobre el panel gráfico, entraña una serie de problemas de diseño. Dependiendo de la opción de interactividad a llevar a cabo, el modo de interaccionar con el panel gráfico varía. Así por ejemplo, la opción de escalar una región rectangular de la imagen requiere que el usuario defina, sobre el panel de la imagen, la región a escalar: pulsando el botón izquierdo del ratón y arrastrándolo, se define la región rectangular; cuando el botón izquierdo es soltado, la región rectangular seleccionada es escalada, de modo que se ajusta hasta ocupar toda el panel de la imagen. Otra opción de interactividad distinta que requiere la interacción del usuario es, por ejemplo, la de extraer una región de interés (ROI) de una imagen. Dicha opción de interactividad permite al usuario seleccionar una región arbitraria de la imagen, la cual es extraída, y con la cual se crea una nueva imagen compuesta únicamente por la ROI seleccionada. Dicha región es seleccionada, punto a punto, mediante pulsaciones del botón derecho del ratón; cuando se acaba de definir la región, una pulsación del botón izquierdo lleva a cabo la extracción de la región y la creación de la nueva imagen. Como puede verse, existen diferentes métodos de interaccionar con el panel gráfico de la imagen. Cada uno de dichos métodos es, generalmente, independiente del otro, de modo que si se está interaccionando de un cierto modo (por ejemplo, seleccionando un área rectangular a ampliar), no puede interaccionarse de un modo distinto (por ejemplo, seleccionando la ROI).

Esta sección describe cómo están diseñados estos dos grupos de opciones de interactividad. Al primer grupo lo llamaremos *opciones de interactividad simples*, y al segundo, *opciones de interactividad complejas*.

6.4.1. Opciones de interactividad simples

Este grupo está formado por aquellas acciones que pueden ejecutarse sin necesidad de la interacción del usuario con el panel gráfico en el que se representa la imagen. Dentro de este grupo se encuentran las siguientes opciones de interactividad:

- Aumentar factor de zoom de la imagen: será implementada como un simple botón situado en la cool bar de la aplicación. Una pulsación de dicho botón supondrá el aumento del factor de zoom de la imagen actualmente seleccionada. El grado en que el factor de zoom es aumentado es pequeño, y está regido por la clase *ListadoFactoresZoom*. Implementada por la clase *AccionAmpliarImagen*.

- Reducir factor de zoom de la imagen: será implementada como un simple botón situado en la cool bar de la aplicación. Una pulsación de dicho botón supondrá la reducción del factor de zoom de la imagen actualmente seleccionada. El grado en que el factor de zoom es reducido es pequeño, y está regido por la clase *ListadoFactoresZoom*. Implementada por la clase *AccionReducirImagen*.

- Reestablecer factor de zoom original de la imagen: será implementada como un simple botón situado en la cool bar de la aplicación. Una pulsación de dicho botón supondrá el reestablecimiento del factor de zoom original de la imagen, permitiendo así que ésta sea visualizada a su tamaño real. Implementada por la clase *AccionReestablecerDimensionesImagen*.

- Eliminar región de interés: será implementada como un simple botoón en la cool bar de la aplicación Una pulsación de dicho botón eliminará la región de interés de la imagen. Está implementada por la clase *AccionEliminarROI*.

6.4.1.1. AccionAmpliarImagen

La clase *AccionAmpliarImagen* representa la acción que permite aumentar el factor de zoom de la imagen actualmente seleccionada. Esta clase hereda de la clase *AccionEditorDelegate*. Esta acción es añadida a la cool bar de la aplicación, de modo que la cool bar presenta dicha acción como un simple botón. Cuando el botón es pulsado, la clase *AccionAmpliarImagen* aumenta el factor de zoom del *ModeloImagen* asociado al *EditorImagen* actualmente activo en la aplicación. Al ser modificado el factor de zoom del *ModeloImagen*, la *VistaImagen* asociada es notificada, haciendo que, en efecto, la imagen representada haya cambiado de tamaño.

6.4.1.2. AccionReducirImagen

La clase *AccionReducirImagen* representa la acción que permite reducir el factor de zoom de la imagen actualmente seleccionada. Esta clase hereda de la clase *AccionEditorDelegate*. Esta acción es añadida a la cool bar de la aplicación, de modo que la cool bar presenta dicha acción como un simple botón. Cuando el botón es pulsado, la clase *AccionReducirImagen* disminuye el factor de zoom del *ModeloImagen* asociado al *EditorImagen* actualmente activo en la aplicación. Al ser modificado el factor de zoom del *ModeloImagen*, la *VistaImagen* asociada es notificada, haciendo que, en efecto, la imagen representada haya cambiado de tamaño.

6.4.1.3. AccionReestablecerDimensionesImagen

La clase *AccionReestablecerDimensionesImagen* representa la acción que permite reestablecer el factor de zoom de la imagen actualmente seleccionada a su valor original. Esta clase hereda de la clase *AccionEditorDelegate*. Esta acción es añadida a la cool bar de la aplicación, de modo que la cool bar presenta dicha acción como un simple botón. Cuando el botón es pulsado, la clase *AccionReestablecerDimensionesImagen* reestablece el valor por defecto del factor de zoom del *ModeloImagen* asociado al *EditorImagen* actualmente activo en la aplicación. Al ser modificado el factor de zoom del *ModeloImagen*, la *VistaImagen* asociada es notificada, haciendo que, en efecto, la imagen representada haya cambiado de tamaño.

6.4.1.4. AccionEliminarROI

La clase *AccionEliminarROI* representa la acción que permite eliminar la ROI de la imagen actualmente seleccionada. Esta clase hereda de la clase *AccionEditorDelegate*. Esta acción es añadida a la cool bar de la aplicación, de modo que la cool bar presenta dicha acción como

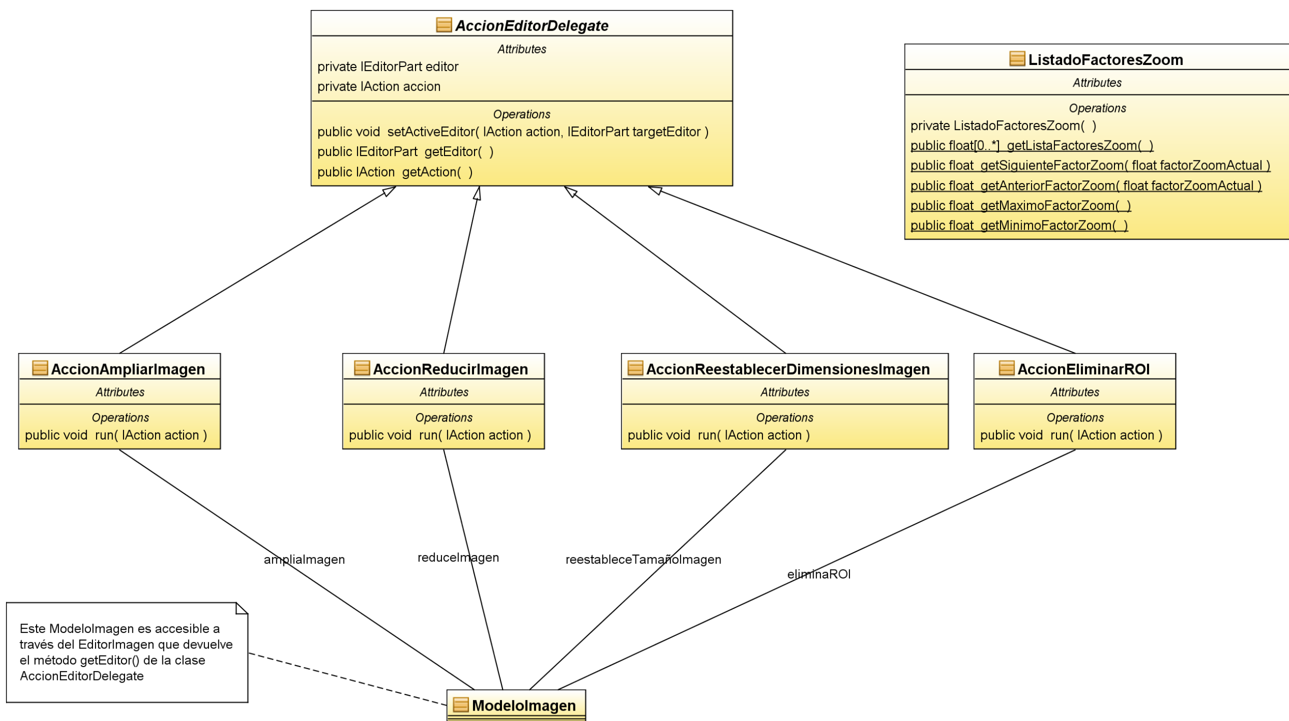

Figura 6.5: Diagrama de clases de las opciones de interactividad que no requieren interacción del usuario con el panel gráfico de la imagen

un simple botón. Cuando el botón es pulsado, la clase *AccionEliminarROI* elimina la ROI del *ModeloImagen* asociado al *EditorImagen* actualmente activo en la aplicación. Al ser eliminada la ROI *ModeloImagen*, la *VistaImagen* asociada es notificada, haciendo que, en efecto, desaparezca la ROI que hubiera definida sobre la imagen.

La figura 6.5 representa el diagrama de clases de este primer grupo de opciones de interactividad.

6.4.2. Opciones de interactividad complejas

Este grupo de opciones de interactividad recoge las que requieren interacción del usuario con el panel gráfico donde se visualiza la imagen. Dentro de de este grupo, se encuentran las siguientes funciones:

- Escalar una región rectangular de la imagen (ajuste de imagen): esta opción de interactividad, implementada como un *check button* en la cool bar de la aplicación, permite al usuario elegir un área rectangular de la imagen, la cual será ajustada hasta ocupar todo el área de visualización de imagen de la interfaz gráfica. Esta opción representa el clásico mecanismo que permite al usuario seleccionar un área rectangular que quiere visualizar con mayor detalle. La clase *AccionAjustarImagen* implementa esta opción de interactividad. Cuando el usuario activa el *check button* asociado, el panel gráfico permite definir sobre él el área rectangular que se quiere escalar.

- Desplazarse a través de la imagen: esta opción de interactividad, implementada como un *check button* en la cool bar de la aplicación, permite al usuario desplazarse por la imagen en caso de que ésta no quepa completamente en el área de visualización de la imagen. Gracias a esta acción, el usuario puede hacer uso del clásico mecanismo de *dragueo*, con el cual, manteniendo pulsado un botón del ratón, y arrastrándolo, se produce el desplazamiento de la imagen. Esta opción de interactividad está implementada mediante

la clase *AccionArrastrarImagen*. Cuando el usuario activa el *check button* asociado, el panel gráfico permite arrastrar la imagen.

- Extraer región de interés: esta opción de interactividad está implementada mediante un *check button* situado en la cool bar. Cuando el usuario activa dicho *check button*, el panel gráfico de la imagen permite definir sobre él una región arbitraria mediante pulsaciones del botón derecho del ratón. Cuando el usuario ha acabado de definir la región, pulsa el botón derecho del ratón, desencadenándose así la creación de una nueva imagen que contiene sólo la ROI antes señalada. La clase *AccionExtraerROI* implementa esta opción de interactividad.

- Definir región de interés: esta opción de interactividad está implementada mediante un *check button* situado en la cool bar. Cuando el usuario activa dicho *check button*, el panel gráfico de la imagen permite definir sobre él una región arbitraria mediante pulsaciones del botón derecho del ratón. Cuando el usuario ha acabado de definir la región, pulsa el botón derecho del ratón, tras lo cual se crea una ROI con la forma del área recién definida. Esta ROI es asociada a la imagen. La clase *AccionDefinirROI* implementa esta opción de interactividad.

- Substraer región de interés: esta opción de interactividad está implementada mediante un *check button* situado en la cool bar. Cuando el usuario activa dicho *check button*, el panel gráfico de la imagen permite definir sobre él una región arbitraria mediante pulsaciones del botón derecho del ratón. Cuando el usuario ha acabado de definir la región, pulsa el botón derecho del ratón, tras lo cual el área seleccionada es substraída de la ROI que hubiera definida en la imagen. La clase *AccionSubstraerROI* implementa esta opción de interactividad.

Hasta ahora parece fácil decir que las clases anteriores *implementan* la funcionalidad descrita, pero lo cierto es que ellas sólo son las encargadas de señalizar que dichas opciones están actualmente activas.

Parece evidente que la clase encargada de gestionar estas opciones de interactividad debería ser la clase *VistaImagen*. La clase *VistaImagen* es la que representa visualmente a una imagen, es decir, representa el panel donde se visualiza la imagen; por tanto, si una opción de interactividad requiere de la interacción del usuario con el panel gráfico, deberá ser la clase *VistaImagen* la encargada de gestionar dicha interacción.

Debe recordarse que la clase *VistaImagen* contiene un objeto de tipo *PanelImagenInteractivo*. La clase *PanelImagenInteractivo* es la que realmente lleva a cabo la representanción de la imagen; la *VistaImagen* no es más que un contenedor que lo abstrae[5]. El *PanelImagenInteractivo* es, por tanto, la clase que realmente debe gestionar todas las opciones de interactividad anteriormente mencionadas. Cuando se crea un objeto de *VistaImagen*, éste, internamente, crea un objeto de tipo *PanelImagenInteractivo*.

El *PanelImagenInteractivo* es un panel gráfico (*JComponent*) que permite al usuario interaccionar con él mediante ciertas opciones de interactividad, como las anteriormente mencionadas. El *PanelImagenInteractivo* permite que sobre él se defina un área rectangular que representa una región a escalar, que se defina una cierta región de interés a extraer de la imagen, etc.

¿Cómo sabe el *PanelImagenInteractivo* qué opción de interactividad está activa en cada momento? El *PanelImagenInteractivo* contiene una instancia de un objeto que implementa la

[5] ¿Y para qué abstraer la clase *PanelImagenInteractivo*, si es ésta la que representa a nivel visual la imagen? Esta es una buena pregunta; el problema radica en que el *PanelImagenInteractivo* es un panel gráfico (*JComponent*) del AWT. Como la aplicación está basada en SWT, necesitamos construir una abstracción en forma de *Composite* del SWT, el cual sí pueda ser usado dentro de la aplicación.

interfaz *IGestorInteractividad*. La interfaz *IGestorInteractividad* define una serie de métodos que devuelven el tipo de opción de interactividad que está activa en cada momento. Por ejemplo, el método *arrastrarActivo()* determina si está activa la opción que permite arrastrar una imagen que no cabe totalmente en la pantalla. El método *seleccionarAreaActivo()* determina si está activa la opción que permite seleccionar un área arbitraria de la imagen para extraer, definir o substraer esa región de interés. Cuando el *PanelImagenInteractivo* recibe algún tipo de evento de ratón, consulta al *IGestorInteractividad* para determinar qué tipo de opción de interactividad está activa, y actuar en consecuencia. Si, por ejemplo, está activa la opción de seleccionar un área arbitraria de la imagen, sólo los eventos de ratón *pulsar botón derecho* y *pulsar botón izquierdo* serán recogidos por el *PanelImagenInteractivo* (esos dos eventos permitirán añadir puntos al área arbitraria o bien extraer la región de interés definida por dicho área). En general se puede decir que, cuando se encuentra activa una opción de interactividad, el *PanelImagenInteractivo* responde a los eventos de ratón asociados a dicha opción de interactividad. Consultando al objeto *IGestorInteractividad*, el panel puede determinar qué opciones de interactividad están, en efecto, activas.

La clase *GestorInteractividadImagenes* es la clase que implementa la interfaz *IGestorInteractividad*. Esta clase, definida según el patrón *singleton*, es la que recibe el *PanelImagenInteractivo* cuando es creado por la *VistaImagen*. Dicho *GestorInteractividadImagenes* es usado como objeto de tipo *IGestorInteractividad* para determinar qué opción de interactividad está activa en cada momento.

La clase *GestorInteractividadImagenes* debe proveer una implementación concreta que determine qué opciones de interactividad están activas en cada momento.

La clase *GestorInteractividadImagenes* define varios métodos que permiten activar o desactivar las diferentes opciones de interactividad. Por ejemplo, el método *activarArrastrarImagen()* permite activar o desactivar la opción de interactividad de arrastrado de imagen. El método *activarAjustarImagen()* permite activar la opción de selección de un área rectangular a escalar de la imagen. Él estado devuelto por los métodos de la interfaz *IGestorInteractividad* varía acorde a las llamadas de los métodos de activación o desactivación de las opciones de interactividad.

Las opciones de interactividad son incompatibles entre sí. Esto significa que no puede haber dos opciones de interactividad simultáneamente activas, y que por tanto activar una supone desactivar las demás.

El gran problema asociado a la gestión de estas opciones de interactividad es cómo se gestiona la activación y desactivación de los demás botones de las opciones de interactividad. Supongamos que, por ejemplo, está activa la opción de interactividad de arrastrado de imagen, y que el usuario selecciona la opción de interactividad de substracción de ROI. Al activarse ésta última, debería desactivarse la opción de arrastrado de imagen.

Para solucionar este problema, a la clase *GestorInteractividadImagenes* se le da la posibilidad de registrar *listeners* en ella. Permite, concretamente, registrar objetos de tipo *IPropertyChangeListener*. Cuando se llama a alguno de los métodos de activación de opciones de interactividad (como por ejemplo *activarArrastrarImagen()*), la clase *GestorInteractividadImagenes* dispara un evento (*PropertyChangeEvent*) en cada uno de los *listeners* registrados. Este evento contiene información relativa al estado de activación de las opciones de interactividad **antes** y **después** de la ejecución de la función.

Mediante este mecanismo es fácil controlar la activación y desactivación de los botones que permiten al usuario seleccionar las opciones de interactividad. Lo que realmente se hace es que las clases que gestionan estas opciones de interactividad, *AccionArrastrarImagen*, *AccionExtraerROI*, *AccionDefinirROI*, *AccionSubstraerROI*, además implementan la interfaz *IPropertyChangeListener*. Estas acciones son registradas como *listeners* en el *GestorInteractividadImagenes*. Así, siempre que cambian las opciones de interactividad en el *GestorInteractividadImagenes*, las acciones son notificadas de ello, permitiendo así que éstas mantengan un

estado de activación-desactivación consistente en los botones que tienen asociados.

Es importante mencionar que el *PanelImagenInteractivo* hace uso de una clase auxiliar, el *MouseAdapterInteractividad*, para controlar todos los eventos de ratón generados sobre él, y filtrarlos según la opción de interactividad actualmente activa.

6.4.2.1. AccionAjustarImagen

La clase *AccionAjustarImagen* representa la acción que permite activar la opción de selección de un área rectangular a escalar de la imagen. Esta clase hereda de la clase *AccionEditorDelegate*, implementa la interfaz *IPropertyChangeListener*, y está implementada como un *check button*, de cuyo estado depende considerar si la opción de selección de área a escalar está activa (botón pulsado), o no (botón despulsado). Cuando el botón es pulsado, la acción invoca al método *activarAjustarImagen()* del *GestorInteractividadImagenes*.

6.4.2.2. AccionArrastrarImagen

La clase *AccionArrastrarImagen* representa la acción que permite arrastrar la imagen (desplazarse por ella), en caso de que no quepa completamente en la pantalla. Esta clase hereda de la clase *AccionEditorDelegate*, implementa la interfaz *IPropertyChangeListener*, y está implementada como un *check button*, de cuyo estado depende considerar si la opción de arrastrado de imagen está activa (botón pulsado), o no (botón despulsado). Cuando el botón es pulsado, la acción invoca al método *activarArrastrarImagen()* del *GestorInteractividadImagenes*.

6.4.2.3. AccionExtraerROI

La clase *AccionExtraerROI* representa la acción que permite seleccionar un área arbitraria de la imagen, y extraer de dicha región de interés en otra nueva imagen. Esta clase hereda de la clase *AccionEditorDelegate*, implementa la interfaz *IPropertyChangeListener*, y está implementada como un *check button*, de cuyo estado depende considerar si la opción de extracción de ROI está activa (botón pulsado), o no (botón despulsado). Cuando el botón es pulsado, la acción invoca al método *activarExtraerAreaImagen()* del *GestorInteractividadImagenes*.

6.4.2.4. AccionDefinirROI

La clase *AccionDefinirROI* representa la acción que permite definir una ROI sobre una imagen. Esta clase hereda de la clase *AccionEditorDelegate*, implementa la interfaz *IPropertyChangeListener*, y está implementada como un *check button*, de cuyo estado depende considerar si la opción de extracción de ROI está activa (botón pulsado), o no (botón despulsado). Cuando el botón es pulsado, la acción invoca al método *activarDefinirAreaImagen()* del *GestorInteractividadImagenes*.

6.4.2.5. AccionSubstraerROI

La clase *AccionSubstraerROI* representa la acción que permite substraer una región de la ROI que haya definida en una imagen. Esta clase hereda de la clase *AccionEditorDelegate*, implementa la interfaz *IPropertyChangeListener*, y está implementada como un *check button*, de cuyo estado depende considerar si la opción de extracción de ROI está activa (botón pulsado), o no (botón despulsado). Cuando el botón es pulsado, la acción invoca al método *activarSubstraerAreaImagen()* del *GestorInteractividadImagenes*.

6.4.2.6. IGestorInteractividad

Esta interfaz define una serie de métodos que permiten consultar qué opción de interactividad está actualmente activa. Si una cierta clase necesita hacer un seguimiento de qué opciones de interactividad están activas, puede hacer uso de un objeto que implemente esta interfaz.

Es importante notar que la opción de interactividad de selección de un área arbitraria sobre la imagen puede estar activada debido a diversas causas. Por ejemplo, si se está extrayendo una ROI, dicha opción de interactividad está activa, pero también si se está definiendo o substrayendo una ROI dicha opción está activa. Para distinguir cuál es la causa subyacente a la activación de la opción de interactividad de selección de área, la interfaz proporciona el método*tipoSeleccionAreaActivo()*.

6.4.2.7. GestorInteractividadImagenes

Esta clase, la cual implementa la interfaz *IGestorInteractividad*, es la clase que, a nivel de aplicación, determina que opción de interactividad está activada.

Esta clase, además de determinar qué opciones de interactividad están activas, se encarga de coordinar dichas opciones entre sí. Las opciones de interactividad, en general, no son compatibles de forma simultanea, es decir, no puede haber más de una simultaneamente activa. La clase *GestorInteractividadImagenes* se encarga de coordinarlas, de modo que, en todo momento, sólo hay opciones de interactividad que son mutuamente compatibles. Cuando se activa una opción de interactividad, el *GestorInteractividadImagenes* dispara un evento de tipo *PropertyChangeEvent* en todos los *listeners* que tiene registrados. El evento recoge el cambio en el estado de las opciones de interactividad. Los *listeners*, por otro lado, son justamente las acciones que controlan las opciones de interactividad, de modo que son notificadas del cambio producido, pudiendo actualizar el estado de los *check buttons* acorde al nuevo estado de las opciones de interactividad.

La clase *GestorInteractividadImagenes* está implementada siguiendo el patrón *singleton*.

6.4.2.8. PanelImagenInteractivo

La clase *PanelImagenInteractivo* es el panel gráfico (hereda de *JComponent*) que permite interaccionar con él según distintas opciones de interactividad. El *PanelImagenInteractivo* almacena una referencia a un objeto de tipo *IGestorInteractividad* (concretamente, almacena una referencia al *GestorInteractividadImagenes* de la aplicación). Dicho objeto es usado para determinar cómo el usuario es capaz de interaccionar con el panel que visualiza a la imagen. Dependiendo de la opción de interactividad activa en cada momento, la clase *PanelImagenInteractivo* permite realizar diferentes funciones sobre él. Para ello, hace uso de una clase auxiliar, a saber, el *MouseAdapterInteractividad*. Éste *MouseAdapter* se encarga de filtrar e interpretar los eventos de ratón producidos por el usuario dependiendo de la opción de interactividad activa en cada momento.

Es importante notar que, dado que este panel permite modificar el *ModeloImagen* (por ejemplo, cuando se escala una región rectangular de la imagen, que implica un aumento del factor de zoom del *ModeloImagen*), debe tener una referencia a éste.

La figura 6.6 representa el diagrama de clases de este segundo grupo de opciones de interactividad.

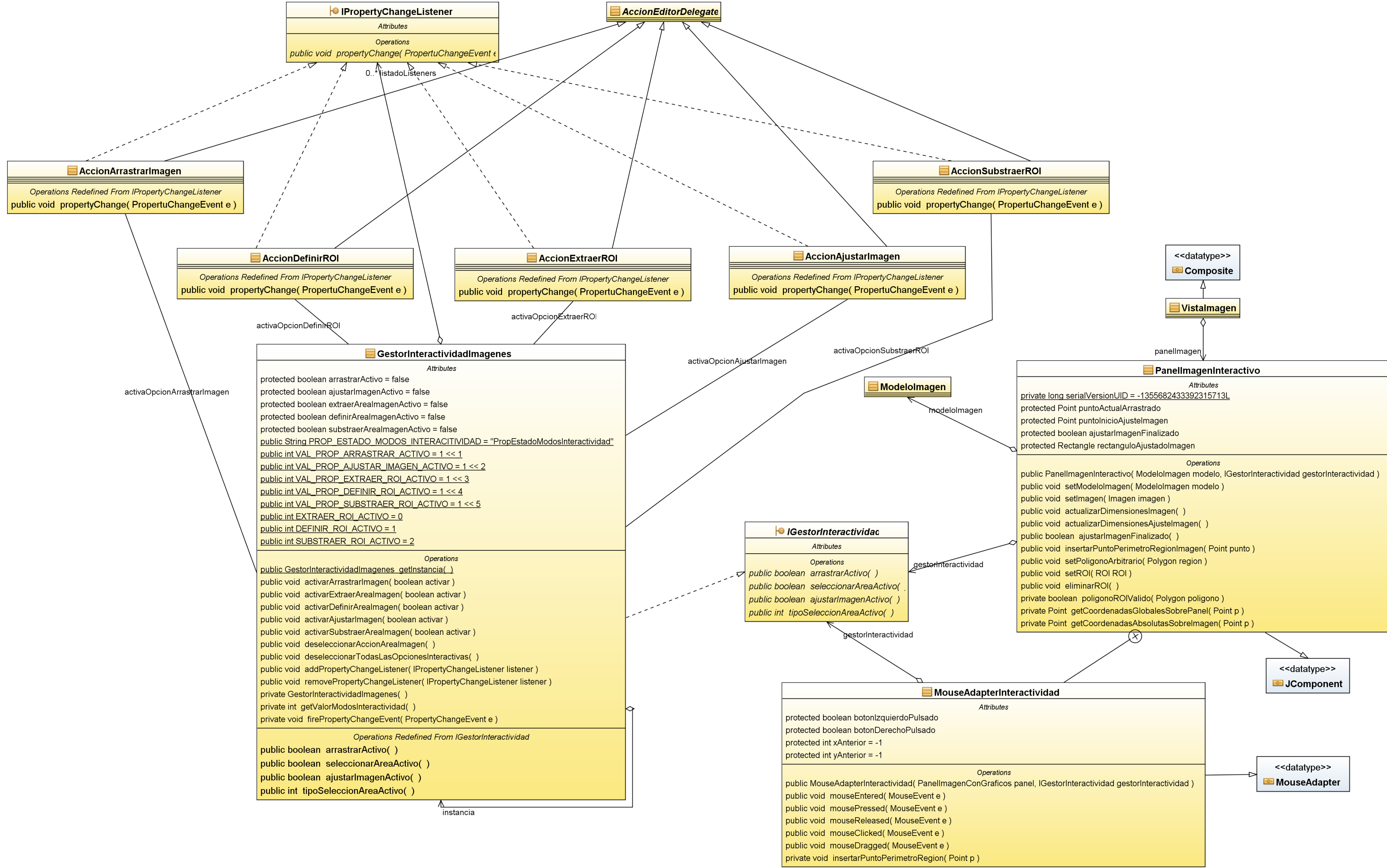

Figura 6.6: Diagrama de clases de las opciones de interactividad que sí requieren interacción del usuario con el panel gráfico de la imagen

6.5. Punto de extensión *operationApplication* e interfaz *IOperadorAplicacion*

El punto de extensión *operationApplication* permite añadir nuevas operaciones al sistema. Para contribuir a este punto de extensión, es necesario implementar la interfaz *IOperadorAplicacion*. Toda operación dentro de Cool Imaging, ya sea de tratamiento o caracterización de imágenes, está representada por una clase que implementa la interfaz *IOperadorAplicacion*. Así, para que un *plugin* externo al *plugin coolimaging* aporte nuevas operaciones, deberá aportar clases que implementen dicha interfaz.

La mayor parte de esta sección estará enfocada a explicar de forma detallada el funcionamiento de la interfaz *IOperadorAplicacion*, así como cómo se cargan, mediante el mecanismo de puntos de extensión de Eclipse RCP, aquellas clases que implementan la interfaz *IOperadorAplicacion*

6.5.1. Interfaz *IOperadorAplicacion*

La interfaz *IOperadorAplicacion* representa la información mínima requerida para la gestión de una operación en Cool Imaging. A través de esta interfaz, el *plugin coolimaging* sabe cómo hacer uso de la operación que la implementa.

A nivel abstracto podría decirse que una operación se caracteriza por una serie de parámetros entrada y por un algoritmo de ejecución de la operación.

En Cool Imaging, el uso de toda operación sigue la misma filosofía: especificar los parámetros requeridos por la operación, y ejecutar la operación haciendo uso de dichos parámetros. Los parámetros son especificados, principalmente, en un panel gráfico que se le muestra al usuario en pantalla.

Por ejemplo, si el usuario quisiera aplicar una operación de rotación, la aplicación le mostraría un panel donde podría seleccionar parámetros como el ángulo de rotación, el algoritmo de interpolación usado para corregir los defectos producidos por la rotación, o la imagen a la que aplicar la rotación en sí. Tras especificar los parámetros de la operación, el sistema estaría capacitado para llevarla a cabo en cualquier momento.

La interfaz *IOperadorAplicacion* está diseñada con esta filosofía en mente.

A grandes rasgos, el método *getPanelInfoOperador()* es el encargado de crear el panel gráfico donde el usuario introduce los parámetros de la operación. Por otro lado, el método *operar()* es el encargado de, haciendo uso de los parámetros especificados, llevar a cabo la operación en sí.

El método *getPanelInfoOperador()* devuelve un objeto de la clase *PanelInfoOperador*. La clase *PanelInfoOperador* cobra especial importancia dentro de Cool Imaging, ya que representa un panel gráfico con la capacidad de proporcionar parámetros para la ejecución de operaciones. Antes explicar cómo el panel es capaz de proporcionar los parámetros para la ejecución de una operación, es necesario describir cómo se representan parámetros de ejecución de una operación. Por *parámetro de ejecución* se entiende cualquier dato que la operación necesita para ser llevada a cabo. Por ejemplo, en el caso de la operación de rotación, un parámetro sería el ángulo de rotación, y otro sería la imagen a rotar (a parte, por supuesto, podría haber más parámetros).

La clase *ParametroOperador* representa un parámetros de ejecución de una operación. Un *ParametroOperador* consta de un **identificador** y de un **valor**. El identificador del parámetro define, de algún modo, la semántica del parámetro. Por ejemplo, en el contexto de la operación de rotación, el *ParametroOperador* asociado al ángulo de rotación podría tener un valor de 34 (34 grados a rotar), y un identificador *ID_PARAMETRO_ANGULO*. Por otro lado, el parámetro asociado a la imagen a rotar podría tener, de valor, un objeto de tipo *Imagen* (imagen a rotar), y su identificador podría ser *ID_PARAMETRO_IMAGEN*. Cada operación debe encargarse de asociar un determinado identificador a cada uno de sus parámetros y que, además, dichos

parámetros **sean distintos entre sí**. En el contexto de una operación concreta no puede haber dos parámetros con el mismo identificador, ya que el identificador es usado, por la operación, para referenciar a cada uno de los parámetros a usar. Si varios parámetros compartieran el mismo identificador, la operación no tendría manera de distinguir entre ellos, y no podría operar correctamente.

Una operación requiere de uno o varios parámetros para poder ejecutarse correctamente. Es necesario, pues, crear una clase que pueda gestionar varios parámetros de forma simultanea. La clase *ConjuntoParametroOperador* representa un conjunto de parámetros asociados a una operación determinada. Esta clase permite añadir parámetros (*ParametroOperador*) al conjunto de parámetros, así como consultarlos y eliminarlos. La clase *ConjuntoParametroOperador* permite acceder a los parámetros que almacena a través de un mecanismo de consulta por **identificador de parámetro**. Así, a un objeto de tipo *ConjuntoParametroOperador* se le pueden añadir objetos de tipo *ParametroOperador*, los cuales pueden ser consultados a través de sus identificadores de parámetro.

Por ejemplo, en el caso de la operación de rotación, se podría crear un *ConjuntoParametroOperador* que almacenará los dos parámetros de la rotación:

- Un objeto *ParametroOperador* con **identificador** *ID_PARAMETRO_ANGULO*, y cuyo **valor** fuera, por ejemplo, 37 (37 grados a rotar la imagen). Este *ParametroOperador* representaría el ángulo de rotación.

- Un objeto *ParametroOperador* con **identificador** *ID_PARAMETRO_IMAGEN*, y cuyo **valor** fuera un determinado objeto de tipo *Imagen*. Este *ParametroOperador* representaría la imagen a rotar.

Posteriormente, cuando la operación de rotación fuera a ejecutarse, podría acceder a los parámetros que requiriese. Al *ConjuntoParametroOperador* le podría pedir el valor del parámetro de identificador *ID_PARAMETRO_ANGULO*, obteniendo así el ángulo de rotación. Por otro lado, también le podría pedir el valor del parámetro de identificador *ID_PARAMETRO_IMAGEN*, obteniendo de ese modo la imagen a rotar. Tras ello, la operación de rotación estaría en disposición de poder ejecutarse, y rotar la imagen el ángulo especificado.

La clase *PanelInfoOperador* es un panel gráfico de la biblioteca SWT (*Composite*), con la capacidad de proporcionar parámetros de operación. De ello se encarga el método abstracto *public abstract void getParametros(ConjuntoParametroOperador parametros)*, cuya finalidad es la de insertar en el objeto *parametros* de entrada todos aquellos parámetros (*ParametroOperador*) que puedan obtenerse a partir de la información introducida en el panel gráfico.

Por ejemplo, podría existir un *PanelInfoOperador* asociado a la operación de rotación, y que permitiera al usuario introducir el ángulo a rotar la imagen. Dicho panel podría contener un simple campo de texto que permitiera introducir el ángulo de rotación. El método *getParametros()* de dicho *PanelInfoOperador* debería encargarse de insertar, en el objeto *ConjuntoParametroOperador* de entrada, un objeto *ParametroOperador* conteniendo el ángulo de rotación. Dicho objeto podría tener un identificador como el descrito con anterioridad (*ID_PARAMETRO_ANGULO*). El panel podría permitir especificar otros parámetros de la operación. En ese caso, su método *getParametros()* también debería encargarse de insertar, en el objeto *ConjuntoParametroOperador* de entrada, él resto de parámetros.

El método *getPanelInfoOperador()* de la interfaz *IOperadorAplicacion* debe devolver un objeto *PanelInfoOperador* tal que su método *getParametros()* devuelva todos los parámetros requeridos por el *IOperadorAplicacion* para poder ejecutar la operación, **a excepción** de los parámetros que almacenan la imagen (o imágenes) de entrada sobre la que aplicar la operación. Esto significa, por ejemplo, que en el caso de la operación de rotación, el método *getPanelInfoOperador()* debería devolver un *PanelInfoOperador* que sólo permitiera extraer los parámetros

asociados al ángulo de rotación o al método de interpolación empleado para corregir el error de la rotación, pero dicho panel **no podría** obtener el parámetro que representase la imagen a rotar.

El conjunto de todos los parámetros que necesita una operación para operar, a excepción de las imágenes de entrada, se conoce como **parámetros fundamentales**.

A través del panel devuelto por el método *getPanelInfoOperador()*, el *plugin coolimaging* es capaz extraer los parámetros fundamentales que la operación requiere para operar.

Ahora bien, a través del *PanelInfoOperador* no se extraen todos los parámetros que la operación necesita para operar. Resta, de algún modo, extraer las imágenes sobre las que aplicar la operación. Si, por ejemplo, la operación fuera de rotación, habría que extraer de algún sitio el parámetro que representase la imagen a rotar. Si la operación fuera una *and* lógica entre dos imágenes, habría que extraer dos parámetros, uno por cada imagen que interviniese en la *and*.

Estos parámetros, conocidos como parámetros de **tipo imagen**, son proporcionados por el *plugin coolimaging*. A diferencia de los parámetros fundamentales, que son construidos directamente por el *PanelInfoOperador*, los parámetros de tipo imagen deben ser construiros por el *plugin coolimaging*. Para ello, la operación debe especificar, de algún modo, cuántas imágenes de entrada necesita para operar, así como cuáles son los identificadores de los parámetros asociados a dichas imágenes de entrada.

Por ejemplo, en el caso de la operación de rotación, el *IOperadorAplicacion* debería indicar que la operación requiere una única imagen (la imagen a rotar), y que el identificador de parámetro asociado a dicha imagen es *ID_PARAMETRO_IMAGEN*. Cuando se hiciera uso de la operación de rotación, el *plugin coolimaging* podría construir el parámetro con la imagen a rotar, basándose en esta información.

Para aportar esta información, la interfaz *IOperadorAplicacion* define el método *getInfoOperador()*.

El método *getInfoOperador()* devuelve un objeto de tipo *InfoOperadorAplicacion*. La clase *InfoOperadorAplicacion* contiene información adicional que el *plugin coolimaging* debe conocer para poder hacer un uso correcto de una operación. Parte de esta información es justamente la que indica cuántas imágenes necesita la operación para operar, así como los identificadores de parámetro asociados a cada una de dichas imágenes de entrada.

El objeto *InfoOperadorAplicacion* devuelto por el método *getInfoOperador()* contiene un método llamado *getInfoImagenes()*, el cual proporciona la información que especifica cuántas imágenes necesita, así como el identificador de cada una de ellas[6].

El método *insertarParametros()* de la interfaz *IOperadorAplicacion* es el encargado de registrar localmente los parámetros a usar en la operación. El método *insertarParametros()* recibe un *ConjuntoParametroOperador* de entrada, el cual representa todos los parámetros necesarios para ejecutar la operación. Antes de hacer uso de una operación determinada (es decir, antes de llamar al método *operar()* de la interfaz *IOperadorAplicacion*[7]), el *plugin coolimaging* llama al método *insertarParametros()*, cuyo *ConjuntoParametroOperador* de entrada contiene todos los parámetros necesarios para llevar a cabo la operación. El método *insertarParametros()* debe encargarse de almacenar localmente dichos parámetros de entrada para que posteriormente, en su método *operar()*, el *IOperadorAplicacion* pueda realizar la operación concreta con los parámetros recibidos.

El método *operar()* es el encargado de llevar a cabo la operación en sí. Antes de llamar a este método, siempre se llama al método *insertarParametros()*. De este modo se consigue que

[6]Dicho método devuelve realmente un objeto de tipo *ConjuntoInformacionImagenOperador*, el cual contiene toda la información asociada a las imágenes requeridas por la operación. Este objeto almacena objetos de tipo *InformacionImagenOperador*, cada uno representando la información de una de las imágenes requeridas por la operación: descripción e identificador de parámetro

[7]El método *operar()* es el que debe realizar los cálculos de la operación, haciendo uso de los parámetros necesarios.

el método *operar()* pueda tener acceso a los parámetros necesarios para realizar la operación. Recuérdese que el método *insertarParametros()* debe encargarse de almacenar los parámetros que recibe de entrada, con la idea de que el método *operar()*, en una posterior llamada, pueda hacer uso de ellos para así realizar la operación.

La dinámica general de funcionamiento de la clase *IOperadorAplicacion*, así como sus interacciones con el *plugin coolimaging*, es la siguiente:

1. El *plugin coolimaging* llama al método *getPanelInfoOperador()* de la interfaz *IOperadorAplicacion*. Así, permite mostrar el panel gráfico donde el usuario debería introducir los **parámetros fundamentales** de la operación. El *plugin coolimaging* llama al método *getParametros()* sobre este panel, obteniendo así los parámetros fundamentales de la operación.

2. El *plugin coolimaging* construye los parámetros de **tipo imagen**. Para ello, hace uso de la información que la interfaz *IOperadorAplicacion* devuelve al llamar a su método *getInfoOperador*. Para cada una de las imágenes que necesita la operación, se construye un nuevo parámetro.

3. El *plugin coolimaging* construye un objeto de tipo *ConjuntoParametroOperador*, el cual contiene los parámetros obtenidos en los pasos 1 y 2. Este objeto *ConjuntoParametroOperador* contiene **todos** los parámetros necesarios para la ejecución de la operación.

4. El *plugin coolimaging* llama al método *insertarParametros()* del *IOperadorAplicacion*, pasándole como argumento de entrada el objeto *ConjuntoParametroOperador* construido en el paso 3. El *IOperadorAplicacion*, en dicho método, se encarga de guardar una referencia a dicho objeto, para poder usar los parámetros con posterioridad.

5. El *plugin coolimaging* llama al método *operar()* del *IOperadorAplicacion*. El método *operar()* se encarga de realizar los cálculos de la operación haciendo uso de los parámetros que se registraron mediante el método *insertarParametros()*, en el paso 4.

6. El *plugin coolimaging* recoge el resultado del método *operar()*, y lo procesa como considere conveniente.

   El diagrama de secuencia de la figura 6.7 muestra las interacciones entre los distintos objetos involucrados en este proceso.

6.5.2. Punto de extensión *operationApplication*

Explicado el mecanismo de funcionamiento de la interfaz *IOperadorAplicacion*, es necesario conocer el mecanismo por el cual el *plugin coolimaging* es capaz de conocer cuáles clases, de los *plugins* conectados a través del punto de extensión *operationApplication*, implementan la interfaz *IOperadorAplicacion*. Esas clases son las operaciones a registrar en Cool Imaging, y las cuales se permite usar al usuario.

Dentro del *plugin coolimaging*, la clase *Configurador* se encarga de realizar esta labor. Haciendo uso de la arquitectura de *plugins* del Eclipse RCP, es capaz de localizar qué clases, conectadas a través del punto de extensión *operationApplication*, implementan la interfaz *IOperadorAplicacion*. En la sección 6.10 se da una descripción detallada de la clase *Configurador*.

6.6. Control de operaciones: menú de operaciones

En la sección 6.5 se dio una descripción detallada de cómo, a través del punto de extensión *operationApplication* así como de la interfaz *IOperadorAplicacion*, se pueden añadir nuevas

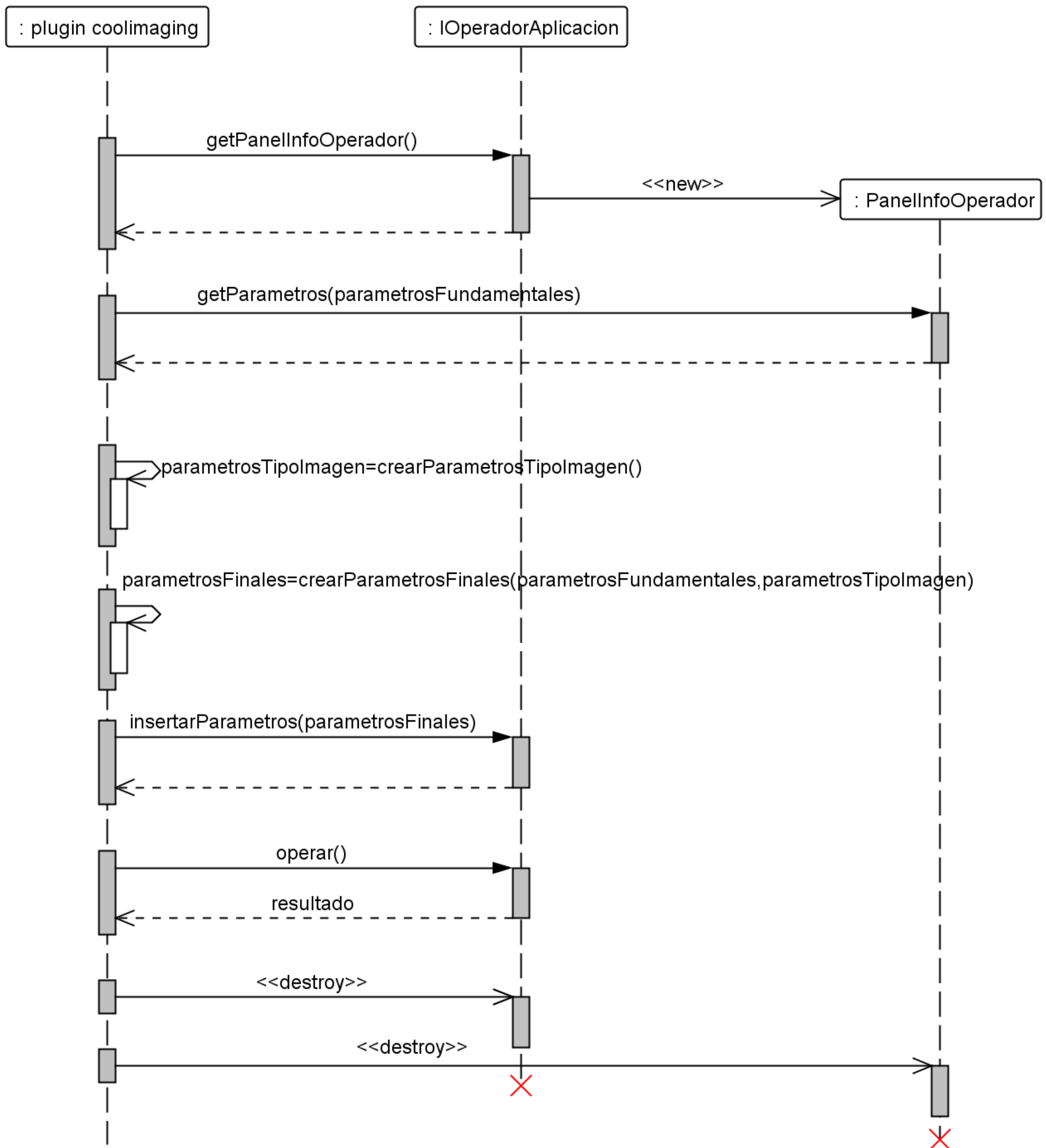

Figura 6.7: Diagrama de secuencia del flujo habitual de una operación en Cool Imaging

operaciones al *plugin coolimaging*, el cual es capaz tanto de reconocerlas como de trabajar con ellas.

En esta sección se describe cómo el *plugin coolimaging* permite al usuario iniciar una operación, es decir, cómo puede indicar al sistema que desea usarla.

Tal y como se describió en el *storyboard*, Cool Imaging ofrece al usuario dos menús de operaciones en forma de árbol. Uno de los menús le permite seleccionar las operaciones de tratamiento de imágenes, mientras que el otro le permite seleccionar las operaciones de caracterización. Cuando el usuario selecciona una operación, ya sea de tratamiento o de caracterización de imágenes, se le muestra un panel gráfico donde se le permite trabajar con la operación seleccionada. Si hace doble click sobre la operación, este panel se le muestra como diálogo emergente.

Veamos cómo se crean los menús de operaciones en forma de árbol.

6.6.1. Menú en árbol de las operaciones

El propósito es construir una estructura de menú en forma de árbol que presente al usuario todas las operaciones presentes en el sistema, de modo que pueda elegir qué operación quiere aplicar de forma rápida y sencilla. El elegir una estructura de árbol no tiene otro propósito que el de estructurar las operaciones según su semántica. Por ejemplo, se podrían agrupar las operaciones en varias categorías, a saber, *operaciones unarias* y *operaciones binarias*. El grupo de operaciones binarias podría contener, a su vez, otras categorías. Dentro de cada categoría se situarían las operaciones concretas. Además, podría haber operaciones sin categoría, situadas en la raíz del árbol. En el capítulo 5 se puede apreciar una representación gráfica de la estructura del árbol.

Así pues, nos disponemos en primer lugar a crear un modelo conceptual que refleje la estructura de árbol deseada. Dicha estructura es independiente de si la operación subyacente es de tratamiento o de caracterización de imágenes, ya que ambas están representadas por interfaz *IOperadorAplicacion*.

El árbol de operaciones consta de dos tipos de nodos básicos: *nodos operación* y *nodos categoría*. Un *nodo categoría* representa una categoría en la que poder incluir tanto otras categorías como diferentes operaciones. Un *nodo operación* representa una operación determinada. Un nodo categoría se caracteriza por poder tener hijos tanto de tipo nodo categoría como nodo operación. Un nodo operación, en cambio, no puede contener hijos. Ambos tipos de nodo pueden contener padres, salvo si se sitúan en el nivel más alto del árbol, en cuyo caso no tienen un padre asignado. Estos tipos de nodo están implementados en tres clases distintas: *NodoArbolOperaciones*, *NodoCategoria* y *NodoOperacion*. La clase *NodoArbolOperaciones* representa la funcionalidad común a los dos tipos de nodos, es decir, la posibilidad de asignar a un nodo tanto un nombre como un padre determinado. El *NodoCategoria*, que hereda de *NodoArbolOperacion*, permite almacenar hijos de tipo *NodoArbolOperacion*. La clase *NodoOperacion*, que hereda de *NodoArbolOperaciones*, almacena una estructura de datos que permite la invocación de una operación determinada. Este último detalle, sin embargo, no nos concierne ahora, así que lo dejaremos a un lado.

La clase *ArbolOperaciones* abstrae un árbol formado por nodos de tipo *NodoArbolOperaciones*, y representa la estructura de datos que se usa para el almacenamiento de los menús de operaciones. El *ArbolOperaciones* puede contener varias raíces, ya que, en el menú en forma de árbol, puede haber varias categorías al máximo nivel, así como operaciones que no estén asignadas a ninguna categoría: todos ellos son situados como raíces del árbol. Así pues, el *ArbolOperaciones* define un conjunto de raíces, que son almacenadas en un *Vector*. El *ArbolOperaciones* define, además, un método de inserción de objetos de tipo *NodoArbolOperaciones* dentro del árbol, el cual es usado para poblar el árbol con las distintas operaciones.

Como se ha explicado anteriormente, existen dos árboles de operaciones: el que contiene las

operaciones de caracterización y el que contiene las operaciones de tratamiento de imágenes. Ambos árboles son mostrados en vistas diferentes del Eclipse RCP (*ViewPart*). La clase *MenuArbolOperacionesTratamientoImagenes* es la encargada de visualizar el menú de operaciones de tratamiento de imágenes, mientras que la clase *MenuArbolOperacionesCaracterizacionImagenes* muestra el menú de operaciones de caracterización de imágenes.

A continuación se recoge una explicación detallada de cada una de las clases anteriormente nombradas. La figura 6.8 representa el diagrama de clases correspondiente.

6.6.1.1. NodoArbolOperaciones

La clase *NodoArbolOperaciones* representa un nodo genérico del árbol de operaciones (*ArbolOperaciones*). Un nodo del *ArbolOperaciones* se caracteriza por tener asociado tanto un nombre como un padre dentro del árbol. Obsérvese que no todo nodo del *ArbolOperaciones* tiene hijos, y de ahí que la clase *NodoArbolOperaciones* no contenga dicha funcionalidad. De esta clase heredan las clases *NodoCategoria* y *NodoOperacion*.

6.6.1.2. NodoCategoria

La clase *NodoCategoria* representa un nodo de tipo *categoría* dentro del *ArbolOperaciones*. Un *NodoCategoria* se caracteriza por poder tener hijos. Un *NodoCategoria* puede contener hijos de tipo *NodoArbolOperaciones*

6.6.1.3. NodoOperacion

La clase *NodoOperacion* representa un nodo de tipo *operacion* dentro del *ArbolOperaciones*. Un *NodoOperacion* se caracteriza porque tiene asociada una estructura de datos que permite, a partir de él, ejecutar una operación. Puesto que por ahora sólo estamos analizando la construcción del árbol, y no la aplicación de operaciones, este aspecto lo dejaremos a un lado. En la sección 6.7 explicaremos con más detalle esta estructura de datos y cómo, a partir de ella, se puede ejecutar una operación determinada.

6.6.1.4. ArbolOperaciones

La clase *ArbolOperaciones* representa un árbol con nodos de tipo *NodoArbolOperaciones*. El *ArbolOperaciones* puede contener varias raíces, y proporciona la funcionalidad necesaria para poder insertar nodos dentro de él. Concretamente, la función *ArbolOperaciones.insertarNodo()* permite introducir objetos de tipo *NodoArbolOperaciones*, especificando para ello la categoría en la que se quiere insertar a dicho nodo (o no especificando ninguna categoría, en caso de que no se quiera asociar dicho nodo a ninguna categoría).

6.6.1.5. MenuArbolOperacionesTratamientoImagenes y MenuArbolOperacionesCaracterizacionImagenes

Las clases *MenuArbolOperacionesTratamientoImagenes* y *MenuArbolOperacionesCaracterizacionImagenes* son las encargadas de visualizar, respectivamente, los menús de operaciones de tratamiento y de caracterización. Ambas clases inicializan los respectivos objetos *ArbolOperaciones*, uno con las operaciones de tratamiento y otro con las operaciones de caracterización.

Estas dos clases son capaces de visualizar gráficamente un objeto *ArbolOperaciones*. Ambas están implementadas como una vista de Eclipse RCP, es decir, ambas extienden la clase *ViewPart*.

Para visualizar el árbol hacen uso de la clase *TreeViewer*, la cual está especializada en la representación de estructuras de tipo árbol. Para la representación de una estructura de

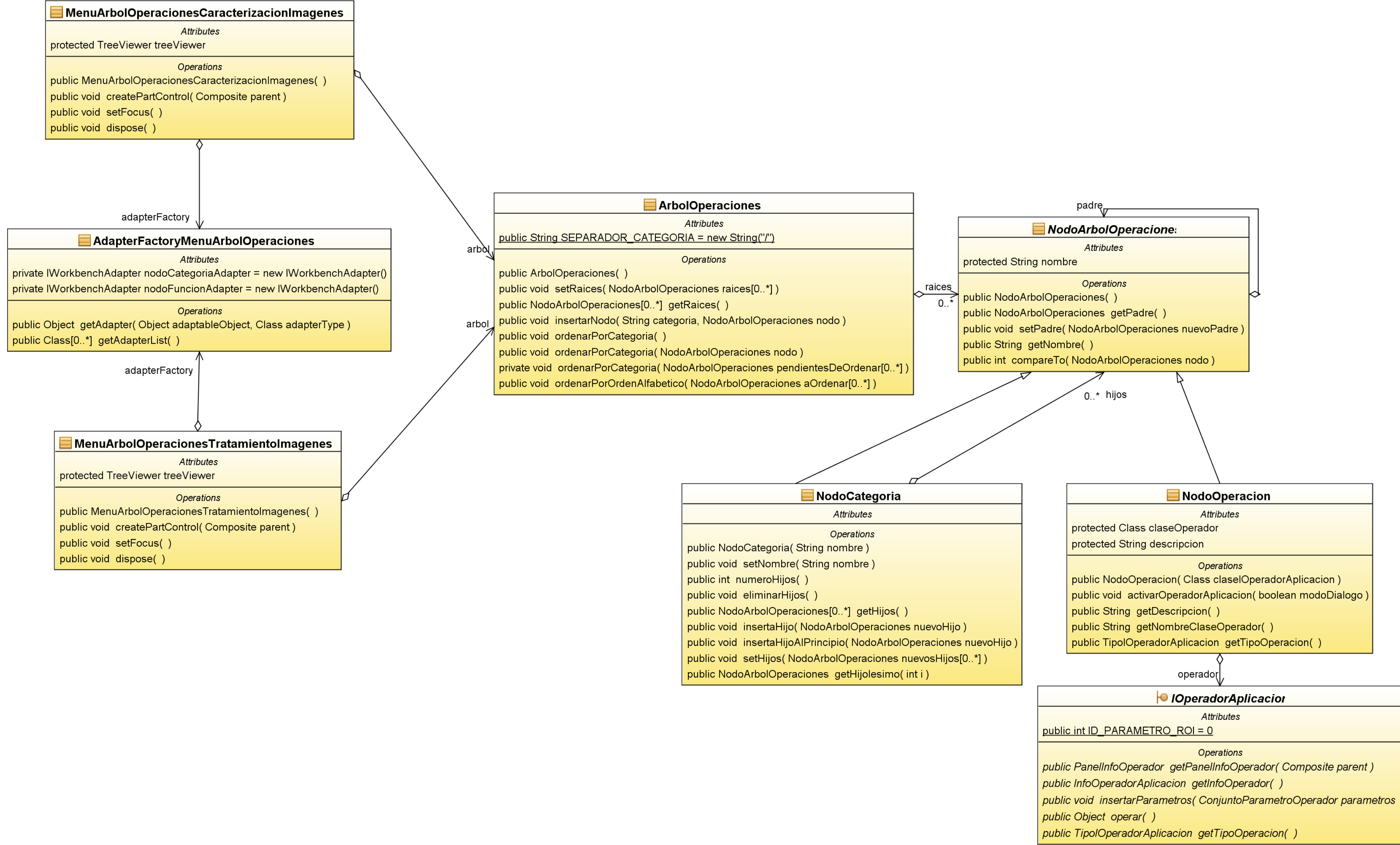

Figura 6.8: Diagrama de clases del menú en árbol de las operaciones

tipo árbol, el *TreeViewer* necesita un *ContentProvider* y un *LabelProvider*. El *ContentProvider* permite transformar cualquier modelo de datos (en nuestro caso el *ArbolOperaciones*) en una estructura jerárquica de tipo árbol que es comprensible por el *TreeViewer*. El *LabelProvider*, por otro lado, permite determinar qué etiquetas textuales e imágenes son visualizadas para cada elemento del árbol que le proporciona el *ContentProvider*. Por comodidad, se ha hecho uso de las clases *BaseWorkbenchContentProvider* y *WorckbenchLabelProvider*, así como de un *IAdapterFactory* (la clase *AdapterFactoryMenuArbolOperaciones*), el cual proporciona los *IWorkbenchAdapter* necesarios para la correcta visualización del *TreeViewer* del *MenuArbolOperacionesTratamientoImagenes* y del *MenuArbolOperacionesCaracterizacionImagenes*[8].

El último detalle a tener en cuenta es el de la creación del *ArbolOperaciones* que contiene las operaciones de tratamiento de imágenes, así como la del *ArbolOperaciones* que contiene las operaciones de caracterización. Las clases *MenuArbolOperacionesTratamientoImagenes* y *MenuArbolOperacionesCaracterizacionImagenes* necesitan dichos árboles, para poder visualizarlos. Dichos árboles, sin embargo, no son creados por estas clases. En cambio, son creados por la clase *Configurador*[9], y es ésta la clase encargada de entregar a cada menú la referencia del árbol que necesita.

6.7. Control de operaciones: inicialización

Las clases descritas en la sección 6.6.1 permiten representar gráficamente una estructura en forma de árbol (incrustada en una vista del RCP) donde se pueden seleccionar las operaciones a usar en la aplicación. La cuestión ahora es: ¿cómo se ejecutan, en efecto, dichas operaciones?

Cuando el usuario selecciona una operación de alguno de los menús (ya sea el de operaciones de tratamiento o de caracterización de imágenes), la aplicación debe presentar al usuario un panel con la información asociada a dicha operación, es decir, un panel donde el usuario debe introducir los datos necesarios para poder hacer uso de la operación que ha seleccionado. Si el usuario hace doble click en una determinada operación, el panel de la operación es mostrado mediante un diálogo emergente, siendo el resto del procedimiento de ejecución de la operación el mismo.

Las clases que desencadenan este proceso son la clase *ArbolOperaciones*, la clase *GestorOperadores*, la clase *ControladorOperadorTratamientoImagenes* y la clase *ControladorOperadorCaracterizacionImagenes*.

Cuando el usuario selecciona una operación en el menú en forma de árbol (clases *MenuArbolOperacionesTratamiento* y *MenuArbolOperacionesCaracterizacion*), se llama al método *public void activarOperadorAplicacion(boolean modoDialogo)* en el *NodoOperacion* seleccionado. En las secciones anteriores comentábamos que, de algún modo, el *NodoOperacion* era el encargado de comenzar la ejecución de una operación. Es a través de este método que se inicia la ejecución de la operación.

Internamente, el método *activarOperadorAplicacio()* llama al método *public void activarOperador(Class claseOperador, boolean modoDialogo)* de la clase *GestorOperadores*. El *Class* que recibe es justamente la clase del *IOperadorAplicacion* almacenada por el *NodoOperador*. Cada *NodoOperador* almacena la clase del *IOperadorAplicacion* asociado a la operación representada por el nodo. Cuando se llama al método *activarOperador()* de la clase *GestorOperadores*, se pretende inicializar justamente dicha operación.

En el método *activarOperador()*, el *GestorOperadores* discierne entre si la operación recibida es de tratamiento de imágenes o de caracterización. A través del objeto *Class* que recibe, crea

[8]Para más información acerca del uso de la clase *TreeViewer* se puede visitar la API de Eclipse en http://help.eclipse.org/stable/nftopic/org.eclipse.platform.doc.isv/reference/api/allclasses-noframe.html.

[9]En la sección 6.10 se da una descripción detallada de la clase *Configurador*.

una instancia de la operación (*IOperadorAplicacion*). A dicha instancia le consulta si se trata de una operación de caracterización o de tratamiento de imágenes haciendo uso del método *getTipoOperacion()* de la interfaz.

Hecha esa distinción, el *GestorOperadores* delega la gestión de la operación a un controlador creado para tal efecto. Si la operación es de tratamiento de imágenes, el *GestorOperadores* crea un *ControladorOperadorTratamientoImagenes*, mientras que si es de caracterización, crea un *ControladorOperadorCaracterizacionImagenes*. El controlador creado será el encargado de hacer visualizar el panel de la operación, así como gestionar cómo el usuario puede hacer uso de ella.

La figura 6.9 representa el diagrama de secuencia de la inicialización de una operación de tratamiento de imágenes.

6.8. Control de operaciones: operaciones de tratamiento de imágenes

Cuando el usuario selecciona una operación en el menú de las operaciones de tratamiento de imágenes, se desencadena una secuencia que desemboca en la creación de un objeto *ControladorOperacionTratamientoImagenes*[10], el cual se encarga de gestionar la visualización del panel que permite al usuario hacer uso de la operación, así como de gestionar las opciones que el panel le ofrece al usuario.

Todo *ControladorOperacionTratamientoImagenes* tiene asociada una operación de tratamiento de imágenes (*IOperadorAplicacion*). Cuando se crea un *ControladorOperacionTratamientoImagenes*, éste se encarga de visualizar al usuario un panel que le permita hacer uso de la operación. Dicho panel puede mostrarse como diálogo emergente o incrustado en una vista (*ViewPart*) de la aplicación, la *VistaOperadorTratamientoImagenes*. En el constructor de la clase *ControladorOperacionTratamientoImagenes* se especifica tanto el *IOperadorAplicacion* como si se quiere visualizar el panel en modo diálogo emergente o no.

El panel que se le muestra al usuario, ya sea en el diálogo emergente o en la *VistaOperadorTratamientoImagenes*, está representado por la clase *PanelOperadorTratamientoImagenes*. Esta clase es un panel gráfico que le permite al usuario, bien aplicar la operación de forma directa, o bien insertarla en una **cadena de operaciones**. El panel gráfico se corresponde con el panel de la figura 5.3 del *storyboard*, y, principalmente, permite al usuario introducir los parámetros de entrada de la operación.

Tanto si el usuario aplica la operación de forma directa como si la operación es insertada en una cadena de operaciones, la gestión de tal evento es controlada por el *ControladorOperacionTratamiento*.

A continuación se describe de forma detallada cada una de las clases mencionadas. La figura 6.10 representa el diagrama de clases de las clases implicadas.

6.8.1. ControladorOperadorTratamientoImagenes

Esta clase es la encargada de gestionar el flujo de eventos relacionados con la visualización y el uso de las operaciones de tratamiento de imágenes.

Cuando el *GestorOperadores* crea un *ControladorOperadorTratamientoImagenes*, éste se encarga, en primer lugar, de visualizar el panel a través del cual el usuario hará uso de la operación. Dicho panel está representado por la clase *PanelOperadorTratamientoImagenes*. Si se quiere hacer uso de la operación en modo diálogo emergente, el *ControladorOperadorTratamientoImagenes* crea un objeto de tipo *DialogoPanelOperador*, que es un simple diálogo de JFace cuyo contenido es el *PanelOperadorTratamientoImagenes*. Por contra, si no se quiere mostrar

[10] Ver sección 6.7.

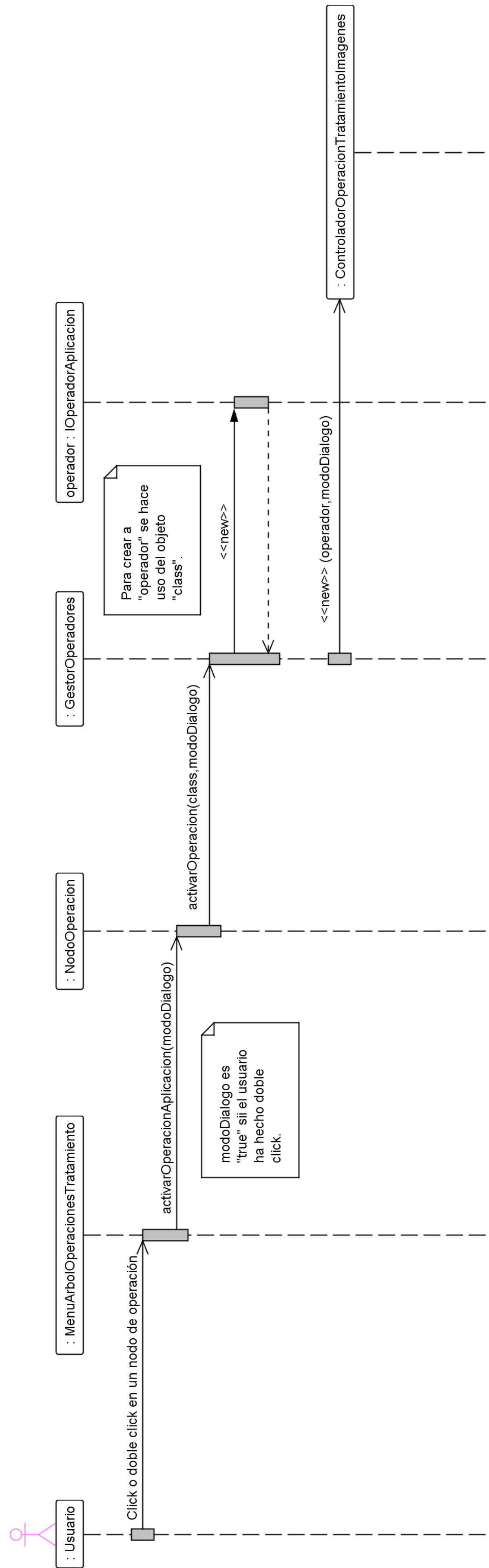

Figura 6.9: Diagrama de secuencia de la inicialización de una operación de tratamiento de imágenes

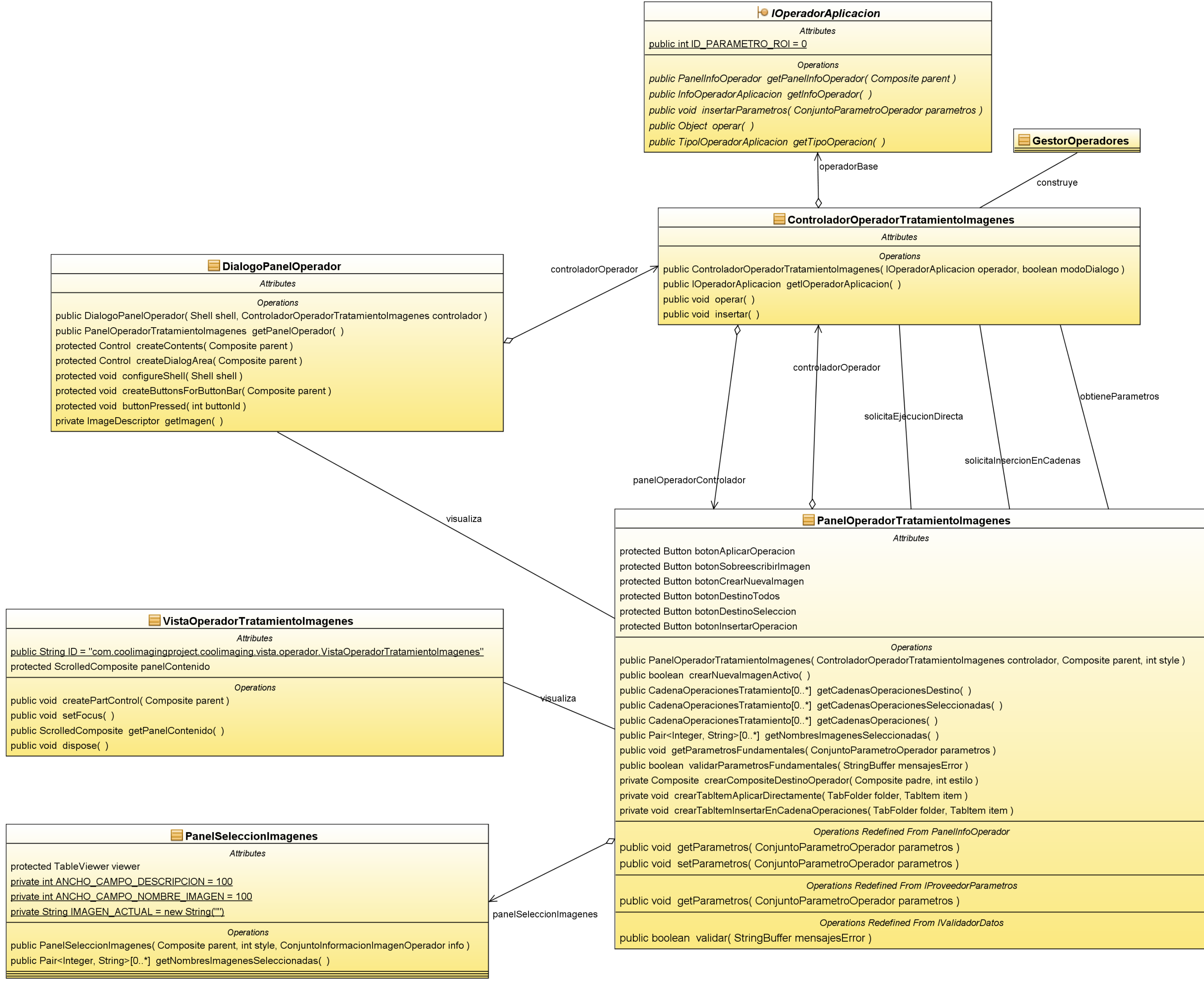

Figura 6.10: Diagrama de clases de la gestión de las operaciones de tratamiento de imágenes

en modo diálogo emergente, el *PanelOperadorTratamientoImagenes* simplemente se incrusta en la *VistaOperadorTratamientoImagenes.*

Creado y visualizado el panel, éste le ofrece al usuario dos posibles opciones. Una es la de hacer uso directo de la operación, es decir, aplicarla a cualesquiera imágenes haya abiertas, para obtener así la imagen de salida. La otra opción es la de insertar la operación en una o varias cadenas de operaciones.

- El uso directo de la operación es controlado por el método *operar()* de la clase *ControladorOperadorTratamientoImagenes.* Cuando el *PanelOperadorTratamientoImagenes* le solicita al *ControladorOperadorTratamientoImagenes* aplicar la operación de forma directa, éste, en primer lugar, extrae los parámetros necesarios para realizar la operación. El *PanelOperadorTratamientoImagenes* es capaz de proporcionar, al *ControladorOperadorTratamientoImagenes*, **todos los parámetros** que necesita para la ejecución de la operación. Para ello, el *ControladorOperadorTratamientoImagenes* llama al método *getParametros()* de la clase *PanelOperadorTratamientoImagenes*, el cual devuelve todos los parámetros que necesita la operación (*IOperadorAplicacion*) para ser ejecutada.

  Tras extraer los parámetros de la operación, el método *operar()* crea una copia del *IOperadorAplicacion*, la cual será la usada para realizar los cálculos de la operación. En breve se explicará por qué se hace una copia, en vez de hacer uso del *IOperadorAplicacion* que almacena el controlador. A esta copia del *IOperadorAplicacion*, el controlador le registra los parámetros necesarios para su funcionamiento. Para ello llama al método *insertarParametros()* de la interfaz *IOperadorAplicacion.* Tras registrar los parámetros, se llama al método *operar()* del *IOperadorAplicacion*, y se obtiene la imagen resultado, que se muestra en la aplicación. La imagen resultado puede mostrarse como una nueva imagen (que es insertada a través del *ControladorImagenes*), o bien puede sobreescribir a la actualmente activa.

  Se crea una copia del *IOperadorAplicacion* porque la mayor parte del código de esta función se planifica para ser ejecutado en una hebra independiente. Justo cuando el controlador acaba de extraer los parámetros del panel, se crea una hebra encargada de realizar el resto de tareas. Con ello se permite que la ejecución de la operación, que es una tarea computacionalmente cara, no bloquee la interfaz de usuario. Además, de este modo, el usuario puede ejecutar una misma operación varias veces de forma simultanea, ya que cada ejecución se planifica en una hebra independiente. Si se usara la misma referencia del *IOperadorAplicacion* que almacena el controlador, las distintas hebras accederían de forma concurrente al *IOperadorAplicacion*, de modo que el resultado de las operaciones podría ser desastroso.

- Cuando se desea insertar una operación de tratamiento de imágenes en una o varias cadenas de operaciones, el *PanelOperadorTratamientoImagenes* llama al método *insertar()* del *ControladorOperadorTratamientoImagenes.* En el *PanelOperadorTratamientoImagenes*, el usuario selecciona las cadenas de operaciones donde desea insertar la operación. El *ControladorOperadorTratamientoImagenes* le pregunta al *PanelOperadorTratamientoImagenes* por dichas cadenas, e inserta la operación en las cadenas seleccionadas. Una cadena de operaciones (*CadenaOperacionesTratamiento*) está compuesta de elementos de tipo *ElementoCadenaOperacionesTratamiento.* Cada uno de estos elementos almacena la operación, así como los **parámetros fundamentales** necesarios para la ejecución de la operación.

  El controlador, en realidad, crea un objeto de tipo *ElementoCadenaOperacionesTratamiento* por cada una de las cadenas de destino. A cada uno de estos elementos le asocia una copia del *IOperadorAplicacion* así como una copia de los parámetros fundamentales.

Se hace así para que las distintas cadenas de operaciones no compartan ni el *IOperaodorAplicacion* subyacente ni los mismos parámetros. De no hacerse así podría haber problemas de consistencia al compartir cada cadena el mismo *IOperadorAplicacion* y los mismos parámetros.

6.8.2. PanelOperadorTratamientoImagenes

La clase *PanelOperadorTratamientoImagenes* representa el panel visual que al usuario se le muestra cuando quiere hacer uso de una operación de tratamiento de imágenes. Como se ha explicado, este panel puede visualizarse bien a través de un diálogo emergente o bien incrustado en la vista *VistaOperadorTratamientoImagenes*.

Este panel tiene dos áreas bien diferenciadas. El área superior presenta al usuario la región donde puede introducir los parámetros fundamentales de la operación. Dicha región coincide exactamente con el *PanelInfoOperador* que le devuelve el *IOperadorAplicacion* a través de su método *getPanelInfoOperador()*. Así pues, cuando se crea un *PanelOperadorTratamientoImagenes*, éste necesita recibir un objeto de tipo *IOperadorAplicacion*, del cual extraer el panel donde introducir los parámetros fundamentales. En lugar de ello, recibe el *ControladorOperadorTratamientoImagenes*, a través del cual, no sólo puede obtener el *IOperadorAplicacion*, sino desencadenar las llamadas a las funciones *operar()* e *insertar()* explicadas en la sección 6.8.1.

El área inferior le da al usuario la doble elección: bien aplicar la operación de forma directa a una o varias imágenes abiertas, o bien insertarla en una o varias cadenas de operaciones.

La región que le permite aplicar la operación de forma directa consta de una tabla donde puede seleccionar la imagen o imágenes a las que aplicar la operación. Esta tabla, representada por la clase *PanelSeleccionImagenes*, muestra un listado donde el usuario selecciona las imágenes requeridas para aplicar la operación. Por ejemplo, si la operación es de rotación, el usuario podría seleccionar en dicha tabla una imagen, la imagen a rotar. Si la operación fuera una *and* lógica, seleccionaría en dicha tabla las dos imágenes que requiere la *and*. Dicha tabla es construida gracias a la información adicional que se puede extraer del *IOperadorAplicacion*. Recuérdese que el *IOperadorAplicacion* dispone de un método que permite acceder a la información que especifica cuántas imágenes requiere la operación para poder ejecutarse. Dicha información es la usada para construir el *PanelSeleccionImagenes*.

Además de mostrar una tabla donde seleccionar la imagen o imágenes, esta región permite especificar una opción: si la imagen obtenida como resultado debe sobreescribir a la actualmente activa en la aplicación o bien se debe construir una nueva.

Cuando el usuario presiona el botón que indica aplicar la operación, se llama al método *operar()* del controlador almacenado por el panel, el cual se encarga de realizar la operación. Cuando el controlador le pide al panel **todos** los parámetros de la operación, a través del método *getParametros()*, el panel hace uso, en primer lugar, del *PanelInfoOperador* devuelto por el método *getPanelInfoOperador()* de la interfaz *IOperadorAplicacion*. Dicho panel proporciona los parámetros fundamentales para la ejecución de la operación. Los parámetros de tipo imagen son extraídos por el *PanelSeleccionImagenes*.

La región que permite insertar la operación en una o varias cadenas de operaciones muestra un listado con todas las cadenas de operaciones definidas en la aplicación. Cuando se selecciona una cadena, otro panel situado a su derecha muestra las operaciones que hay actualmente en la cadena seleccionada. El panel que permite seleccionar las cadenas y mostrar las operaciones que hay en la cadena seleccionada está representado por la clase *VistaCoordinadaCadenaOperacionesTratamiento*.

Además, en esta región se puede especificar si se quiere insertar la operación en todas las cadenas de operaciones o sólo en las seleccionadas.

Cuando el usuario presiona el botón que indica insertar la operación en las cadenas selec-

cionadas, el panel llama al método *insertar()* del controlador almacenado, desencadenando la inserción de la operación en las cadenas seleccionadas.

El diagrama de colaboración de la figura 6.11 muestra el flujo de operaciones entre objetos que se produce desde que el usuario pulsa el botón de ejecutar operación (uso directo de la operación), hasta que el sistema finalmente produce la nueva imagen.

El diagrama de secuencia de la figura 6.12 muestra el flujo de operaciones entre objetos que se produce cuando el usuario pulsa el botón de insertar la operación en las cadenas de operaciones seleccionadas.

6.9. Control de operaciones: operaciones de caracterización de imágenes

El control de las operaciones de caracterización de imágenes sigue una filosofía similar a la de las operaciones de tratamiento.

Cuando el usuario selecciona una operación del menú de operaciones de caracterización, se desencadena una secuencia de eventos que desemboca en la creación, por parte del *GestorOperadores*, de un *ControladorOperacionCaracterizacionImagenes*.

Al igual que ocurre con el *ControladorOperacionTratamientoImagenes*, este controlador almacena el *IOperadorAplicacion* de la operación que gestiona. También se encarga de la visualización del panel que permite al usuario aplicar la operación. Igualmente, el panel puede visualizarse como diálogo emergente o incrustado en una vista (*ViewPart*), la *VistaOperadorCaracterizacionImagenes*. El panel que permite hacer uso de la operación está representado por la clase *PanelOperadorCaracterizacionImagenes*, y su aspecto coincide con el de la figura 5.15 del *storyboard*. Este panel permite al usuario, bien aplicar la operación de caracterización de forma directa sobre una imagen, o bien insertarla en un **generador de vector de caracterización**.

En cualquier caso, es la clase *ControladorOperacionCaracterizacionImagenes* la encargada de gestionar los eventos relacionados con la ejecución de las operaciones de caracterización.

A continuación se describe de forma detallada cada una de las clases mencionadas. La figura 6.13 representa el diagrama de clases de las clases implicadas.

6.9.1. ControladorOperadorCaracterizacionImagenes

Esta clase es la encargada de gestionar el flujo de eventos relacionados con la visualización y el uso de las operaciones de caracterización de imágenes.

Cuando el *GestorOperadores* crea un *ControladorOperadorCaracterizacionImagenes*, éste se encarga, en primer lugar, de visualizar el panel a través del cual el usuario hará uso de la operación. Dicho panel está representado por la clase *PanelOperadorCaracterizacionImagenes*. Si se quiere hacer uso de la operación en modo diálogo emergente, el *ControladorOperadorCaracterizacionImagenes* crea un objeto de tipo *DialogoPanelOperador*, que es un simple diálogo de JFace cuyo contenido es el *PanelOperadorCaracterizacionImagenes*. Por contra, si no se quiere mostrar en modo diálogo emergente, el *PanelOperadorCaracterizacionImagenes* simplemente se incrusta en la *VistaOperadorCaracterizacionImagenes*.

Creado y visualizado el panel, éste le ofrece al usuario dos posibles opciones. Una es la de hacer uso directo de la operación de caracterización, es decir, aplicarla a cualquier imagen que haya abierta, y obtener así un informe de caracterización (*InformeCaracterizacion*) de salida. La otra opción es la de insertar la operación en uno o varios generadores de vector de caracterización.

- El uso directo de la operación es controlado por el método *operar()* de la clase *ControladorOperadorCaracterizacionImagenes*. Cuando el *PanelOperadorCaracterizacionImage-*

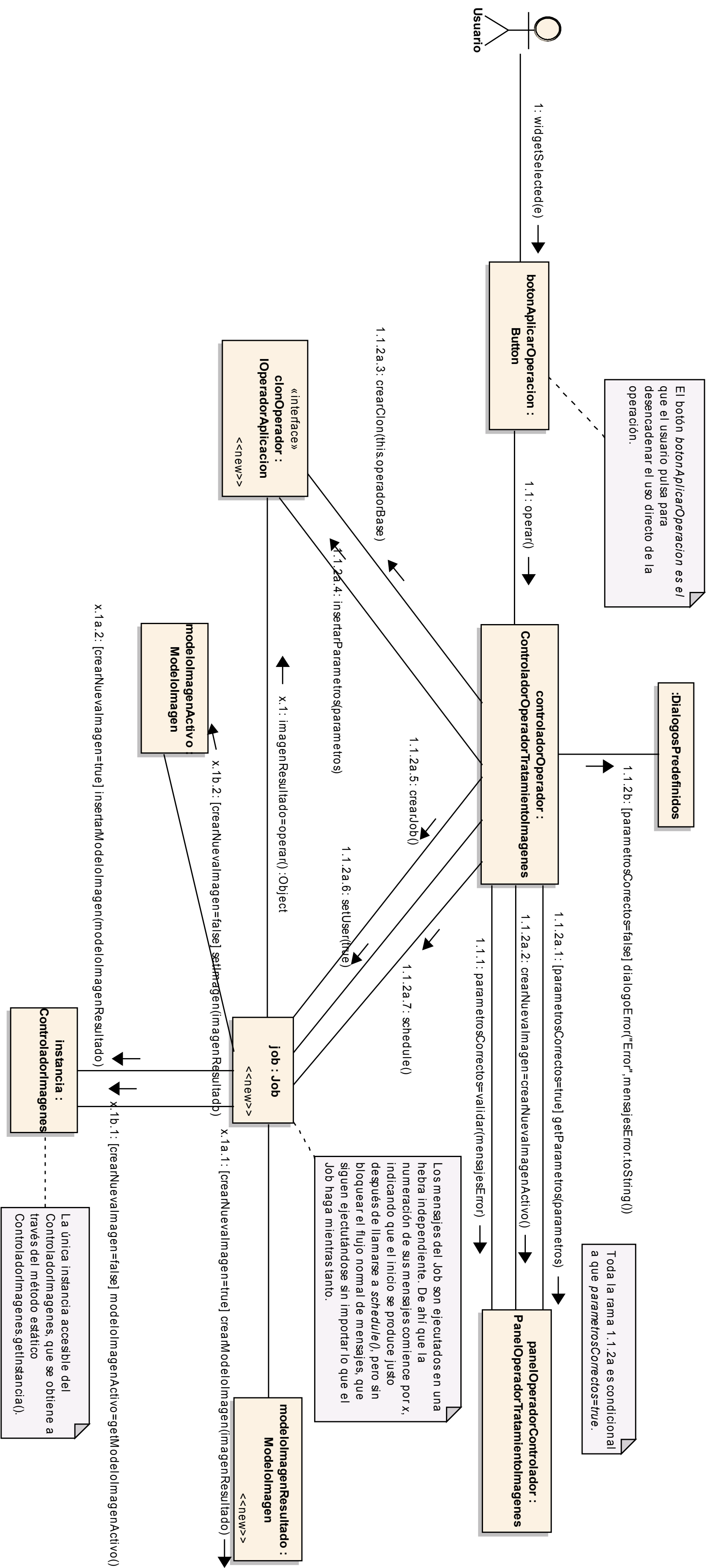

Figura 6.11: Diagrama de colaboración del uso directo de una operación de tratamiento de imágenes

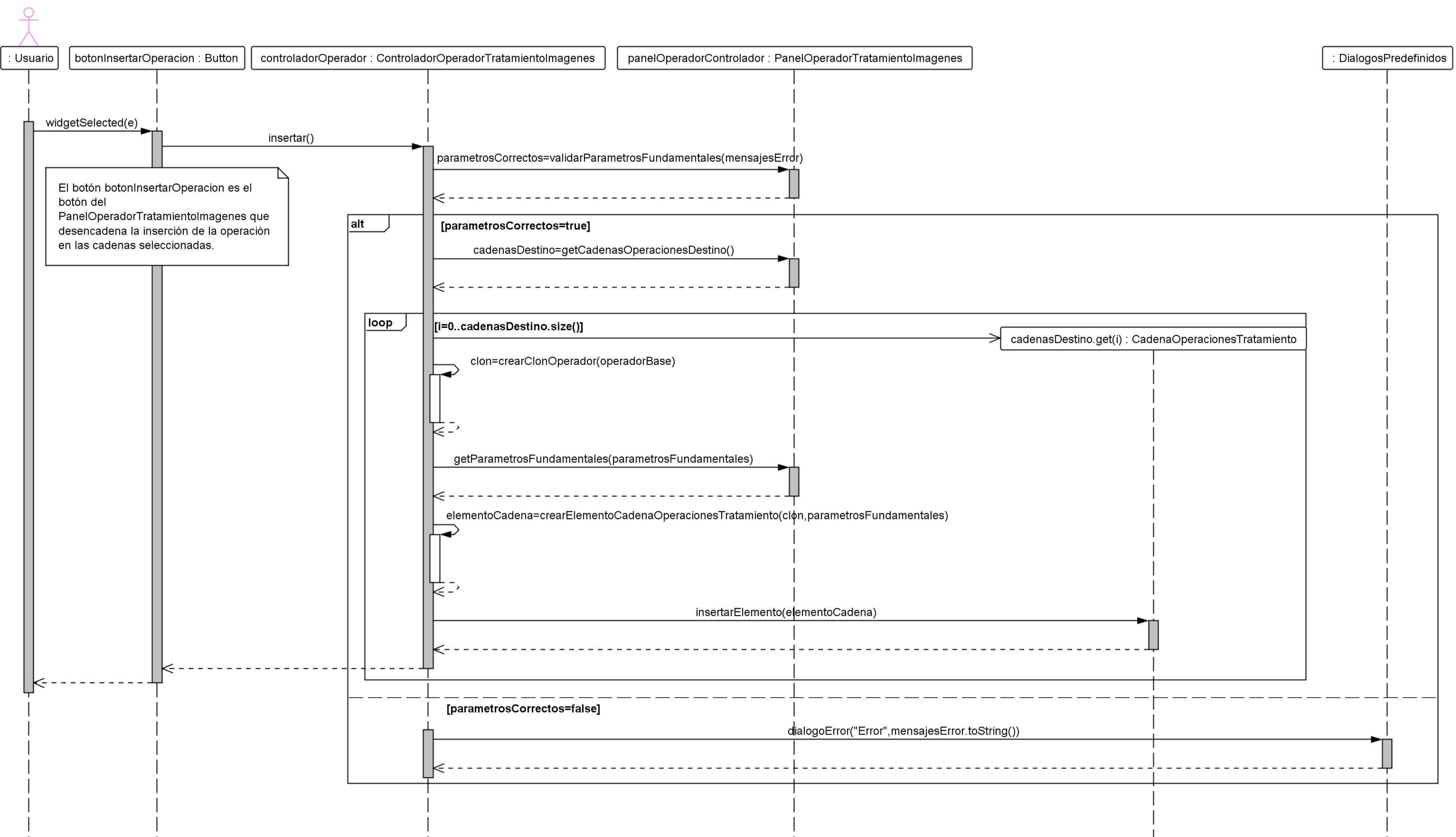

Figura 6.12: Diagrama de secuencia de la inserción de una operación de tratamiento de imágenes en varias cadenas de operaciones

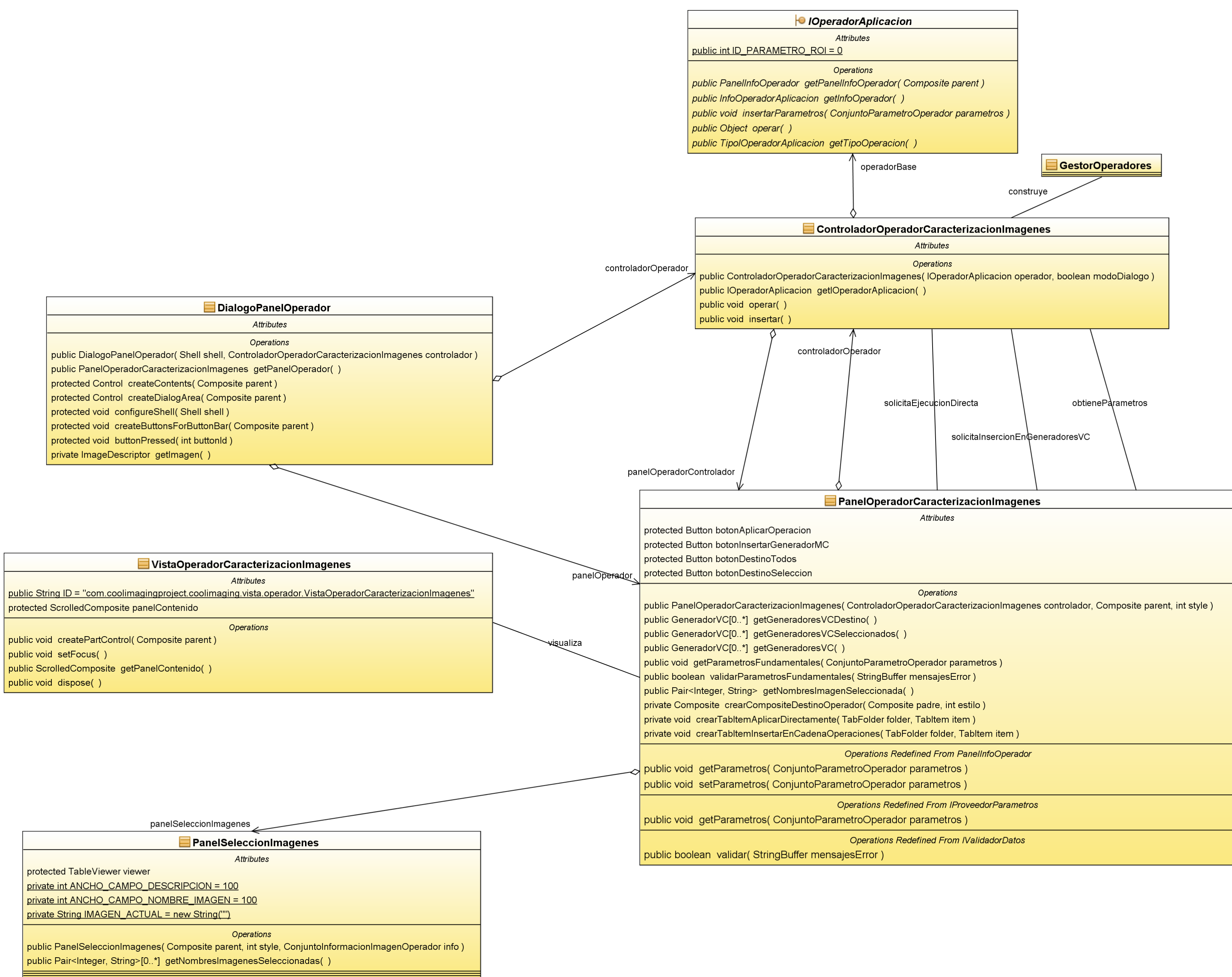

Figura 6.13: Diagrama de clases de la gestión de las operaciones de caracterización de imágenes

nes le solicita al *ControladorOperadorCaracterizacionImagenes* aplicar la operación de forma directa, éste, en primer lugar, extrae los parámetros necesarios para realizar la operación. El *PanelOperadorCaracterizacionImagenes* es capaz de proporcionar, al *ControladorOperadorCaracterizacionImagenes*, **todos los parámetros** que necesita para la ejecución de la operación de caracterización. Para ello, el *ControladorOperadorCaracterizacionImagenes* llama al método *getParametros()* de la clase *PanelOperadorCaracterizacionImagenes*, el cual devuelve todos los parámetros que necesita la operación (*IOperadorAplicacion*) para ser ejecutada.

Tras extraer los parámetros de la operación, el método *operar()* crea una copia del *IOperadorAplicacion*, la cual será la usada para realizar los cálculos de la operación. En breve se explicará por qué se hace una copia, en vez de hacer uso del *IOperadorAplicacion* que almacena el controlador. A esta copia del *IOperadorAplicacion*, el controlador le registra los parámetros necesarios para su funcionamiento. Para ello llama al método *insertarParametros()* de la interfaz *IOperadorAplicacion*. Tras registrar los parámetros, se llama al método *operar()* del *IOperadorAplicacion*, y se obtiene el informe de caracterización resultado, que se muestra en la aplicación.

Se crea una copia del *IOperadorAplicacion* porque la mayor parte del código de esta función se planifica para ser ejecutado en una hebra independiente. Justo cuando el controlador acaba de extraer los parámetros del panel, se crea una hebra encargada de realizar el resto de tareas. Con ello se permite que la ejecución de la operación, que es una tarea computacionalmente cara, no bloquee la interfaz de usuario. Además, de este modo, el usuario puede ejecutar una misma operación varias veces de forma simultanea, ya que cada ejecución se planifica en una hebra independiente. Si se usara la misma referencia del *IOperadorAplicacion* que almacena el controlador, las distintas hebras accederían de forma concurrente al *IOperadorAplicacion*, de modo que el resultado de las operaciones podría ser desastroso.

- Cuando se desea insertar una operación de caracterización de imágenes en uno o varios generadores de vector de caracterización, el *PanelOperadorCaracterizacionImagenes* llama al método *insertar()* del *ControladorOperadorCaracterizacionImagenes*. En el *PanelOperadorCaracterizacionImagenes*, el usuario selecciona los generadores de vector de caracterización donde desea insertar la operación. El *ControladorOperadorCaracterizacionImagenes* le pregunta al *PanelOperadorCaracterizacionImagenes* por dichos generadores, e inserta la operación en los generadores seleccionados. Un generador de vector de caracterización (*GeneradorVC*) está compuesto de elementos de tipo *GeneradorMC*. Cada uno de estos elementos almacena la operación, así como los **parámetros fundamentales** necesarios para la ejecución de la operación. Recuérdese que, en última instancia, un *IOperadorAplicacion* de caracterización devuelve una medida de caracterización (*MC*).

  El controlador, en realidad, crea un objeto de tipo *GeneradorMC* por cada uno de los generadores de destino. A cada uno de estos elementos le asocia una copia del *IOperadorAplicacion* así como una copia de los parámetros fundamentales. Se hace así para que los distintos generadores no compartan ni el *IOperaodorAplicacion* subyacente ni los mismos parámetros. De no hacerse así podría haber problemas de consistencia al compartir cada generador el mismo *IOperadorAplicacion* y los mismos parámetros.

6.9.2. PanelOperadorCaracterizacionImagenes

La clase *PanelOperadorCaracterizacionImagenes* representa el panel visual que al usuario se le muestra cuando quiere hacer uso de una operación de caracterización de imágenes. Como

se ha explicado, este panel puede visualizarse bien a través de un diálogo emergente o bien incrustado en la vista *VistaOperadorCaracterizacionImagenes*.

Este panel tiene dos áreas bien diferenciadas. El área superior presenta al usuario la región donde puede introducir los parámetros fundamentales de la operación de caracterización. Dicha región coincide exactamente con el *PanelInfoOperador* que le devuelve el *IOperadorAplicacion* a través de su método *getPanelInfoOperador()*. Así pues, cuando se crea un *PanelOperador-CaracterizacionImagenes*, éste necesita recibir un objeto de tipo *IOperadorAplicacion*, del cual extraer el panel donde introducir los parámetros fundamentales. En lugar de ello, recibe el *ControladorOperadorCaracterizacionImagenes*, a través del cual, no sólo puede obtener el *IOperadorAplicacion*, sino desencadenar las llamadas a las funciones *operar()* e *insertar()* explicadas en la sección 6.9.1.

El área inferior le da al usuario la doble elección: bien aplicar la operación de forma directa a una imagen de las abiertas, o bien insertarla en uno o varios generadores de vector de caracterización.

La región que le permite aplicar la operación de forma directa consta de una tabla donde puede seleccionar la imagen a la que aplicar la operación de caracterización. Esta tabla, representada por la clase *PanelSeleccionImagenes*, muestra un listado donde el usuario selecciona las imágenes requeridas para aplicar la operación. Al tratarse de una operación de caracterización, la operación sólo requiere una única imagen para operar, y por tanto la tabla muestra una única entrada. Dicha tabla es construida gracias a la información adicional que se puede extraer del *IOperadorAplicacion*. Recuérdese que el *IOperadorAplicacion* dispone de un método que permite acceder a la información que especifica cuántas imágenes requiere la operación para poder ejecutarse. Dicha información es la usada para construir el *PanelSeleccionImagenes*.

Cuando el usuario presiona el botón que indica aplicar la operación, se llama al método *operar()* del controlador almacenado por el panel, el cual se encarga de realizar la operación. Cuando el controlador le pide al panel **todos** los parámetros de la operación, a través del método *getParametros()*, el panel hace uso, en primer lugar, del *PanelInfoOperador* devuelto por el método *getPanelInfoOperador()* de la interfaz *IOperadorAplicacion*. Dicho panel proporciona los parámetros fundamentales para la ejecución de la operación. Los parámetros de tipo imagen son extraídos por el *PanelSeleccionImagenes*.

La región que permite insertar la operación en uno o varios generadores de vector de caracterización muestra un listado con todos los generadores definidos en la aplicación. Cuando se selecciona un generador, otro panel situado a su derecha muestra las operaciones de caracterización que hay actualmente en el generador seleccionado. El panel que permite seleccionar los generadores y mostrar las operaciones que hay en el generador seleccionado está representado por la clase *VistaCoordinadaGeneradorVC*.

Además, en esta región se puede especificar si se quiere insertar la operación en todos los generadores o sólo en los seleccionados.

Cuando el usuario presiona el botón que indica insertar la operación en los generadores seleccionados, el panel llama al método *insertar()* del controlador almacenado, desencadenando la inserción de la operación en los generadores seleccionados.

Como puede verse, la gestión de las operaciones de caracterización es análoga a la de las operaciones de tratamiento de imágenes.

El diagrama de colaboración de la figura 6.14 muestra el flujo de operaciones entre objetos que se produce desde que el usuario pulsa el botón de ejecutar operación (uso directo de la operación), hasta que el sistema finalmente produce el informe de caracterización de la caracterización llevada a cabo.

El diagrama de secuencia de la figura 6.15 muestra el flujo de operaciones entre objetos que se produce cuando el usuario pulsa el botón de insertar la operación en los generadores de vector de caracterización seleccionados.

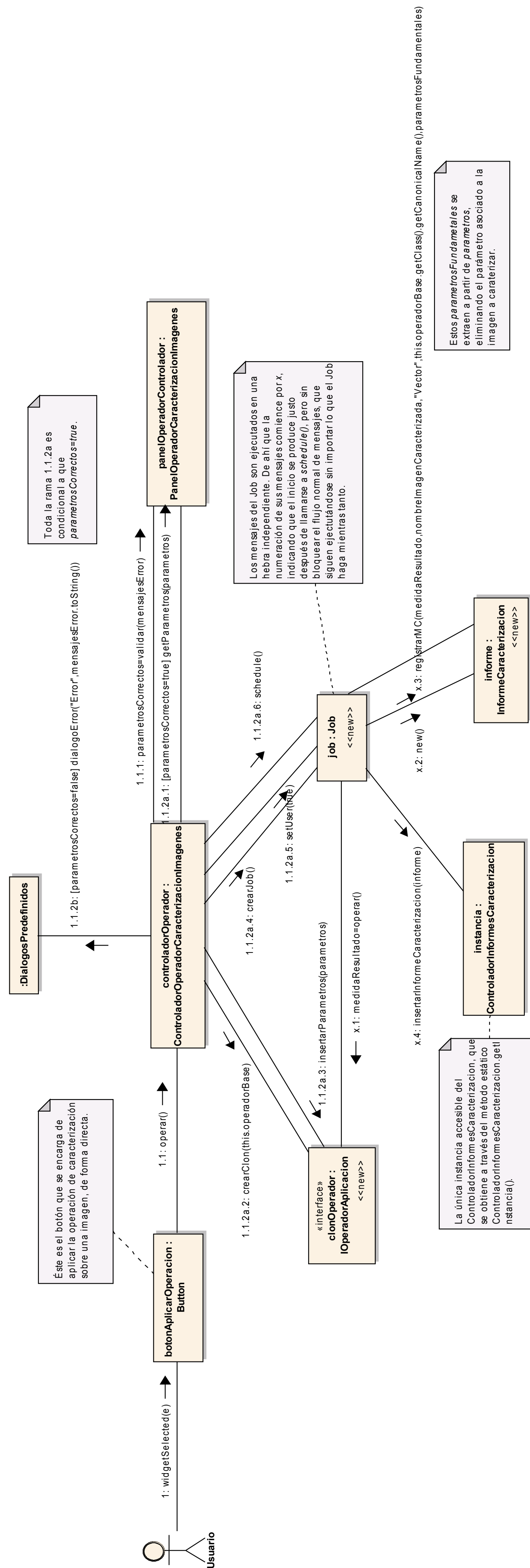

Figura 6.14: Diagrama de colaboración del uso directo de una operación de caracterización de imágenes

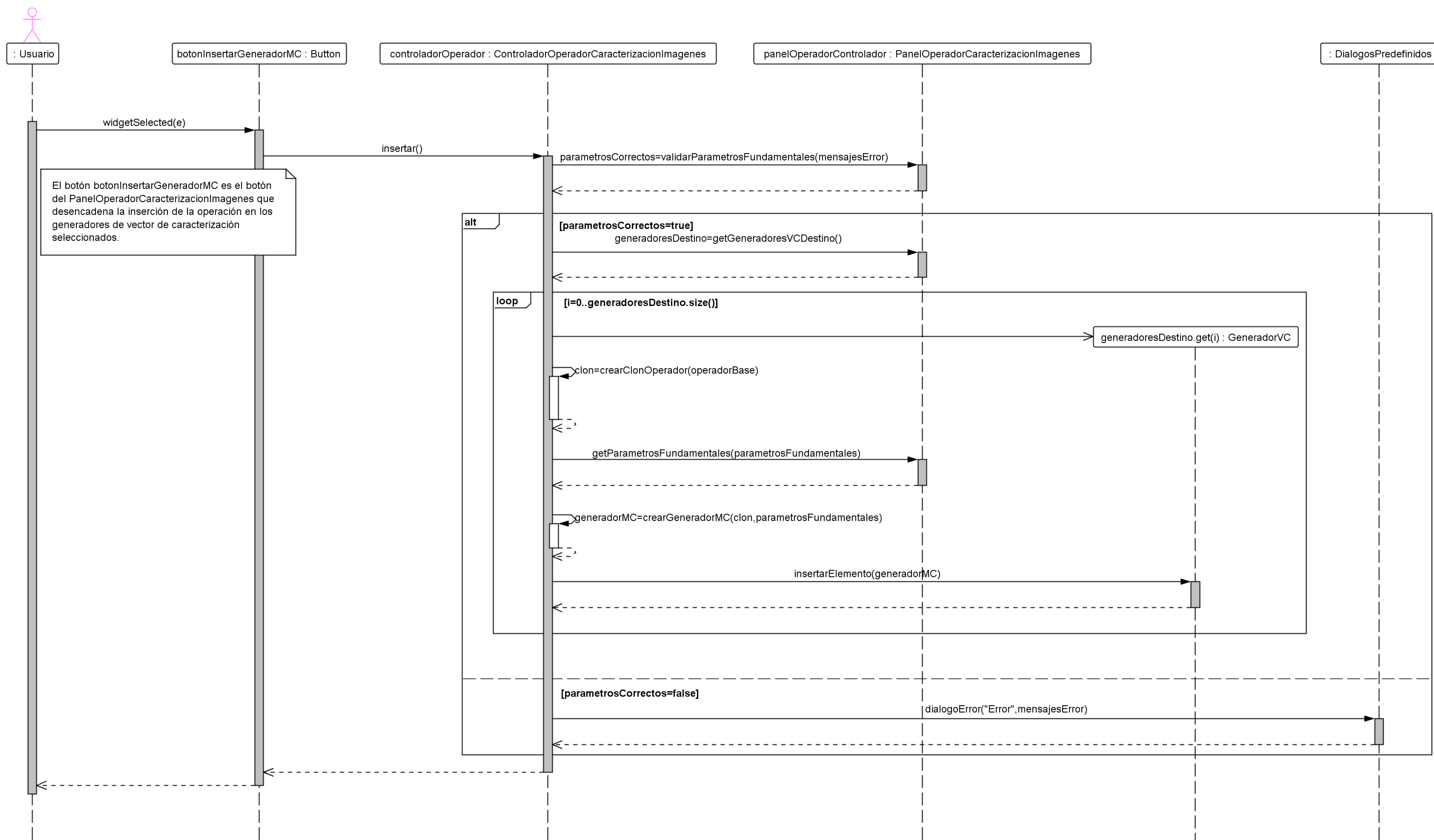

Figura 6.15: Diagrama de secuencia de la inserción de una operación de caracterización de imágenes en varios generadores de vector de caracterización

6.10. Clase Configurador

Antes de arrancar de forma completa, el *plugin coolimaging* debe llevar a cabo una serie de pasos de configuración inicial.

Este proceso de configuración inicial es común en gran cantidad de aplicaciones, donde, por ejemplo, es necesario reestablecer ciertos valores que conviene mantener entre distintas sesiones de uso de la aplicación, como las opciones del clásico menú de *preferencias*.

En particular, la clase de configuración del *plugin coolimaging* se encarga de:

- Cargar las operaciones (*IOperadorAplicacion*) conectadas al punto de extensión *operationApplication* (ver secciónes 6.5.2 y6.1.2).
- Cargar las operaciones de JAI conectadas al punto de extensión *operationApplication* (ver sección 6.1.2).
- Construir los árboles de operaciones de tratamiento y caracterización de imágenes.
- Comprobar la existencia de las bibliotecas JAI y MediaLib.
- Inicializar el *look and feel* nativo para los componentes AWT-Swing empleados.
- Cargar y almacenar variables permanentes (de preferencia).

6.10.1. Carga de las operaciones *IOperadorAplicacion*

Ésta es la labor más importante de la clase *Configurador*. Cool Imaging permite añadir nuevas operaciones al sistema mediante el punto de extensión *operationApplication*. Todos los *plugins* que se conectan a *coolimaging* mediante dicho punto de extensión pueden aportar una o varias operaciones en la forma de clases que implementan la interfaz *IOperadorAplicacion*.

Para cargar las clases que implementan la interfaz *IOperadorAplicacion* se hace uso del mecanismo de puntos de extensión que ofrece la arquitectura Eclipse RCP.

El método estático *getExtensionRegistry()* de la clase *Platform* devuelve un registro con todos los puntos de extensión definidos por el *plugin*. Ese registro, encapsulado en un objeto que implementa la interfaz *IExtensionRegistry*, permite acceder a todos los *elementos de configuración* (*IConfigurationElement*) que han implementado un punto de extensión determinado. Así, la clase *Configurador* pregunta al *IExtensionRegistry* por todos aquellos elementos de configuración que implementan el punto de extensión *operationApplication*. Como respuesta, se devuelve un array de *IConfigurationElement*. Cada elemento del array está asociado a una clase que implementa la interfaz *IOperadorAplicacion*, y permite, de hecho, crear instancias de la clase que implementa dicha interfaz.

Es así como el *plugin coolimaging* consigue cargar las clases de las operaciones registradas a través del punto de extensión *operationApplication*. Todo este proceso es llevado a cabo en el método *cargarClasesIOperadorAplicacion()*, y tiene como resultado la construcción de un listado de objetos *IOperadorAplicacion*, uno por cada una de las operaciones definidas en la aplicación. Realmente, cada uno de los objetos *IOperadorAplicacion* del listado no son usados de forma directa. Estos objetos se usan como contenedores a través de los cuales obtener la clase (objeto *Class*), asociado a cada una de las operaciones. Cada uno de estos objetos *Class* es usado para, posteriormente, crear múltiples instancias de las operaciones.

6.10.2. Carga de las nuevas operaciones de JAI

A través del punto de extensión *operationRegisterJAI*, otros *plugins* pueden añadir operaciones de JAI al sistema. Para ello, el *plugin coolimaging* hace uso del sistema de *plugins* de la arquitectura Eclipse RCP, de una manera análoga a como carga las operaciones de tratamiento y caracterización mediante el punto de extensión *operationApplication*.

En particular, en este caso, el *plugin coolimaging* pregunta por el punto de extensión *operationRegisterJAI*, y obtiene un listado de todos los objetos que están conectados a dicho punto de extensión y que implementan la interfaz *IOperationRegisterJAI*.

Finalmente, para proceder al registrado de cada una de las operaciones, llama al método *register()* en cada uno de los objetos que implementan la interfaz. Todo este proceso es llevado a cabo por la función *cargarNuevosOperadoresJAI()*.

6.10.3. Construcción de los árboles de operaciones

Como resultado de la carga de las operaciones instaladas a través del punto de extensión *operationApplication*, el *plugin coolimaging*(ver sección 6.10.1) obtiene un listado que contiene una instancia de cada uno de los *IOperadorAplicacion* que han sido reconocidos en el punto de extensión.

Una vez que se dispone de todas las operaciones, la creación de los árboles de operaciones de tratamiento y caracterización es bastante simple.

Un árbol de operaciones, representado por la clase *ArbolOperaciones*, contiene dos tipos de nodos. Los principales son los *NodoOperacion*, tal y como se explicaba en la sección 6.7, pues dichos nodos son los que iniciaban el proceso de activación de las operaciones.

El problema es que el árbol de operaciones proporciona una determinada estructura a los nodos del árbol. La idea es poder organizar, en categorías, las distintas operaciones (*NodoOperaciones*). Por tanto, la estructura del árbol debe especificarse de algún modo.

A grandes rasgos, la estructura del árbol se almacena en ficheros XML que indican, para cada operación, la categoría en la que se encuentra. La clase *LectorCarpetaMenu* es la encargada de leer los ficheros XML que contienen la estructura del árbol, y, a partir de dicha estructura, y del listado de las operaciones almacenado por la clase *Configurador*, construir de forma definitiva el *ArbolOperaciones*.

En la sección6.14 se da una descripción más detallada de cómo la clase *LectorCarpetaMenu* se encarga de leer los ficheros XML que contienen la estructura del árbol de operaciones, y como lo va construyendo paso a paso.

Dado que hay dos árboles de operaciones, uno de operaciones de tratamiento y otro de operaciones de caracterización, la clase *LectorCarpetaMenu* se encarga de leer los ficheros XML que contienen la estructura de ambos árboles, y construye ambos objetos *ArbolOperaciones*, que son almacenados en la clase *Configurador* para su posterior uso por parte de las clases *MenuArbolOperacionesTratamientoImagenes* y *MenuArbolOperacionesCaracterizacionImagenes*.

Todo este proceso es llevado a cabo por la función *construirArbolesOperaciones()*.

6.10.4. Comprobación de las bibliotecas JAI y MediaLib

Cool Imaging hace uso de las bibliotecas JAI y MediaLib para el procesamiento de imágenes.

JAI es la biblioteca de procesamiento de imágenes propiamente dicha, y su presencia es necesaria para el funcionamiento correcto de la aplicación. MediaLib es la biblioteca que, a nivel nativo, lleva a cabo las optimizaciones de gran parte de las operaciones de JAI, y puede no estar instalada en el sistema.

El método *comprobarExistenciaJAIyMediaLib()* comprueba la existencia de ambas bibliotecas en el sistema. Si no se encuentra alguna de ellas, se mostrará un mensaje informando al usuario.

6.10.5. Inicialización del *look and feel* de AWT-Swing

Cool Imaging es una aplicación construida sobre Eclipse RCP. Ello la obliga a hacer uso de la biblioteca SWT como sistema de gestión de la interfaz de usuario.

A diferencia de AWT y Swing, los *toolkits* alternativos de interfaces gráficas de usuario en Java, SWT usa el *look and feel* (L&F) nativo del sistema operativo sobre el que se esté ejecutando la aplicación. AWT y Swing usan L&Fs alternativos por defecto. Para imponer homogeneidad en la interfaz gráfica de la aplicación, AWT y Swing han de configurarse para que también hagan uso del L&F nativo del sistema operativo. Este L&F, sin embargo, es una emulación, de modo que en ciertos detalles se puede apreciar la diferencia entre las componentes realmente nativas (SWT), y las emuladas (AWT y Swing).

Para forzar a que AWT y Swing hagan uso del L&F nativo, se hace uso de la función *setLookAndFeel()* de la clase *UIManager*.

Este proceso se lleva a cabo dentro de la función *establecerLoonAndFeelAwtSwing()*.

6.10.6. Lectura y escritura de variables de preferencia

Ciertas variables y opciones de configuración son mantenidas entre distintas sesiones de ejecución de Cool Imaging. Estas variables se conocen como variables de preferencia.

El mecanismo de gestión de escritura y lectura de estas variables es gestionado por las funciones *cargarVariablesPreferencias()* y *guardar()* de la clase *Configurador*.

Internamente, estas funciones hacen uso de un mecanismo que proporciona Eclipse RCP para guardar variables entre distintas sesiones de ejecución. Este mecanismo se basa en el uso de las clases *ConfigurationScope* y *Preferences*.

6.11. Gestión de paquetes de imágenes

Cool Imaging permite la creación de *paquetes de imágenes*. Un *paquete de imágenes* no es más que un repertorio de imágenes agrupados lógicamente bajo un mismo nombre (el nombre del paquete).

La clase *PaqueteImagenes* representa un paquete de imágenes. Un *PaqueteImagenes* no almacena imágenes en estado puro (es decir, no almacena objetos de tipo *Imagen*), pues sería altamente ineficiente en espacio consumido. Por contra, un objeto de la clase *PaqueteImagenes* almacena simplemente los identificadores de las imágenes. La idea de la clase *PaqueteImagenes* es poder acceder posteriormente a las imágenes identificadas a través de sus identificadores. Recuérdese que el identificador de una imagen es el nombre del archivo que la almacena, de modo que a través de su identificador, la imagen puede ser fácilmente recuperada. Un *PaqueteImagenes* tiene asociado un identificador (nombre).

A nivel global existe un repertorio de paquetes de imágenes al que el usuario puede acceder. Estos paquetes son los que el usuario ha creado o bien ha cargado de disco. Este conjunto de paquetes de imágenes global está representado por la clase *ConjuntoPaqueteImagenes*. Esta clase representa un conjunto de objetos de tipo *PaqueteImagenes*.

A través de la clase *GestorPaquetesImagenes*[11] se puede acceder al *ConjuntoPaqueteImagenes* global de la aplicación.

[11] Implementada siguiendo el patrón *singleton*.

La aplicación permite crear y eliminar paquetes de imágenes de forma rápida e intuitiva. La gestión de los paquetes de imágenes se realiza a través de la *VistaPaquetesImagenes*. Esta clase representa un *ViewPart* del Eclipse RCP, es decir, una vista del RCP.

La *VistaPaquetesImagenes*, que se corresponde con la vista definida en la figura 5.8, ofrece un repertorio de elementos gráficos a través de los cuales el usuario puede visualizar los paquetes de imágenes existentes en la aplicación, así como qué imágenes contiene cada paquete. En realidad, la *VistaPaquetesImagenes* muestra únicamente un objeto de tipo *GestorVisualPaquetesImagenesAplicacion*. Es este panel el que realmente contiene los elementos gráficos que se encargan de gestionar la manipulación de los paquetes de imágenes.

Existe tres clases que se encargan de la visualización directa de los paquetes de imágenes. La clase *VistaConjuntoPaqueteImagenes* se corresponde con el panel de la figura 5.11 del *storyboard*, y permite visualizar un listado de todos los paquetes de imágenes almacenados en un *ConjuntoPaqueteImagenes*, así como eliminar y renombrar paquetes, crear nuevos paquetes (que son añadidos al *ConjuntoPaqueteImagenes* y guardar los paquetes mostrados. La clase *VistaPaqueteImagenes*, que se corresponde con el panel de la figura 5.9, permite visualizar los identificadores de las imágenes de un *PaqueteImagenes*, así como eliminar imágenes del paquete. La clase *VistaCoordinadaPaqueteImagenes* muestra dos objetos, uno de tipo *VistaConjuntoPaqueteImagenes* y otro de tipo *VistaPaqueteImagenes*. Cuando se selecciona un paquete en la *VistaConjuntoPaqueteImagenes*, el listado de imágenes del paquete seleccionado es mostrado en la *VistaPaqueteImagenes*.

Las clases *VistaConjuntoPaqueteImagenes* y *ConjuntoPaqueteImagenes* están implementadas siguiendo el patrón *Observer*: cuando se insertan, eliminan o renombran paquetes de imágenes del *ConjuntoPaqueteImagenes*, la *VistaConjuntoPaqueteImagenes* refleja dichos cambios de forma automática.

Igualmente, las clases *VistaPaqueteImagenes* y *PaqueteImagenes* están implementadas siguiendo el patrón *Observer*: cuando se inserta o añade un identificador de imagen al paquete de imágenes, la *VistaPaqueteImagenes* refleja dichos cambios de forma automática.

La clase *GestorVisualPaquetesImagenesAplicacion* está construida fundamentalmente mediante estos controles básicos. Además, el *GestorVisualPaquetesImagenesAplicacion* ofrece diversas funciones públicas que son usadas por las acciones de la cool bar de la *VistaPaquetesImagenes* y que aumentan su interactividad.

La figura 6.16 recoge las clases básicas para la representación de los paquetes de imágenes.

La figura 6.17 presenta el diagrama de la *VistaPaquetesImagenes* así como del *GestorVisualPaquetesImagenesAplicacion*. Obsérvese el detalle de todas las acciones asociadas a la *VistaPaquetesImagenes*. La mayoría de esas acciones, mostradas en la cool bar de la vista, acceden a la funcionalidad pública ofrecida por el *GestorVisualPaquetesImagenesAplicacion*.

A continuación se ofrece una explicación detallada de las clases mencionadas.

6.11.1. PaqueteImagenes

La clase *PaqueteImagenes* representa un repertorio de identificadores de imágenes agrupados bajo un nombre (el nombre del paquete). Un *PaqueteImagenes* tiene un identificador asociado.

La clase *PaqueteImagenes* extiende la clase *Observable* según la filosofía del patrón *Observer*. Todo cambio producido en el *PaqueteImagenes* se ve automáticamente reflejado el *Observer* correspondiente.

6.11.2. VistaPaqueteImagenes

La clase *VistaPaqueteImagenes* representa un panel gráfico que permite visualizar los identificadores de las imágenes presentes en un *PaqueteImagenes*. Esta clase implementa la interfaz

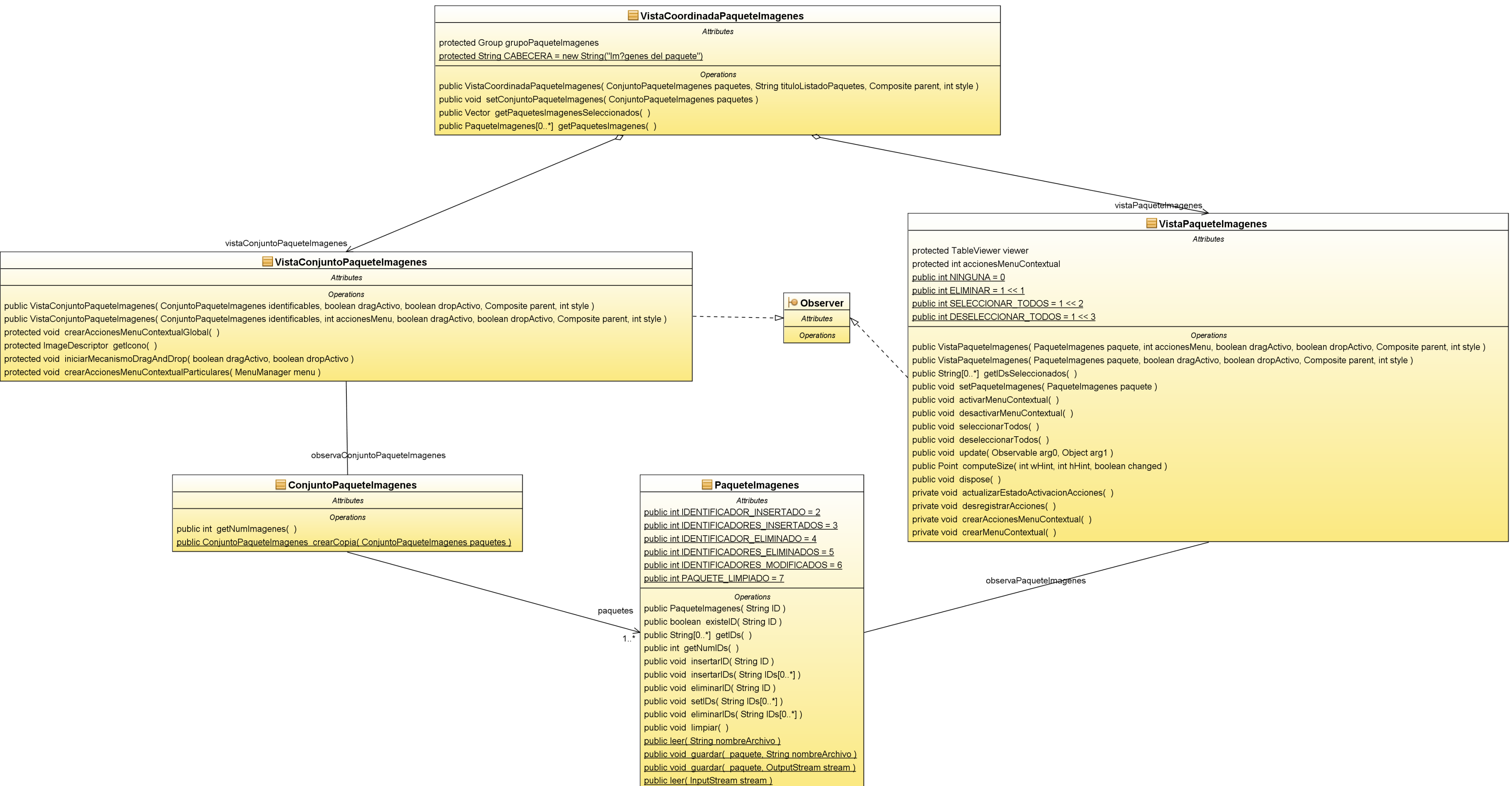

Figura 6.16: Clases básicas para la visualización de paquetes de imágenes

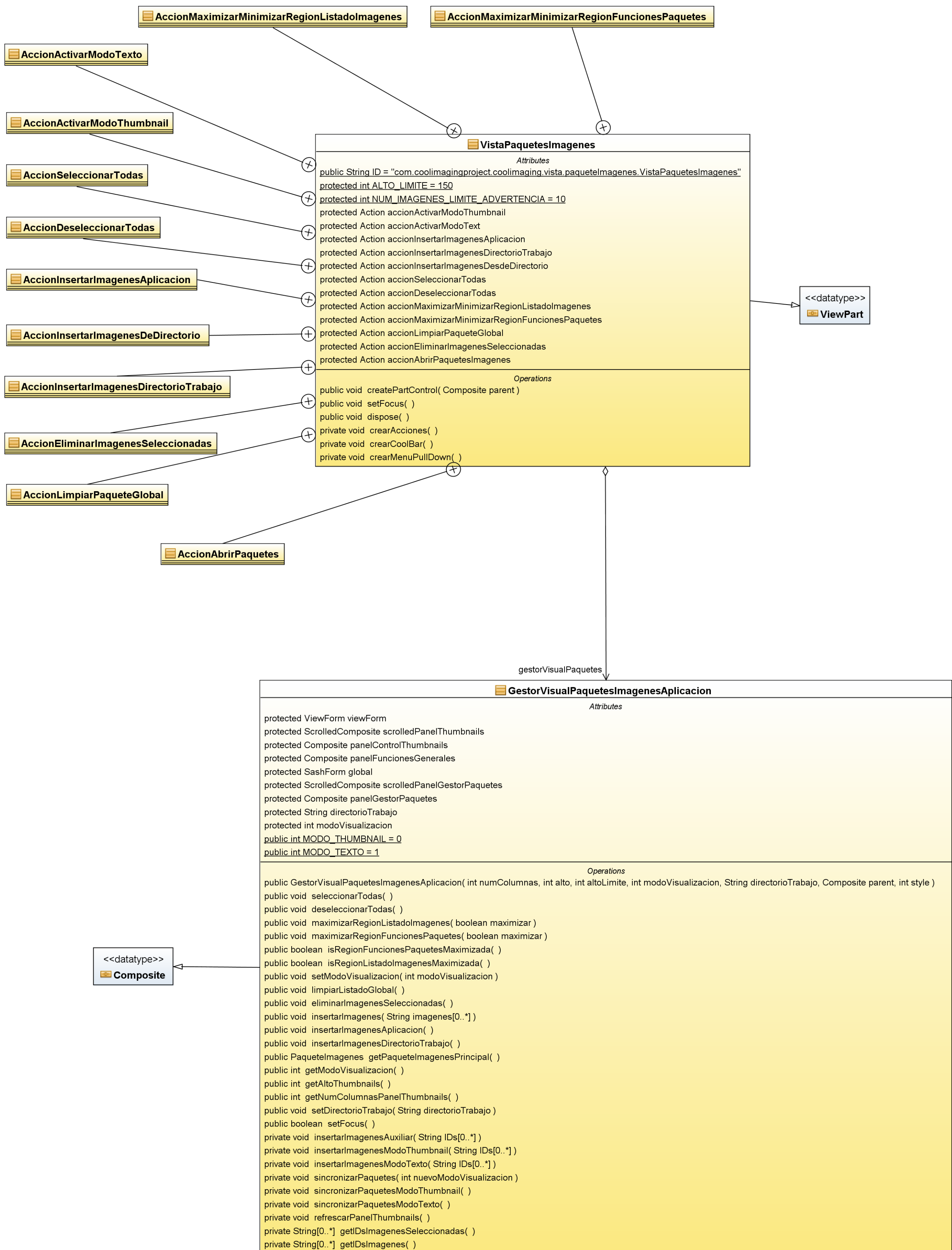

Figura 6.17: Vista de gestión de paquetes de imágenes

Observer, ya que sigue los cambios que se producen en el modelo de datos asociado, el *PaqueteImagenes*. En particular, cuando se insertan o eliminan identificadores del *PaqueteImagenes*, la *VistaPaqueteImagenes* refleja automáticamente dichos cambios.

6.11.3. ConjuntoPaqueteImagenes

La clase *ConjuntoPaqueteImagenes* representa un conjunto de objetos de tipo *PaqueteImagenes*. Esta clase extiende a la clase *Observable* según la filosofía del patrón *Observer*, ya que representa un modelo de datos que tiene la capacidad de ser observado.

6.11.4. VistaConjuntoPaqueteImagenes

La clase *VistaConjuntoPaqueteImagenes* es un panel gráfico con la capacidad de visualizar el listado de todos los paquetes almacenados en un *ConjuntoPaqueteImagenes*. Muestra, concretamente, el identificador de cada uno de los paquetes presentes en el *ConjuntoPaqueteImagenes*. Esta clase implementa la interfaz *Observer*, ya que sigue los cambios producidos en su modelo de datos asociado, el *ConjuntoPaqueteImagenes*. En particular, cuando se insertan, eliminan o renombran paquetes del *ConjuntoPaqueteImagenes*, la *VistaConjuntoPaqueteImagenes* refleja los cambios.

6.11.5. GestorPaquetesImagenes

La aplicación consta de un repertorio global de paquetes de imágenes cargados actualmente. Dicho *ConjuntoPaqueteImagenes* es gestionado por el *GestorPaquetesImagenes*. La clase *GestorPaquetesImagenes*, implementada siguiendo el patrón *singleton*, simplemente permite acceder al *ConjuntoPaqueteImagenes* global de la aplicación.

6.11.6. GestorVisualPaquetesImagenesAplicacion

La clase *GestorVisualPaquetesImagenesAplicacion* representa un panel gráfico a través del cual el usuario puede gestionar la creación y eliminación de los paquetes de imágenes de la aplicación. Este panel muestra controles que permiten crear paquetes, eliminar paquetes y renombrar paquetes.

Además, la clase *GestorVisualPaquetesImagenesAplicacion* ofrece una serie de métodos públicos que permiten extender su funcionalidad. Estos métodos son empleados por la clase *VistaPaquetesImagenes*, a través de las acciones que ofrece en su cool bar.

6.11.7. VistaPaquetesImagenes

La clase *VistaPaquetesImagenes* es una vista del Eclipse RCP (*ViewPart*), que permite al usuario gestionar los paquetes de imágenes de la aplicación. La *VistaPaquetesImagenes* muestra al usuario un *GestorVisualPaquetesImagenesAplicacion*, que se encarga de la mayor parte de la gestión de los paquetes. Además, la vista ofrece a través de su cool bar un repertorio de acciones que aumentan las posibilidades de gestión de los paquetes de imágenes. La mayor parte de estas acciones acceden a los métodos públicos ofrecidos por el *GestorVisualPaquetesImagenesAplicacion*.

6.12. Cadenas de operaciones al detalle. Procesamiento de paquetes de imágenes

En la sección 6.8 se explicó como una operación de tratamiento de imágenes podía ser aplicada de forma directa a una o varias imágenes abiertas en la aplicación. También se explicó cómo una operación de tratamiento de imágenes podía ser insertada en una o varias de las cadena de operaciones (*CadenaOperacionesTratamiento*) definidas en el sistema. Las cadenas de operaciones son especialmente importantes de cara al usuario, pues le permiten procesar paquetes de imágenes a través de secuencias de operaciones que transforman una imagen de entrada en una imagen de salida.

6.12.1. Lógica de procesamiento de paquetes mediante cadenas de operaciones

La clase *CadenaOperacionesTratamiento* representa una secuencia ordenada de operaciones unarias[12]. Una *CadenaOperacionesTratamiento* permite transformar una imagen de entrada en una de salida, mediante la aplicación ordenada de las operaciones que contiene.

Una *CadenaOperacionesTratamiento* almacena objetos de tipo *ElementoCadenaOperacionesTratamiento*. Cada *ElementoCadenaOperacionesTratamiento* almacena un operador unario, es decir, un *IOperadorAplicacion* que implementa una operación unaria. Cada uno de estos *IOperadorAplicacion* representa una de las operaciones que la cadena aplica a la imagen de entrada.

El proceso a través del cual una *CadenaOperacionesTratamiento* transforma una imagen de entrada en una imagen de salida se basa en hacer un uso ordenado de cada uno de los *ElementoCadenaOperacionesTratamiento* que almacena; la imagen de entrada es procesada por el primero *ElementoCadenaOperacionesTratamiento*, produciendo una imagen de salida; esta primera imagen de salida es procesada por el segundo *ElementoCadenaOperacionesTratamiento*, produciendo una segunda imagen de salida, que es procesada por el tercer *ElementoCadenaOperacionesTratamiento*, y así sucesivamente, hasta que se han hecho uso de todos los *ElementoCadenaOperacionesTratamiento* de la cadena. En pseudocódigo, el algoritmo sería similar a:

```
Imagen imagenOriginal=IniciarImagen();
Imagen imagenActual=imagenOriginal;

for(int i=0;i<cadena.numElementos();i++){
   imagenActual=cadena.getElemento(i).procesar(imagenActual);
}

return imagenActual;
```

Sin embargo, la clase *CadenaOperacionesTratamiento* es demasiado simple como para llevar a cabo el procesamiento de paquetes de imágenes.

Es por ello que se crean las clases *ProcesadorPaquetesTratamiento* y *ConfiguracionProcesadorPaquetesTratamiento*. Estas dos clases son las encargadas de procesar paquetes de imágenes y almacenar las imágenes resultantes en una determinada localización de salida.

La clase *ProcesadorPaquetesTratamiento* es la encargada de llevar a cabo el procesamiento de paquetes de imágenes mediante cadenas de operaciones. Esta clase se encarga de cargar cada una de las imágenes de los paquetes a procesar, y procesarlas mediante las cadenas de

[12]Es decir, que requiere una única imagen de entrada.

operaciones especificadas, para obtener así un repertorio de imágenes que es almacenado en un directorio de salida. Hay ciertos aspectos del procesamiento de paquetes que, sin embargo, deben especificarse al *ProcesadorPaquetesTratamiento* para que pueda proceder de forma correcta. Por ejemplo, sería necesario indicar cuál debería ser el directorio de salida de las imágenes procesadas, o cuál debería ser el formato de salida de dichas imágenes. La clase *ConfiguracionProcesadorPaquetesTratamiento* es la encargada de proporcionar esta información al *ProcesadorPaquetesTratamiento*. Cuando el *ProcesadorPaquetesTratamiento* procesa paquetes de imágenes, éste hace uso de un objeto de tipo *ConfiguracionProcesadorPaquetesTratamiento* que le proporciona la información que necesita para completar el procesamiento de forma satisfactoria.

A nivel global de la aplicación existe un repertorio de cadenas de operaciones de tratamiento. La clase *ConjuntoCadenaOperacionesTratamiento* representa un conjunto de objetos *CadenaOperacionesTratamiento*. En todo momento, el sistema mantiene un listado global con todas las cadenas que están actualmente abiertas y listas para ser usadas. Dicho listado es accesible a través de la clase *singleton GestorCadenasOperacionesTratamiento*, la cual provee acceso al *ConjuntoCadenaOperacionesTratamiento* que contiene todas las cadenas globales de la aplicación.

El diagrama de clases de la figura 6.18 describe las clases anteriormente explicadas, y que permiten procesar paquetes de imágenes mediante cadenas de operaciones.

6.12.2. Gestión visual de las cadenas de operaciones

En la sección 6.12.1 se ha explicado el funcionamiento de las clases que permiten el procesamiento de paquetes de imágenes mediante cadenas de operaciones. Resta explicar cómo, al usuario, se le ofrece la posibilidad de trabajar con dichas cadenas de operaciones.

La gestión de las cadenas de operaciones se lleva a cabo en una vista del Eclipse RCP (*ViewPart*). La vista que gestiona las cadenas de operaciones está implementada por la clase *VistaProcesamientoCadenasOperacionesTratamiento*, y equivale a la vista descrita en la figura 5.12 del *storyboard*. Esta vista ofrece al usuario una serie de controles que le permiten procesar varios paquetes de imágenes mediante varias cadenas de operaciones.

La clase *VistaCadenaOperacionesTratamiento* representa gráficamente a una *CadenaOperacionesTratamiento*, y se corresponde con el panel de la figura 5.7. Este panel muestra el listado de operaciones (*ElementoCadenaOperacionesTratamiento*) actualmente almacenadas en la *CadenaOperacionesTratamiento*, y permite, a través de su menú contextual, eliminar elementos (operaciones) de dicho listado. Esta clase, además, está implementada siguiendo el patrón *Observer*, de modo que los cambios que se producen en la *CadenaOperacionesTratamiento* son reflejados de forma automática por la *VistaCadenaOperacionesTratamiento*.

La clase *ConjuntoCadenaOperacionesTratamiento* representa un conjunto de objetos de tipo *CadenaOperacionesTratamiento*, y es representado gráficamente por la clase *VistaConjuntoCadenaOperacionesTratamiento* (que se corresponde con el panel de la figura 5.6 del *storyboard*). Esta clase muestra un listado con el nombre de todas las *CadenaOperacionesTratamiento* almacenadas en el *ConjuntoCadenaOperacionesTratamiento*, y permite, a través de su menú contextual, crear nuevas cadenas, así como renombrar, eliminar y almacenar en disco las cadenas mostrados. Además, implementa el patrón *Observer*, de modo que es capaz de reflejar de forma automática los cambios producidos en el modelo de datos que tiene asociado, el *ConjuntoCadenaOperacionesTratamiento*.

La clase *VistaCoordinadaCadenaOperacionesTratamiento* almacena un objeto de tipo *VistaConjuntoCadenaOperacionesTratamiento* y otro de tipo *VistaCadenaOperacionesTratamiento*. Cuando se selecciona una cadena en la *VistaConjuntoCadenaOperacionesTratamiento*, dicha cadena es visualizada en la *VistaCadenaOperacionesTratamiento* adyacente.

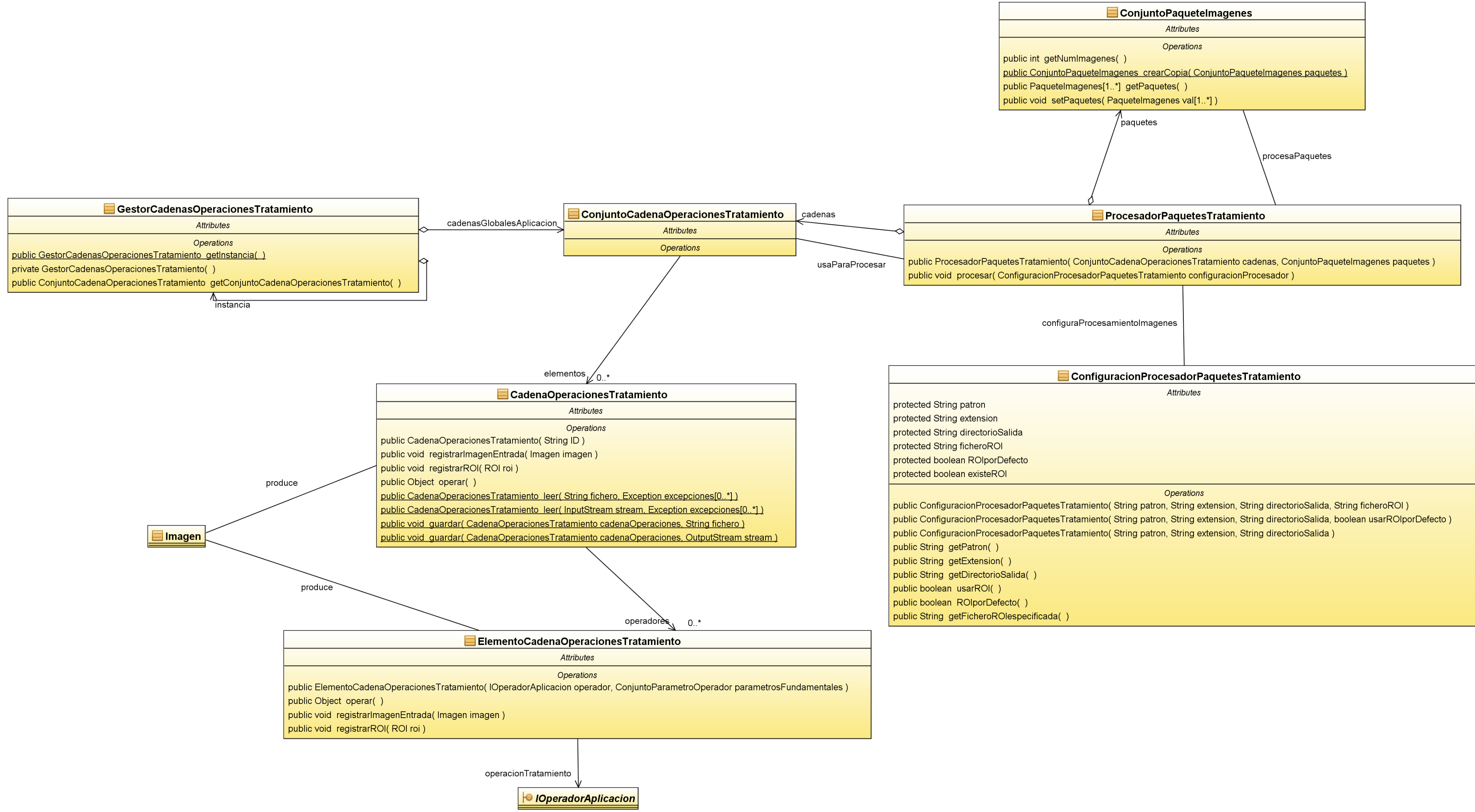

Figura 6.18: Diagrama de clases de la lógica de procesamiento de paquetes de imágenes mediante cadenas de operaciones

La *VistaProcesamientoCadenasOperacionesTratamiento* realmente muestra únicamente un objeto de la clase *GestorVisualCadenasOperacionesTratamiento*. Es la clase *GestorVisualCadenasOperacionesTratamiento* la que, en realidad, contiene todos los elementos gráficos que permiten al usuario manipular cadenas de operaciones.

El contenido gráfico de dicho panel está construido fundamentalmente gracias a las clases anteriormente mencionadas, junto con las que permiten la visualización de paquetes de imágenes (*VistaCoordinadaPaqueteImagenes*, *VistaConjuntoPaqueteImagenes* y *VistaPaqueteImagenes*).

La cool bar de la *VistaProcesamientoCadenasOperacionesTratamiento* ofrece dos acciones al usuario. Una de ellas permite cargar cadenas de operaciones de disco, y asociarlas al conjunto de cadenas global de la aplicación (accesible a través de la clase *GestorCadenasOperacionesTratamiento*). La otra acción desencadena la creación de un *ProcesadorPaquetesTratamiento*, que será el encargado de llevar a cabo el procesamiento de las imágenes. Además, dicha acción desencadena la aparición de un diálogo emergente a través del cual el usuario puede configurar el procesamiento de los paquetes seleccionados, a través de las cadenas seleccionadas. Dicho diálogo es implementado por la clase *DialogoConfiguracionProcesadorPaquetesTratamiento*. A través de dicho diálogo el usuario especifica las opciones de procesamiento (que se transformarán, en última instancia, en un objeto *ConfiguracionProcesadorPaquetesTratamiento*), e indica que quiere realizar el procesamiento. Como resultado, se usa el *ProcesadorPaquetesTratamiento* anteriormente creado con la configuración extraída del panel, lo cual desencadena el procesamiento de las imágenes de los paquetes mediante las cadenas de operaciones especificadas y, en última instancia, el almacenamiento de las imágenes producidas en un cierto directorio de disco.

La figura 6.19 representa el diagrama de clases de la vista de gestión de las cadenas de operaciones, y las principales relaciones con el resto de clases que permiten el procesamiento de paquetes de imágenes mediante cadenas de operaciones.

6.13. Caracterización al detalle. Informes de caracterización y caracterización de paquetes de imágenes

Tal y como se ha explicado en la sección 6.11, dentro de la aplicación pueden crearse paquetes de imágenes (*PaqueteImagenes*) globalmente accesibles mediante el *GestorPaqueteImagenes*. En la sección 6.9, además, se explicaba cómo se podía hacer uso de las operaciones de caracterización: aplicarlas de forma directa a una imagen o bien insertarlas en los generadores de vector de caracterización (*GeneradorVC*) definidos en el sistema. Los generadores de vector de caracterización se usan en el contexto de la creación de informes de caracterización, y cobran especial importancia cuando se usan para caracterizar paquetes completos de imágenes.

6.13.1. Informe de caracterización

La clase *InformeCaracterizacion* representa un informe de caracterización. La clase *InformeCaracterizacion* permite almacenar de forma compacta datos de caracterización obtenidos de imágenes. Un informe de caracterización contiene los vectores de caracterización obtenidos de las imágenes caracterizadas.

Recuérdese que el proceso de caracterización puede llevarse a cabo a nivel de paquetes de imágenes o a nivel de imágenes aisladas (imágenes que no están asociadas a ningún paquete de imágenes).

Si la caracterización se realiza a nivel de paquetes, el informe almacena la correspondencia entre cada imagen y el paquete del que previene. Para cada imagen, además, almacena un listado con todos los vectores de caracterización obtenidos de dicha imagen. Recuérdese que un informe de caracterización puede incluir *medidas de resumen*, calculadas a partir de varios

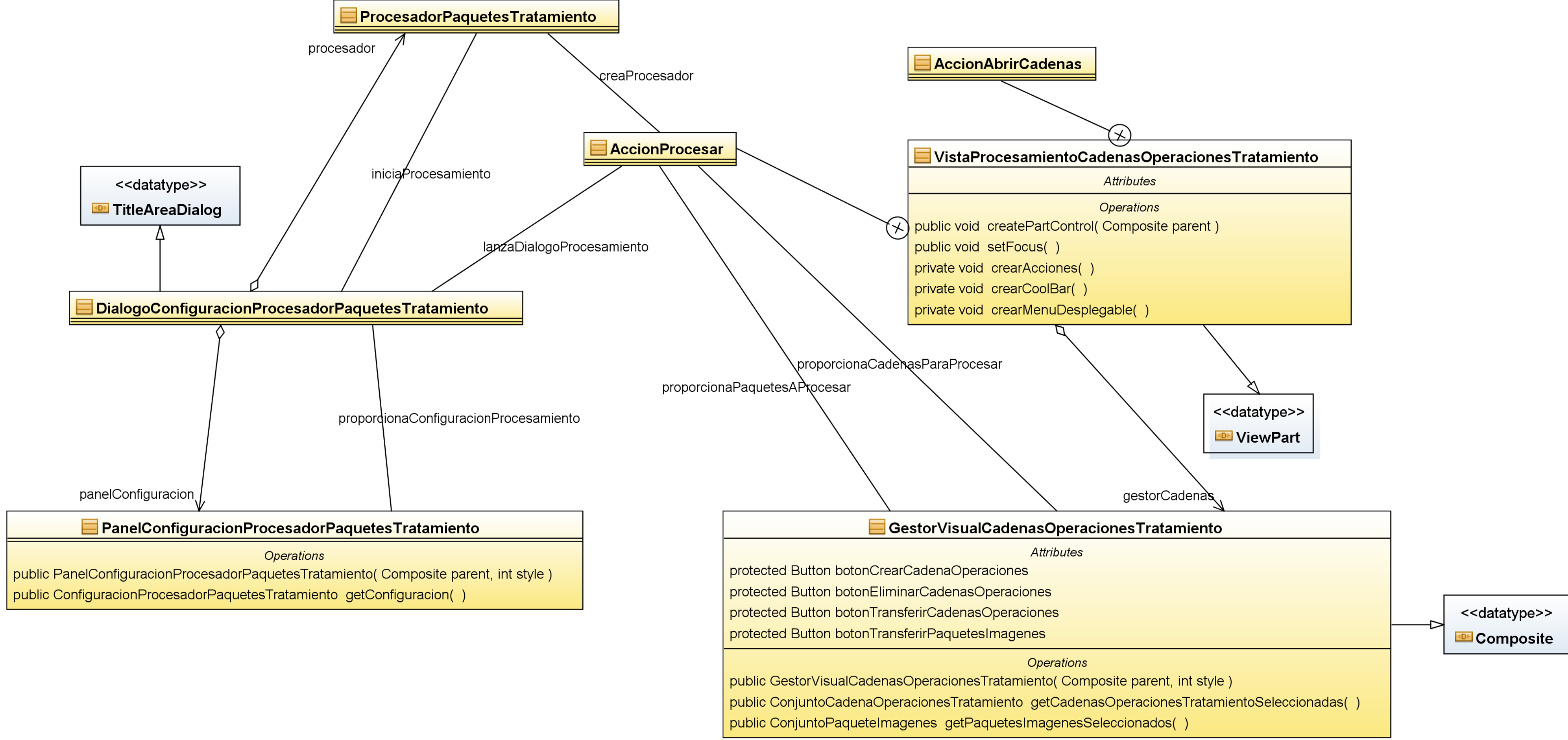

Figura 6.19: Diagrama de clases de la vista de cadenas de operaciones

vectores de caracterización. Cada paquete de imágenes de un informe de caracterización puede incluir varias medidas de resumen de los vectores de caracterización de las imágenes de dicho paquete.

Cuando la caracterización se realiza a nivel de imágenes aisladas, su estructura es similar: almacena, para cada una de las imágenes caracterizadas, un listado con todos los vectores de caracterización obtenidos de dicha imagen. El informe de caracterización puede incluir medidas de resumen de los vectores de caracterización de las imágenes aisladas.

La clase *InformeCaracterizacion* ofrece varios métodos que permiten añadir contenido al informe. Estos métodos, de signatura *registrar()*, permiten insertar una *MC* en el *InformeCaracterizacion*, y asociarla a un vector de una determinada imagen, la cual puede estar o no asociada a un paquete de imágenes.

Un *InformeCaracterizacion* es visualizado a través de un objeto *VistaInformeCaracterizacion*. La clase *VistaInformeCaracterizacion* muestra los datos del *InformeCaracterizacion* de forma estructurada, en forma de tabla. Además, la *VistaInformeCaracterizacion* consta de una región de detalles donde se muestran los detalles del elemento seleccionado actualmente en la tabla. La clase *VistaInformeCaracterizacion*, además, ofrece métodos públicos que permiten ordenar alfabéticamente el listado de datos del informe, filtrar los datos que son mostrados (permitiendo mostrar, por ejemplo, sólo las medidas de resumen del informe), y aumentar o disminuir el número de decimales de los datos mostrados del informe.

Para visualizar el *InformeCaracterizacion*, la *VistaInformeCaracterizacion* hace uso de un *TreeViewer*, al cual le proporcionan contenido y etiquetas un *ContentProvider* y un *LabelProvider*.

Dentro de la aplicación, los *InformeCaracterizacion* son mostrados en editores del Eclipse RCP (*EditorPart*). En particular, los *InformeCaracterizacion* son mostrados en editores de tipo *EditorCaracterizacion*, los cuales, simplemente, almacenan un objeto de tipo *VistaInformeCaracterizacion* a través del cual visualizar el *InformeCaracterizacion* asociado.

A nivel global de la aplicación, existe un listado de todos los informes de caracterización abiertos actualmente.

La clase *ConjuntoInformeCaracterizacion* representa un conjunto de objetos de tipo *InformeCaracterizacion*. A nivel global la aplicación consta de un *ConjuntoInformeCaracterizacion* que contiene todos los informes actualmente abiertos. Dicho conjunto es accesible a través de la clase *ControladorInformesCaracterizacion*, la cual está implementada mediante el patrón *singleton*.

El listado de informes de caracterización es gestionado siguiendo la misma filosofía que el listado global de imágenes de la aplicación (descrito en la sección 6.3).

La clase *ConjuntoInformeCaracterizacion* extiende a la clase *Observable*. Por otro lado, la clase *VistaConjuntoInformeCaracterizacion* implementa la interfaz *Observer*, observando los cambios producidos en su modelo de datos asociado, el *ConjuntoInformeCaracterizacion*. Cuando se insertan informes de caracterización en el *ConjuntoInformeCaracterizacion* global de la aplicación, la clase *VistaConjuntoInformeCaracterizacion* se encarga de crear y abrir un *EditorCaracterizacion* que permita visualizar dicho *InformeCaracterizacion*. Igualmente, cuando se eliminan informes del *ConjuntoInformeCaracterizacion*, la *VistaConjuntoInformeCaracterizacion* cierra el *EditorCaracterizacion* asociado.

El *ControladorInformesCaracterizacion* es un simple controlador de instanciación que asocia el modelo de datos (*ConjuntoInformeCaracterizacion*), con su vista (*VistaConjuntoInformeCaracterizacion*). Además, el *ControladorInformeCaracterizacion* ofrece métodos para acceder y manipular este conjunto de informes de caracterización global de la aplicación.

El diagrama de clases de la figura 6.20 representa las clases descritas y sus relaciones más significativas.

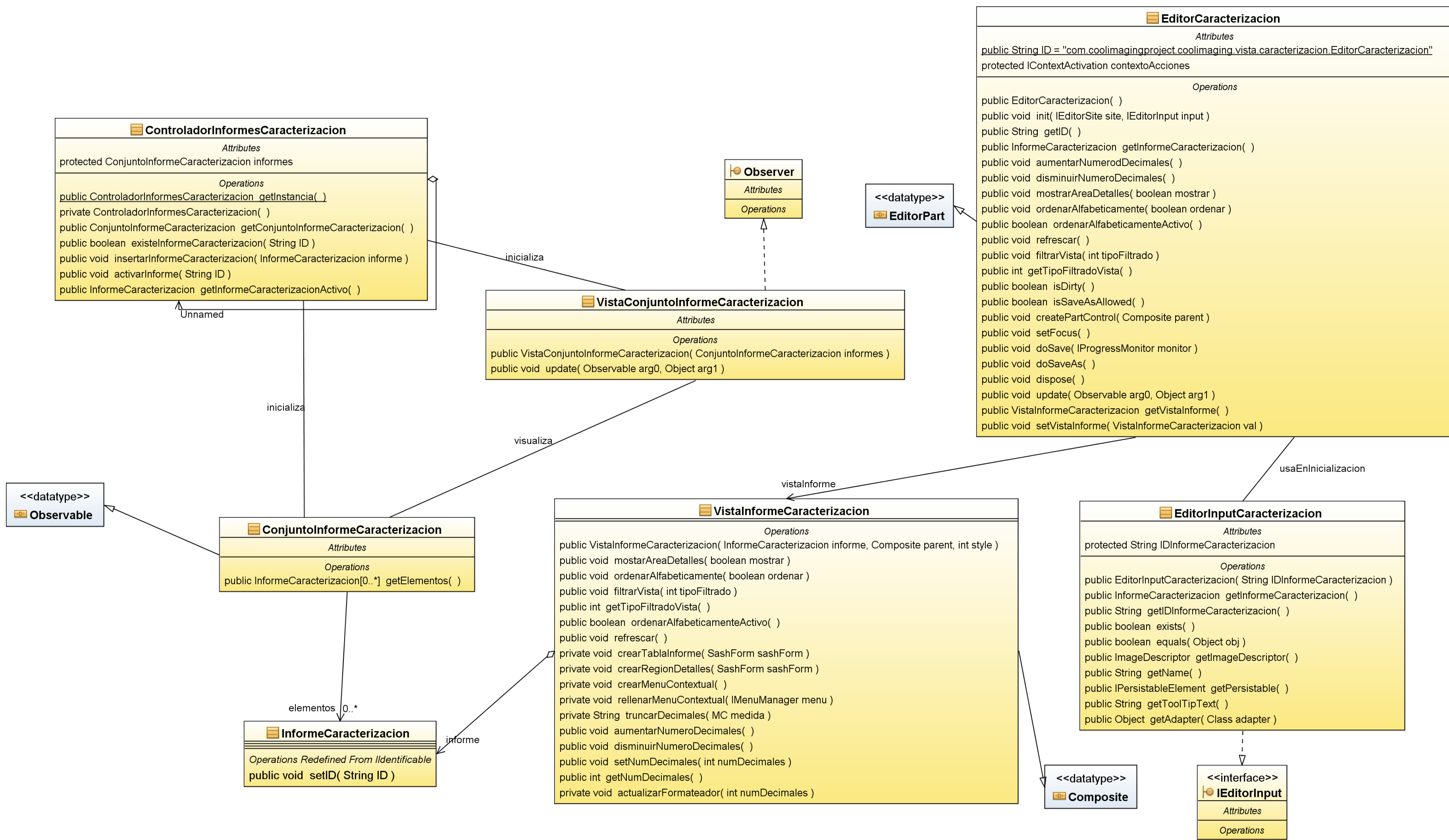

Figura 6.20: Diagrama de clases del control de los informes de caracterización de la aplicación

6.13.2. Lógica de creación de informes de caracterización

Un *GeneradorVC* es una entidad con la capacidad de crear **vectores de caracterización** (*VC*) a partir de imágenes. Un *VC* no es más que un repertorio de **medidas de caracterización** (*MC*). Recuérdese que las operaciones de caracterización (definidas a través de *IOperadorAplicacion*) calculan una *MC* a partir de una imagen de entrada.

Un *GeneradorVC* está formado por objetos de tipo *GeneradorMC*. Un *GeneradorMC* es un objeto con la capacidad de obtener una *MC* sobre una imagen, y contiene, fundamentalmente, un *IOperadorAplicacion* (la operación de caracterización). Para obtener el vector de caracterización de una imagen, el *GeneradorVC* hace uso de cada uno de los *GeneradorMC* que almacena internamente. Cada uno de estos *GeneradorMC* obtiene una medida de caracterización (*MC*) que será añadida al vector de caracterización (*VC*)devuelto. En suma, se puede decir que un *GeneradorVC* es un repertorio de operaciones de caracterización (*IOperadorAplicacion*) a través del cual se construye el vector de caracterización.

La clase *GeneradorVC* es el eje fundamental de la caracterización de imágenes. Cool Imaging, de hecho, da especial protagonismo a los *GeneradorVC*, ya que permite, incluso, almacenarlos en disco para su posterior uso. Así, si el usuario define un *GeneradorVC* con ciertas operaciones, y considera que puede resultarle de utilidad en el futuro, podría almacenarlo en disco y recuperarlo posteriormente.

A nivel global, dentro de la aplicación, existe un listado de todos los generadores de vector de caracterización existentes en la aplicación. La clase *ConjuntoGeneradorVC* representa un listado de objetos *GeneradorVC*. Dentro de la aplicación existe un *ConjuntoGeneradorVC* global, accesible mediante la clase *singleton GestorGeneradoresVC*.

La caracterización de paquetes de imágenes, sin embargo, requiere de una lógica más compleja que la provee la clase *GeneradorVC*.

Las clases *ProcesadorPaquetesCaracterizacion* y *ConfiguracionProcesadorPaquetesCaracterizacion* son las encargadas de gestionar la caracterización de paquetes de imágenes. El objetivo de estas clases es el de construir un informe de caracterización (*InformeCaracterizacion*) que contenga todos los datos de caracterización de los paquetes de imágenes caracterizados.

La clase *ProcesadorPaquetesCaracterizacion* es la encargada de llevar a cabo la caracterización en sí de los paquetes de imágenes. Esta clase se encarga de cargar cada una de las imágenes de los paquetes de imágenes a caracterizar, caracterizarlas, y construir un *InformeCaracterizacion* que aglutine todos los datos de caracterización producidos. Sin embargo, hay ciertos parámetros que determinan cómo se lleva a cabo este proceso de caracterización. Por ejemplo, se podría especificar que la caracterización de cada imagen se hiciera sobre toda la imagen, o bien sobre una región de interés de ésta. La clase *ConfiguracionProcesadorPaquetesCaracterizacion* se encarga de proporcionar estos parámetros de configuración, a través de los cuales el *ProcesadorPaquetesCaracterizacion* sabrá perfectamente como proceder en la caracterización de los paquetes.

El diagrama de clases de la figura 6.21 se visualizan las clases descritas y las principales relaciones entre ellas.

6.13.3. Gestión visual de la caracterización de paquetes

En las secciones 6.13.1 y 6.13.2 se han detallado las clases que forman el modelo de datos que permite la creación de informes de caracterización. Resta explicar cómo, al usuario, se le da la posibilidad de caracterizar paquetes de imágenes, y con ello, crear completos informes de caracterización.

La gestión de la caracterización de paquetes de imágenes se lleva cabo en una vista del Eclipse RCP (*ViewPart*). La vista que gestiona la caracterización está implementada por la clase *VistaCaracterizacion* (equivalente a la vista descrita en las figuras 5.19 y 5.20 del *storyboard*),

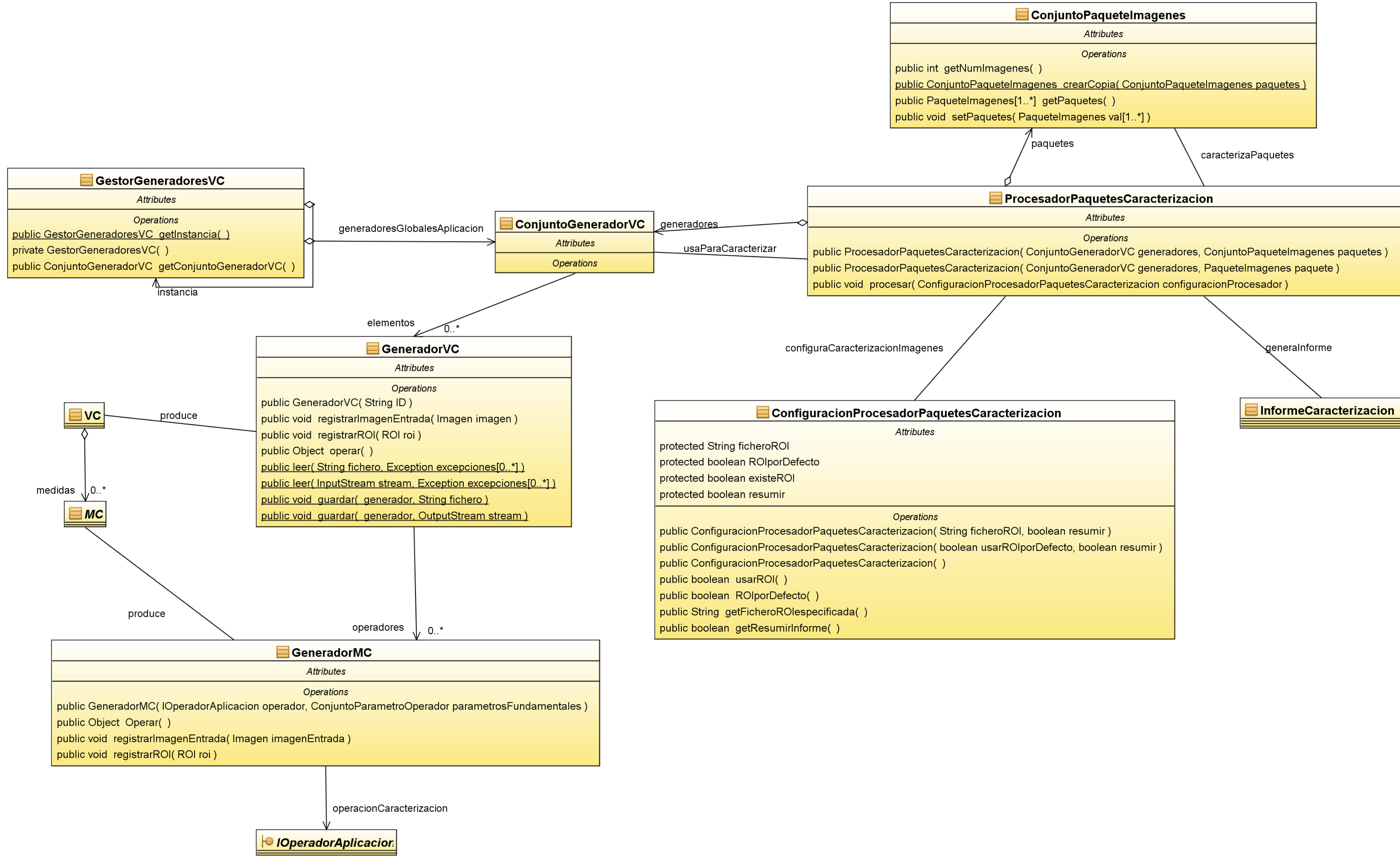

Figura 6.21: Diagrama de clases de la lógica de caracterización de imágenes

y ofrece al usuario una serie de controles que le permiten caracterizar paquetes de imágenes mediante uno o varios generadores de vector de caracterización.

La clase *VistaGeneradorVC* representa gráficamente un *GeneradorVC*, y se corresponde con el panel de la figura 5.18. Este panel muestra el listado de operaciones (*GeneradorMC*) actualmente almacenadas en el *GeneradorVC*, y permite, a través de su menú contextual, eliminar elementos (operaciones) de dicho listado. Esta clase, además, está implementada siguiendo el patrón *Observer*, de modo que los cambios que se producen en el *GeneradorVC* son reflejados de forma automática por la *VistaGeneradorVC*.

La clase *ConjuntoGeneradorVC* representa un conjunto de objetos de tipo *GeneradorVC*, y es representado gráficamente por la clase *VistaConjuntoGeneradorVC* (que se corresponde con el panel de la figura 5.17 del *storyboard*). Esta clase muestra un listado con el nombre de todos los *GeneradorVC* almacenados en el *ConjuntoGeneradorVC*, y permite, a través de su menú contextual, crear nuevos generadores, así como renombrar, eliminar y almacenar en disco los generadores mostrados. Además, implementa el patrón *Observer*, de modo que es capaz de reflejar de forma automática los cambios producidos en el modelo de datos que tiene asociado, el *ConjuntoGeneradorVC*.

La clase *VistaCoordinadaGeneradorVC* almacena un objeto de tipo *VistaConjuntoGeneradorVC* y otro de tipo *VistaGeneradorVC*. Cuando se selecciona un generador en la *VistaConjuntoGeneradorVC*, dicho generador es visualizado en la *VistaGeneradorVC* adyacente.

La *VistaCaracterizacion* realmente muestra únicamente un objeto de la clase *GestorVisualCaracterizacion*. Es la clase *GestorVisualCaracterizacion* la que, en realidad, contiene todos los elementos gráficos que permiten al usuario caracterizar imágenes.

El contenido gráfico de dicho panel está construido fundamentalmente gracias a las clases anteriormente mencionadas, junto con las que permiten la visualización de paquetes de imágenes (*VistaCoordinadaPaqueteImagenes*, *VistaConjuntoPaqueteImagenes* y *VistaPaqueteImagenes*).

La *VistaCaracterizacion* ofrece dos modos de funcionamiento. Uno permite caracterizar paquetes de imágenes, y otro permite caracterizar un listado de imágenes no asociadas a ningún paquete.

La cool bar de la *VistaCaracteriazacion* ofrece dos acciones al usuario. Una de ellas permite cargar generadores de vector de caracterización de disco, y asociarlos al conjunto de generadores global de la aplicación (accesible a través de la clase *GestorGeneradoresVC*). La otra acción desencadena la creación de un *ProcesadorPaquetesCaracterizacion*, que será el encargado de llevar a cabo la caracterización de las imágenes. Además, dicha acción desencadena la aparición de un diálogo emergente a través del cual el usuario puede configurar la caracterización de las imágenes o paquetes seleccionados, a través de los generadores seleccionados. Dicho diálogo es implementado por la clase *DialogoConfiguracionProcesadorPaquetesCaracterizacion*. A través de dicho diálogo el usuario especifica las opciones de caracterización (que se transformarán, en última instancia, en un objeto *ConfiguracionProcesadorPaquetesCaracterizacion*), e indica que quiere realizar la caracterización. Como resultado, se usa el *ProcesadorPaquetesCaracterizacion* anteriormente creado con la configuración extraída del panel, lo cual desencadena la caracterización de las imágenes y, en última instancia, la creación del *InformeCaracterizacion*, que es añadido al repertorio de informes globales de la aplicación (accesible a través del *ControladorInformesCaracterizacion*).

En la cool bar, además, el usuario puede especificar el modo de funcionamiento de la vista: caracterización de imágenes o de paquetes de imágenes.

La figura 6.22 representa el diagrama de clases de la vista de caracterización, y las principales relaciones con el resto de clases que permiten la caracterización y creación de informes de caracterización.

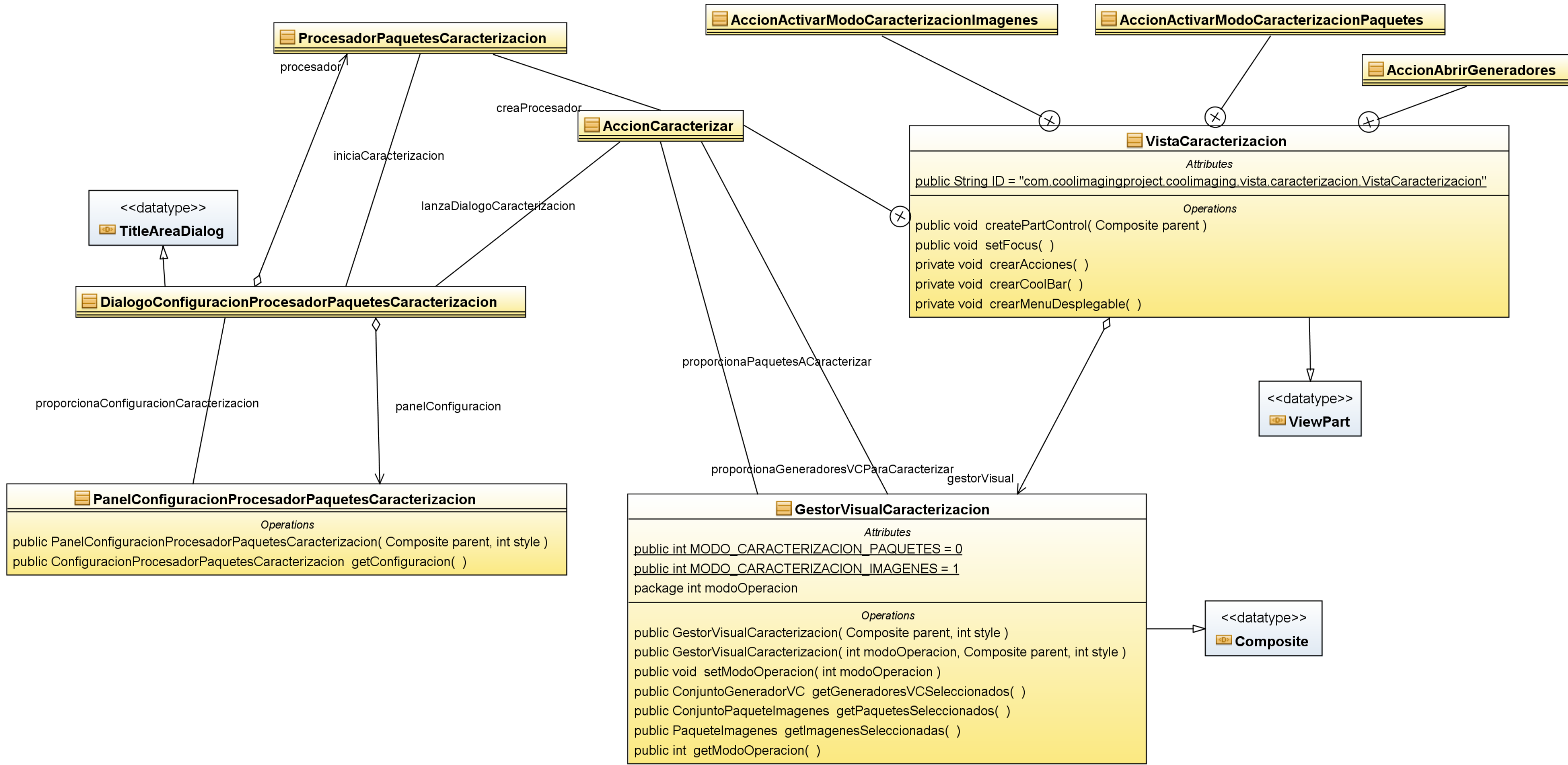

Figura 6.22: Diagrama de clases de la vista de caracterización

6.14. Lector XML del menú de las operaciones

Basándonos en la idea de desarrollar una aplicación extensible donde es fácil añadir nuevas operaciones debido a la estructura de *plugins* desarrollada (sección 6.1.2), necesitabamos crear un modelo donde también fuese fácil añadir y organizar estas nuevas operaciones de cara al usuario. El modelo elegido ha sido una estructura de árbol, donde a nivel de usuario cada operación no es más que un elemento del menú árbol de operaciones (sección 6.6.1); y a nivel de programación, toda operación no es más que una clase que implementa la interfaz *IOperadorAplicacion* y extiende el punto de extensión *operationApplication* (sección 6.5), y que, *de algún modo*, es cargada por la aplicación. Asumiendo estos requisitos, se pensó en la creación de un modelo de control que permitiera especificar, en un archivo de texto plano, la estructura del menú de las operaciones que el usuario desease. Este archivo de texto permitiría especificar fundamentalmente, para cada operación que se desease añadir a la aplicación, tanto la clase que implementara la interfaz *IOperadorAplicacion*, como la categoría dentro del árbol de las operaciones donde dicha operación sería situada. Por tanto, se pensó en la implementación de un modelo basado en el lenguaje *XML (eXtensible Markup Language)*, el cual almacenaría la estructura del menú de operaciones de la aplicación, así como la localización de cada una de las clases *IOperadorAplicacion* que conformarían las operaciones del sistema.

En esta sección describiremos qué son los documentos XML, la estructura XML elegida para nuestro problema concreto, y las clases diseñadas para implementar toda esta estructura.

6.14.1. XML y DTD

El lenguaje XML es un metalenguaje extensible de etiquetas, que permite la creación de lenguajes de etiquetas *a medida*, los cuales satisfagan una determinada necesidad de información. En nuestro caso, se desea hacer uso del lenguaje XML para definir la estructura sintáctica y semántica del fichero de texto plano que almacene la estructura del menú en árbol de las operaciones. El archivo de texto que defina el menú estará escrito siguiendo la sintaxis definida por nuestro lenguaje de etiquetas.

Los documentos XML pueden tener asociados documentos de tipo DTD (*Document Type Definition*), los cuales representan la estructura y sintaxis que se quiere dar al lenguaje de etiquetas definido. Su función básica es la descripción del formato de datos, para usar un formato común y mantener la consistencia entre todos los documentos que utilicen la misma DTD.

6.14.2. DTD elegido y estructura XML

Sabiendo qué son los documentos DTD y XML, pasaremos a definir los DTDs elegidos y pondremos un ejemplo de un posible documento XML que puede ser interpretado por nuestra aplicación.

6.14.2.1. DTD

La aplicación consta de dos menús de operaciones, el menú de operaciones de tratamiento de imágenes y el menú de operaciones caracterización de imágenes, que se diferencian por el tipo de operaciones que contienen. El menú de operaciones de tratamiento alberga operaciones de tratamiento de imágenes, como su nombre indica, al igual que el de caracterización contiene operaciones de caracterización de imágenes. Por este motivo, esta aplicación consta de dos DTDs: uno que define la estructura del menú de tratamiento y otro DTD para definir el menú de caracterización.

```
<!ELEMENT image_processing_index (operation | category)* >
<!ELEMENT category (name+, (category | operation)* )>
<!ELEMENT operation (class_location)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT class_location (#PCDATA)>

<!ATTLIST name lang CDATA #IMPLIED>
<!ATTLIST name country CDATA #IMPLIED>
<!ATTLIST name variant CDATA #IMPLIED>
```

Figura 6.23: DTD: procesamiento de imágenes

```
<!ELEMENT image_characterizing_index (operation | category)* >
<!ELEMENT category (name+, (category | operation)* )>
<!ELEMENT operation (class_location)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT class_location (#PCDATA)>

<!ATTLIST name lang CDATA #IMPLIED>
<!ATTLIST name country CDATA #IMPLIED>
<!ATTLIST name variant CDATA #IMPLIED>
```

Figura 6.24: DTD: caracterización de imágenes

Ambos DTDs son equivalentes en su semántica, y prácticamente idénticos en su sintáctica, a excepción que se diferencian en su nodo raíz. De esta forma al crear los documentos XML, la aplicación puede saber si está trabajando con operaciones de tratamiento o caracterización de imágenes, y así situarlas en su menú correspondiente, o en caso de algún error, poder notificárselo al usuario. Se ha decidido utilizar esta pequeña diferenciación por dos motivos principalmente: para que al usuario que implemente estos menús no le resulte un coste adicional en aprendizaje la implementación de un tipo de menú u otro, y para que la aplicación pueda discernir entre los diferentes tipos de menús.

Para poder comprender un DTD, se debe saber cuáles son los elementos que pueden formar parte de ellos. Puesto que no es nuestro cometido explicar la estructura de un DTD[13] ni sus componentes principales, pasaremos a explicar el ejemplo mostrado en la figura 6.25, que nos hará comprender de una forma más sencilla la estructura que debe cumplir un documento XML para formar parte de uno de los menús de operaciones de la aplicación.

6.14.2.2. Estructura XML

Todo documento XML que vaya a formar parte de uno de los menús de operaciones de la aplicación debe seguir las restricciones de uno de los DTDs que se han definido en las figuras 6.23 y 6.24. Como hemos dicho anteriormente, ambos DTDs son equivalentes, por lo que explicaremos con el ejemplo de la figura 6.25 la estructura de un posible documento XML para un menú de operaciones de tratamiento, y comentaremos las diferencias con respecto al documento XML de la figura 6.28 para un posible menú de caracterización.

Cualquier documento debe comenzar con las dos primeras líneas que se muestran en la figura 6.26, donde se define la versión del documento XML y el DTD que deben cumplir. En

[13]En Internet se pueden encontrar numerosas referencias al término DTD así como tutoriales explicativos.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE image_processing_index SYSTEM
"index/dtd/definitionImageProcessing.dtd">
<image_processing_index>
  <category>
    <name>Geometric transformation</name>
    <name lang="es">Transformaci\u00F3n geom\u00E9trica</name>
    <name lang="en">Geometric transformation</name>
    <operation>
      <class_location>com.coolimagingproject.basicImageProcessingOperations.
      operadorAplicacion.operadoresTratamientoImagenes.
      OperadorEscaladoAplicacion</class_location>
    </operation>
    <operation>
      <class_location>com.coolimagingproject.basicImageProcessingOperations.
      operadorAplicacion.operadoresTratamientoImagenes.
      OperadorTransposicionAplicacion</class_location>
    </operation>
  </category>
  <operation>
    <class_location>com.coolimagingproject.basicImageProcessingOperations.
    operadorAplicacion.operadoresTratamientoImagenes.
    OperadorExtraerBandaAplicacion</class_location>
  </operation>
</image_processing_index>
```

Figura 6.25: Ejemplo de menu XML de procesamiento de imágenes

este caso, hemos supuesto que se debe cumplir el DTD del menú de procesamiento de imágenes (figura 6.23) y que éste está almacenado en el fichero *index/dtd/definitionImageProcessing.dtd*.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE image_processing_index SYSTEM
"index/dtd/definitionImageProcessing.dtd">
```

Figura 6.26: Lineas al comienzo del documento XML

A continuación, debe aparecer un único elemento raíz llamado `<image_processing_index>`. Dentro de este elemento raíz pueden aparecer tantos elementos `<operation>` como `<category>` necesitemos para representar el menú, sin necesidad de guardar ningún orden entre ellos.

Un elemento `<category>` debe contener al menos un elemento `<name>` de forma obligatoria, que representa el nombre de la categoría; éste, a su vez, puede contener tantos elementos `<operation>` o `<category>` como necesitemos. El elemento `<name>`, además puede ir acompañado de varios atributos (`lang`, `country` y `variant`) utilizados para poder *internacionalizar* los índices. De esta manera, el valor de la etiqueta `<name>` dependiendo del valor de los atributos que la acompañen puede contener el nombre de la categoría de forma general, al nombre de la categoría de la variante de un idioma de un país. Por lo que distintas etiquetas `<name>` acompañadas de distintos valores con los distintos atributos, representan distintos nombres de la misma categoría para distintos idiomas, de distintos países, e incluso diferentes variantes.

Un elemento <operation> debe contener un único elemento <class_location> que representa la ubicación de la clase *IOperadorAplicacion* que implementa la operación.

La estructura aquí descrita también puede ser utilizada para los menús de caracterización, como el de la figura 6.28, con dos salvedades. La primera de ellas es que el DTD que debe cumplir el documento XML no será el mismo, por lo que tendremos que cambiar la segunda línea de la cabecera (figura 6.27) por una nueva línea donde se especifique que se cumple el DTD de caracterización, y la ubicación de este nuevo DTD, en este caso suponemos que se encuentra almacenado en el fichero *index/dtd/definitionImageCharacterizing.dtd*. La segunda salvedad será el nodo raíz, que en vez de ser el nodo <image_processing_index>, será el <image_characterizing_index>.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE image_characterizing_index SYSTEM
"index/dtd/definitionImageCharacterizing.dtd">
```

Figura 6.27: Lineas al comienzo del documento XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE image_characterizing_index SYSTEM
"index/dtd/definitionImageCharacterizing.dtd">
<image_characterizing_index>
  <category>
    <name>Co-ocurrence matrix</name>
    <name lang="es">Matriz de co-ocurrencia</name>
    <name lang="en">Co-ocurrence matrix</name>
    <operation>
      <class_location>com.coolimagingproject.imageCharacterizingOperations.
      operadoresAplicacion.matrizCoocurrencia.
      OperadorMatrizCoocurrenciaAplicacionMedia</class_location>
    </operation>
    <operation>
      <class_location>com.coolimagingproject.imageCharacterizingOperations.
      operadoresAplicacion.matrizCoocurrencia.
      OperadorMatrizCoocurrenciaAplicacionUniformidad</class_location>
    </operation>
  </category>
  <operation>
    <class_location>com.coolimagingproject.imageCharacterizingOperations.
    operadoresAplicacion.general.OperadorMediaAplicacion</class_location>
  </operation>
</image_characterizing_index>
```

Figura 6.28: Ejemplo de menu XML de caracterización de imágenes

6.14.3. Lector XML

6.14.3.1. API XML elegida

Una vez explicada la estructura XML que vamos a utilizar, necesitamos saber cómo vamos a leer estos documentos dentro de nuestra aplicación.

Existen diversas tecnologías *Java* para la manipulación de documentos XML, de entre las cuales destacan:

- SAX (Simple API for XML)
- DOM (Document Object Model API from W3C)
- XSLT (XML Style Sheet Language Transformations from W3C)

Cada una de estas tecnologías consta de una serie de ventajas y desventajas, pero quizás la que mejor se amolda a nuestras necesidades es la tecnología SAX[14], por ofrecernos una API simple, con un bajo uso de memoria y basada en eventos.

6.14.3.2. Diagrama de clases

A partir de la tecnología XML elegida, hemos diseñado un sistema que se encarga de buscar en una carpeta del sistema local de archivos todos los documentos XML. De cada uno de estos documentos XML, se comprueba que cumpla el DTD del menú de operaciones correspondiente, y si estos documentos cumplen el DTD, entonces se pasa a leer uno por uno para formar el menú. El diagrama de clases elegido se encuentra en la figura 6.29.

6.14.3.3. XMLTAG

Esta clase alberga todas las etiquetas que se necesitan para poder leer un documento XML encargado de definir el menú de operaciones.

6.14.3.4. ArbolOperacionesHandler

Esta clase se ocupa de albergar el manejador de eventos. Encargada de procesar las etiquetas XML que se encuentran definidas en la clase *XMLTAG* y de utilizar la clase *ArbolOperaciones* para crear el árbol asociado a los documentos XML que se procesan. **Esta es la clase que, en definitiva, construye los árboles de operaciones de la aplicación**.

6.14.3.5. LectorArbolOperaciones

Esta clase es la encargada de leer el documento XML y comprobar que cumple el DTD. Si el documento está bien formado, se encarga de llamar a la clase *LectorArbolOperacionesHandler*, que es quién lee la estructura del documento XML en sí.

6.14.3.6. LectorCarpetaMenu

Esta es la clase encargada de abrir una carpeta y buscar todos los documentos XML que se encuentren dentro de ésta. Dentro de la carpeta, selecciona todos aquellos documentos que sean XML y se encarga de llamar a la clase *LectorArbolOperaciones* para que los abra y se ocupe de montar el árbol de operaciones que se utilizará en la aplicación.

[14] http://www.saxproject.org/.

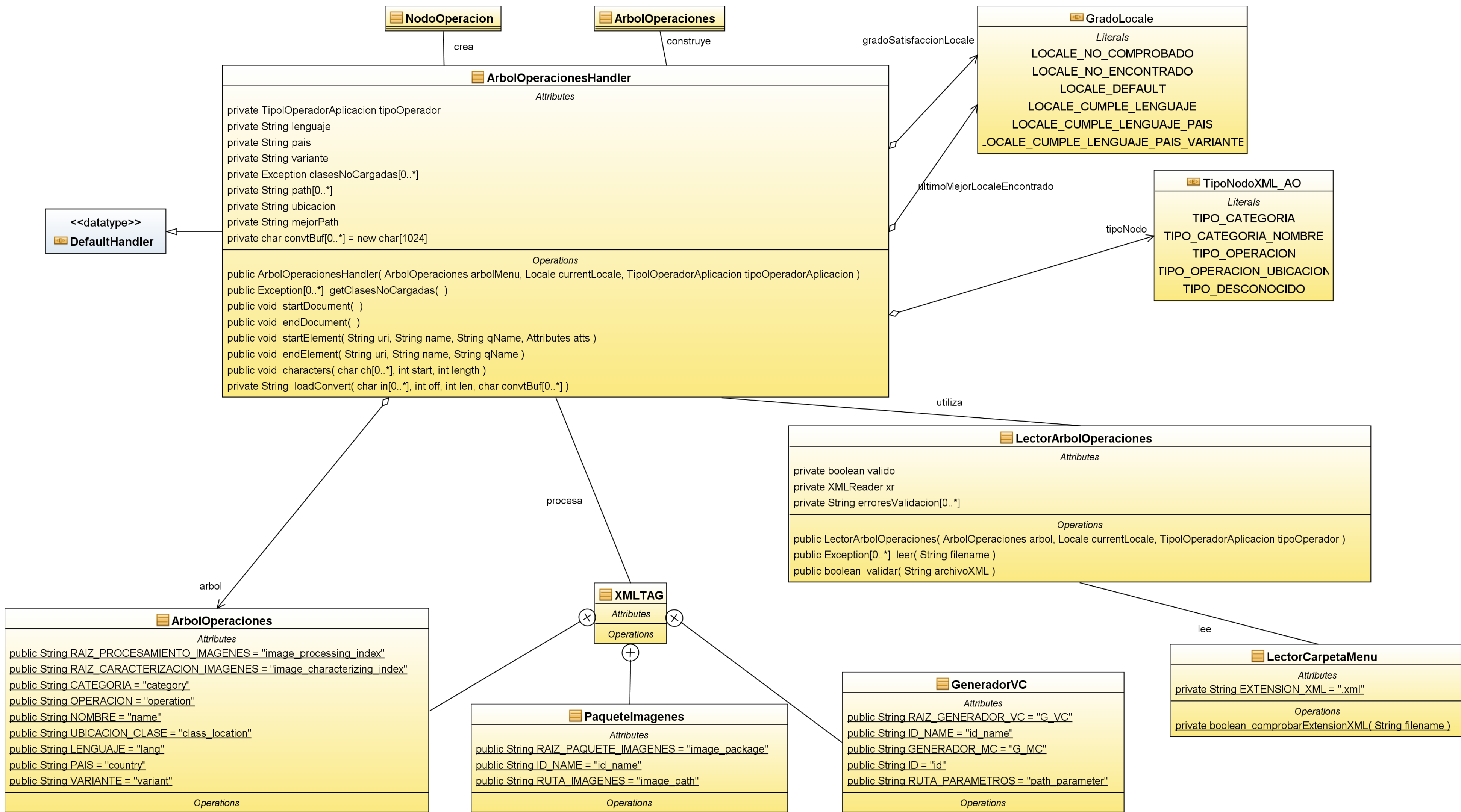

Figura 6.29: Diagrama de clases del Lector XML

6.15. Internacionalización

La internacionalización se define como el proceso de diseñar software de manera tal que pueda adaptarse a diferentes idiomas y regiones sin necesidad de recompilar la aplicación. Esta aplicación ha sido diseñada con este fin, de tal manera que se han separado todas las cadenas de caracteres y rutas de recursos suceptibles a cambios por idioma o región, a ficheros independientes a los del código fuente.

Para realizar esta tarea nos hemos apoyado en las herramientas ofrecidas por la plataforma Eclipse, consistiendo el proceso en separar todas las cadenas de caracteres y rutas de recursos en archivos, donde cada fila de estos archivos almacena un par clave-valor. Estos ficheros, llamados *properties*, son *interpretados* por una clase hija de la clase NLS, clase que proporciona la plataforma Eclipse para la manipulación de mensajes, y que contiene un listado de las claves, que comparte con el fichero *properties*. Para terminar, se deben situar las claves de la clase hija de la clase NLS en el código, donde se quiera sustraer una cadena de texto o ruta de recurso para su internacionalización.

Por tanto, lo que se consigue al ejecutar la aplicación es que la clase NLS se encargue de sustituir cada clave por su correspondiente valor del archivo *properties*, de tal manera que se consigue tener en varios archivos localizados toda la información relevante a ser internacionalizada.

6.16. Barra de estado

Dentro de una aplicación RCP, toda ventana de trabajo tiene asociada una barra de estado. La barra de estado se muestra en la parte inferior de la ventana, y en ella se pueden mostrar tanto mensajes como elementos más complejos.

La barra es controlada a través de la interfaz *IStatusLineManager*. El *IStatusLineManager* asociado a una ventana se obtiene a partir del *IActionBarConfigurer* de la misma ventana.

Para facilitar la gestión de la barra de estado, dentro del sistema se ha creado la clase *ControladorInfoBarraEstado* (implementada mediante el patrón *singleton*).

Cool Imaging no hace un gran uso de la barra de estado, ya que es poca la información que necesita mostrar. Es por ello que la clase *ControladorInfoBarraEstado* ofrece una interfaz bastante simple, que permite mostrar tanto la información asociada a un píxel de una imagen, como el factor de zoom de una imagen.

Cuando el usuario mueve el ratón sobre una imagen, la barra de estado muestra la información asociada al píxel actual: posición, nombre de cada componente del color, valor de las componentes del píxel y valor de las componentes del color del píxel. De ello se encarga la clase *PanelImagenInteractivo*. Del mismo modo que el *PanelImagenInteractivo* definía un *mouse listener*, el *MouseAdapterInteractividad*, que gestionaba el filtrado de eventos del ratón acorde a la función de interactividad seleccionada, también define un *MouseAdapterInfoImagen*, el cual se encarga de visualizar en la barra de estado la información del píxel de la imagen sobre el que se sitúa el puntero del ratón.

Además, en la barra de estado se muestra el factor de zoom de la imagen actualmente activa en la aplicación. Cuando el factor de zoom de la imagen cambia, el valor mostrado en la barra de estado también cambia. Para seguir el rastro del zoom de la imagen actual, se define la clase *singleton* *ControladorInfoZoomImagen*. Esta clase implementa el patrón *Observer*, y observa al *ModeloImagen* actualmente activo (asignado por el *EditorImagen* activo). Cuando cambia el factor de zoom de la imagen, el *ControladorInfoZoomImagen* es notificado, y visualiza el nuevo factor de zoom en la barra de estado a través del *ControladorInfoBarraEstado*.

El diagrama de clases de la figura 6.30 muestra las clases que participan en el proceso de mostrar y actualizar la información de la barra de estado.

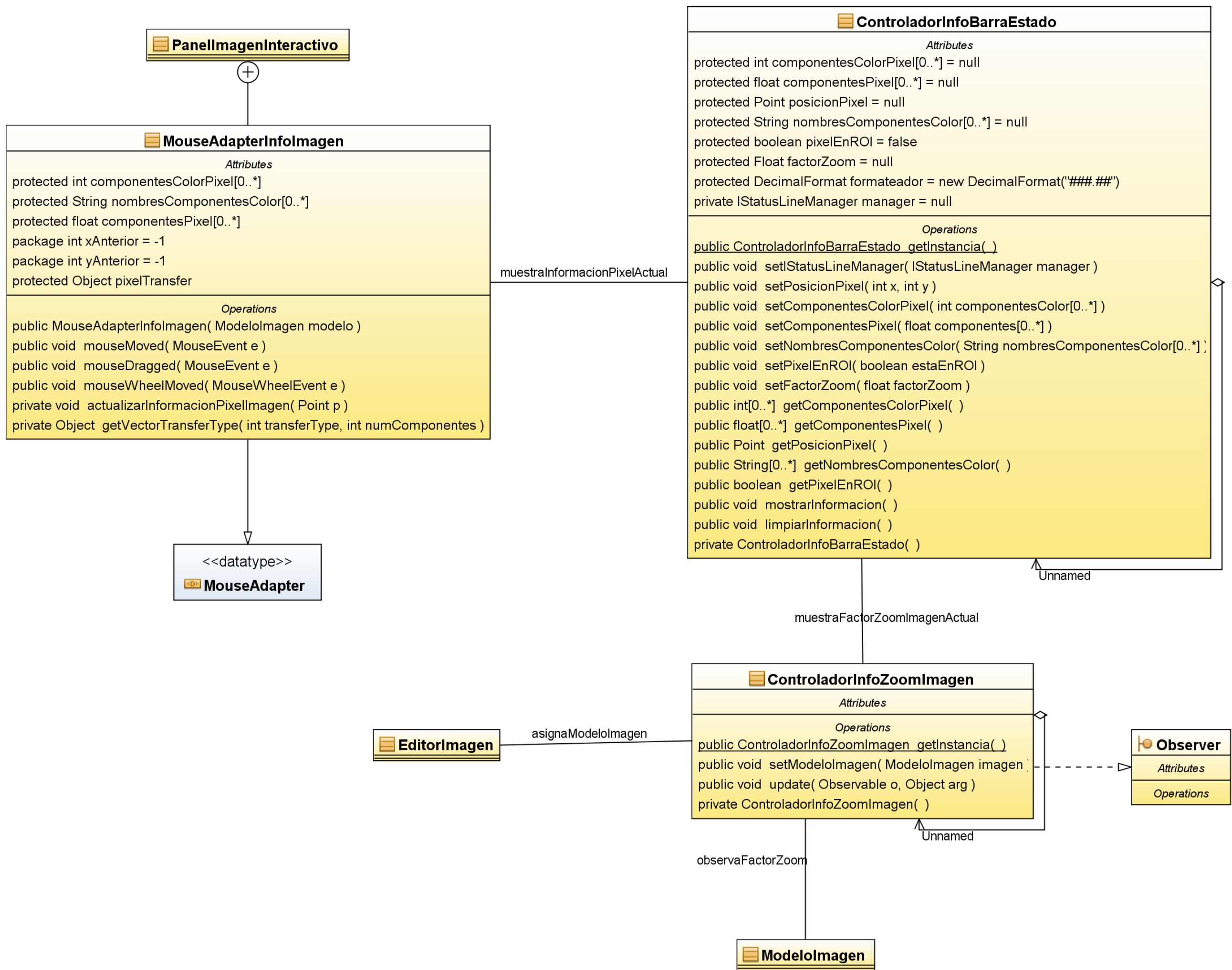

Figura 6.30: Diagrama de clases de la gestión de la barra de estado

Capítulo 7

Pruebas de rendimiento

Cool Imaging es una aplicación enfocada principalmente al procesamiento de lotes de imágenes. Aunque permite trabajar con imágenes individuales de forma interactiva, su gran potencia radica en la posibilidad de tratar de forma conjunta paquetes de cientos o miles de imágenes.

En el ámbito de la caracterización tiene especial importancia la posibilidad de tratar grandes repertorios de imágenes. La caracterización de imágenes es especialmente útil cuando los resultados de caracterización se calculan sobre una gran muestra de imágenes, ya que de este modo se obtienen resultados estadísticamente significativos.

El rendimiento de la aplicación cuando se procesan grandes lotes de imágenes es un aspecto de vital importancia. No es sólo una tarea que vaya a ser realizada con gran frecuencia, sino que, además, es una tarea que implica un gran consumo de recursos, tanto de memoria como de tiempo de cómputo.

Se espera consumir una gran cantidad de memoria, dado que se van a procesar cientos de imágenes. En este sentido las pruebas servirán, además, para comprobar la existencia de fugas de memoria. Una pequeña fuga de memoria durante el procesamiento de lotes de imágenes puede ser fatal, ya que el tamaño de la fuga puede verse incrementado de forma alarmante a causa del proceso iterativo de tratamiento de cientos de imágenes. Una pequeña fuga, en definitiva, puede llegar a colapsar el sistema.

Por otro lado, se espera analizar el tiempo consumido durante el procesamiento de los lotes de imágenes. El procesamiento de imágenes es una tarea, en general, computacionalmente cara, ya que requiere el procesamiento de cientos de megabytes de información. Si este procesamiento se aplica a cientos de imágenes, el problema del tiempo de cómputo puede volverse realmente grave.

Gracias a la biblioteca JAI y a sus optimizaciones para ciertas plataformas, los tiempos de ejecución no deberían ser elevados.

7.1. Paquetes de prueba

Se propone el procesamiento de tres paquetes de imágenes de prueba.

- Knots: contiene 438 imágenes de nudos en piezas de madera. Las imágenes son pequeñas, de 57x63, y a color. Este paquete de imágenes ha sido extraído de la página web http://www.ee.oulu.fi/ olli/Projects/Lumber.Grading.html.

- Aerials: contiene 37 imágenes de fotos aéreas a gran altura, todas ellas en color. 12 de ellas tienen un tamaño de 512x512, 24 tienen un tamaño de 1024x1024, y una tienen un tamaño de 2250x2250. Este paquete de imágenes ha sido extraído de la página web http://sipi.usc.edu/database/.

- Uva: contiene 312 imágenes de objetos varios con diferentes iluminaciones. Son imágenes a color, y su tamaño es de 384x288. Este paquete de imágenes ha sido extraído de la página web http://staff.science.uva.nl/ aloi/.

El procesamiento de estos paquetes tendrá una doble vertiente, una de tratamiento y otra de caracterización de imágenes.

La vertiente de tratamiento consistirá en aplicar cadenas de operaciones de tratamiento de imágenes a cada uno de los paquetes seleccionados.

En particular, se aplicarán las siguientes cadenas de operaciones:

- Cadena *preprocesamientoKnots*: esta cadena de operaciones escala la imagen (triplicando su alto y su ancho), aplica un filtrado gaussiano (de media 0 y desviación típica 1), convierte la imagen al espacio de color HSI, y finalmente extrae la banda de intensidad. Esta cadena será aplicada al paquete *Knots*.

- Cadena *preprocesamientoAerials*: esta cadena de operaciones convierte la imagen al espacio de color HSI, le aplica un filtrado de mediana (de tamaño 3, forma cuadrada), y finalmente la convierte de nuevo al espacio de color RGB. Esta cadena será aplicada al paquete *Aerials*.

- Cadena *preprocesamientoUva*: esta cadena de operaciones convierte la imagen al espacio de color HSI, extrae la banda de intensidad, le suma una constante (25), y finalmente le aplica un filtrado gaussiano (de media 0 y desviación típica 1). Esta cadena será aplicada al paquete *Uva*.

La vertiente de caracterización consistirá en obtener vectores de caracterización de las imágenes a través de los siguientes generadores de vector de caracterización:

- Generador *caracterizacionKnots*: construye un vector de caracterización que contiene la media sobre el histograma así como la desviación típica sobre el histograma.

- Generador *caracterizacionAerials*: construye un vector de caracterización que contiene la uniformidad sobre la matriz de co-ocurrencia así como la entropía sobre la matriz de co-ocurrencia.

- Generador *caracterizacionUva*: construye un vector de caracterización que contiene el umbral iterativo sobre el histograma, el umbral por el método de la moda sobre el histograma, y el contraste sobre la matriz de co-ocurrencia.

En todos los casos, el histograma y la matriz de co-ocurrencia creados tienen 64 intervalos, y son construidos considerando todos los píxeles de la imagen (es decir, no se hace un submuestreo para reducir el tiempo de cómputo).

Para las pruebas se han usado tres equipos informáticos: dos ordenadores portátiles y un PC de sobremesa, con las siguientes características:

Componente	PC sobremesa	Portátil 1	Portátil 2
Procesador	Pentium 4 3,2 GHz	Intel Core 2 Duo 2,26 GHz	Intel Pentium 1,73GHz
Ram	1024 MB	4096 MB	1024 MB

Tabla 7.1: Características de los ordenadores de pruebas

Para la medición de la memoria usada por cada ejecución, se ha hecho uso de la clase *Runtime* de Java. Esta clase proporciona métodos de utilidad que permiten medir la memoria usada en una aplicación en cada momento. Concretamente, permite:

- Medir cuánta memoria tiene actualmente reservada la máquina virtual. Esta memoria puede estar libre o no.
- Medir cuánta memoria hay libre actualmente en la máquina virtual.
- Medir cuál es la máxima cantidad de memoria que podrá reservar la máquina virtual. Éste suele ser un valor estático, que nosotros hemos fijado a 1 GB.

Se han llevado a cabo ejecuciones en dos plataformas distintas: linux y win32. En particular, se ha testeado en dos distribuciones de Windows, Windows XP y Windows Vista, y en una distribución de Linux, Ubuntu 9.04.

En la tabla 7.2 se resumen las ejecuciones en el Portátil 1, bajo Windows Vista. La tabla 7.3 resume las ejecuciones del Portátil 1 bajo Ubuntu 9.04. La tabla 7.4 recoge las ejecuciones del Portátil 2, bajo Windows XP. La tabla 7.5 recoge las ejecuciones del Portátil 2, bajo Ubuntu 9.04.

Los resultados ofrecidos por el ordenador de sobremesa han sido omitidos por ser especialmente similares a los del Portátil 1. Es interesante fijarse en cómo el uso de CPU del Portátil 1 llega sólo al 50 % aproximadamente, mientras que el uso de CPU del Portátil 2 llega casi al 100 %. Esto se debe al carácter multinúcleo del Portátil 1. Cuando Cool Imaging lanza la tarea de procesamiento de paquetes de imágenes, la asigna a una hebra independiente, de modo que en el Portátil 1, al tener dos núcleos, sólo se aprovecha el 50 % de capacidad de procesamiento de la CPU. En el Portátil 2, dicha hebra ocupa la única CPU existente, motivo por el cual el porcentaje de uso de CPU asciende al 100 % aproximadamente.

En lo que respecta al tiempo de ejecución, se puede apreciar que Cool Imaging muestra un buen comportamiento. A pesar de procesar un gran número de imágenes (de tamaños no muy grandes ciertamente), el tiempo de procesamiento es en general bastante pequeño, no sobrepasando los minutos. Por supuesto, en caso de procesar imágenes más grandes, el tiempo de ejecución crecería sensiblemente, pero entraría aun así dentro de unos márgenes razonables. El tratamiento y caracterización de imágenes no exige tiempos de ejecución especialmente bajos, por lo que se puede concluir que Cool Imaging responde positivamente a los requerimientos de tiempo establecidos.

Cabe notar el buen comportamiento de Cool Imaging en la gestión de memoria. Tanto en Windows Vista con en Ubuntu 9.04 generalmente se mantiene casi toda la memoria de la máquina virtual libre, como se desprende de las tablas de resultados.

Es de especial mención que el rendimiento en memoria ha mostrado un mejor comportamiento en Windows Vista y Windows XP que en Ubuntu 9.04. Este problema, sin embargo, no se debe a un fallo de diseño de Cool Imaging, sino a la gestión de memoria que realiza la máquina virtual en cada uno de los sistemas operativos. La memoria total usada por la máquina virtual no puede ser gestionada por la aplicación, así que es una variable por la que no deberíamos preocuparnos. Lo que realmente merece la pena es ver que la cantidad de memoria libre en relación al total de memoria ocupada, algo que sí depende de Cool Imaging, es equivalente en ambos sistemas operativos, motivo por el cual podemos concluir que la gestión de la memoria es eficiente y, posiblemente, libre de fugas de memoria.

Portátil 1, Windows Vista						
		Tiempo	Uso CPU	Memoria libre (promedio)	Memoria total (promedio)
Cadenas de operaciones	preprocesamientoKnots	1 min 10,17 s	48 %	31 MB	50 MB
	preprocesamientoAerials	26,97 s	47 %	81 MB	125 MB
	preprocesamientoUva	1 min 29,56 s	47 %	30 MB	51 MB
Generadores de VC	caracterizacionKnots	1 min 52,2 s	52 %	52 MB	74 MB
	caracterizacionAerials	17,56 s	55 %	43 MB	75 MB
	caracterizacionUva	1 min 59,48 s	52 %	32 MB	56 MB

Tabla 7.2: Pruebas de rendimiento en el Portátil 1 bajo Windows Vista.

Portátil 1, Ubuntu 9.04					
		Tiempo	Uso CPU	Memoria libre (promedio)	Memoria total (promedio)
Cadenas de operaciones	preprocesamientoKnots	1 min 10,27 s	48 %	244 MB	265 MB
	preprocesamientoAerials	19,21 s	47 %	265 MB	313 MB
	preprocesamientoUva	57,13 s	47 %	11 MB	35 MB
Generadores de VC	caracterizacionKnots	1 min 44,36 s	56 %	165 MB	198 MB
	caracterizacionAerials	18,1 s	50 %	180 MB	210 MB
	caracterizacionUva	1 min 55,89 s	55 %	168 MB	204 MB

Tabla 7.3: Pruebas de rendimiento en el Portátil 1 bajo Ubuntu 9.04.

Portátil 2, Windows XP					
		Tiempo	Uso CPU	Memoria libre (promedio)	Memoria total (promedio)
Cadenas de operaciones	preprocesamientoKnots	2 min 2,4 s	98 %	33 MB	56 MB
	preprocesamientoAerials	55,55 s	99 %	79 MB	121 MB
	preprocesamientoUva	1 min 53,64 s	97 %	34 MB	59 MB
Generadores de VC	caracterizacionKnots	3 min 20,2 s	99 %	55 MB	77 MB
	caracterizacionAerials	26,66 s	95 %	40 MB	70 MB
	caracterizacionUva	2 min 43,17 s	96 %	35 MB	66 MB

Tabla 7.4: Pruebas de rendimiento en el Portátil 2 bajo Windows XP.

Portátil 2, Ubuntu 9.04					
		Tiempo	Uso CPU	Memoria libre (promedio)	Memoria total (promedio)
Cadenas de operaciones	preprocesamientoKnots	2 min 5,95 s	95 %	235 MB	257 MB
	preprocesamientoAerials	34,05 s	99 %	264 MB	319 MB
	preprocesamientoUva	1 min 43 s	99 %	224 MB	265 MB
Generadores de VC	caracterizacionKnots	2 min 59,45 s	94 %	176 MB	205 MB
	caracterizacionAerials	32,16 s	97 %	189 MB	214 MB
	caracterizacionUva	2 min 43,22 s	99 %	182 MB	224 MB

Tabla 7.5: Pruebas de rendimiento en el Portátil 2 bajo Ubuntu 9.04.

Capítulo 8

Conclusiones

El campo de la caracterización de imágenes está en continuo avance. Con el desarrollo de Cool Imaging, nuestro proyecto fin de carrera, hemos contribuido al desarrollo de este área técnico-científica.

Inicialmente nos propusimos un objetivo principal: desarrollar un sistema de caracterización de imágenes digitales, que pudiera realizar estudios tanto locales como globales, y que estuviera enfocado a la obtención de resultados.

Sin duda, Cool Imaging cumple con este objetivo. Cool Imaging permite **caracterizar** tanto imágenes individuales como **lotes** completos de imágenes. Los resultados de la caracterización son mostrados al usuario de forma compacta e intuitiva a través de *informes de caracterización*, que puede almacenar para su posterior uso. Estos resultados, además, pueden ser **exportados** a ficheros en diversos formatos estadísticos de aceptación universal, como son:

- Hoja de cálculo ODF (.ods).
- Microsoft Excel 97/2000/XP (.xls).
- Microsoft Excel 2007 XML (.xlsx).
- Fichero de texto plano (.dat).

Con ello, el usuario podrá analizar y manipular los resultados de la caracterización de forma profesional a través de paquetes estadísticos como SPSS[1] o R-Project.

El siguiente objetivo que guió el desarrollo de Cool Imaging fue el de su **extensibilidad**. Desde un inicio se fue consciente de que el campo de la caracterización de imágenes, en continuo avance, obligaba a construir una aplicación que permitiera la incorporación de nuevos algoritmos de caracterización y procesamiento de imágenes.

Nuevamente se puede decir que Cool Imaging ha cumplido su objetivo. Gracias a la arquitectura subyacente usada, el *Eclipse Rich Client Platform*, la posibilidad de extender la funcionalidad de la aplicación mediante *plugins* es un hecho. Con unos mínimos conocimientos de la arquitectura Eclipse RCP, se puede añadir a Cool Imaging nuevos algoritmos de caracterización y tratamiento de imágenes. Es más, los repertorios de operaciones que inicialmente se incluyen en la aplicación están implementados mediante *plugins* externos a la aplicación principal, siguiendo la misma filosofía que otros desarrolladores pueden utilizar para extender la funcionalidad del sistema.

La extensibilidad de Cool Imaging va más allá, ya que permite, no sólo añadir nuevos algoritmos de tratamiento y caracterización de imágenes, sino añadir nuevos módulos de funcionalidad inicialmente no contemplados.

[1] *Statistical Package for the Social Sciences.*

La **facilidad de uso** se planteó como objetivo desde el inicio del proyecto, pensando en que los potenciales usuarios de Cool Imaging podrían provenir de muy distintas áreas.

Gracias al uso del *framework* del Eclipse RCP, la aplicación no sólo muestra una apariencia profesional, sino que además es fácilmente navegable y altamente reconfigurable, permitiendo así que el usuario la adapte a sus necesidades.

Pensando en el posible uso de la aplicación por parte de la comunidad de habla no hispana, Cool Imaging ha sido diseñada para ser fácilmente traducible a otros idiomas, sin necesidad de recompilar la aplicación o acceder a su código fuente. Basta que se proporcione los ficheros de texto con la traducción al idioma deseado, y Cool Imaging la incorporará de forma automática.

Por último, y no por ello menos importante, el software desarrollado es **multiplataforma**. Cool Imaging está disponible para las plataformas win32, linux y macosx.

A lo largo de esta memoria se ha descrito el proceso de desarrollo de Cool Imaging, que ha estado guiado por una metodología ágil de desarrollo de software. La experiencia ha demostrado que los métodos clásicos son insuficientes en el desarrollo de software, y que los métodos ágiles tienen grandes ventajas sobre los primeros.

Partiendo de esta base, se definió una lista de tareas o *product backlog*, que se convirtió en los requerimientos funcionales del sistema.

De forma iterativa se resolvieron los requerimientos del listado de tareas. En cada iteración se obtenía un software completo y testeable, lo cual permitía percibir una clara evolución del sistema. Durante el desarrollo de cada iteración se recorrían las fases básicas del proceso del software: modelado de requisitos mediante casos de uso, análisis del problema, diseño orientado a Java e implementación. Una de las grandes ventajas de obtener un software testeable en cada iteración es que las fase de pruebas se pudo relajar y distribuir temporalmente a lo largo de todo el desarrollo, evitando así aplicar ingentes baterías de pruebas a módulos complejos y no testeados previamente.

Por último, se han incluido varios tests de rendimiento de Cool Imaging, con la idea de medir de forma cuantitativa su potencia, en términos de tiempo y espacio consumidos durante el procesamiento de grandes lotes de imágenes. Se ha podido comprobar que Cool Imaging hace una buena gestión de los recursos computacionales, pues ofrece unos tiempos de ejecución competitivos y además gestiona de forma eficiente la memoria.

Acerca del proceso de desarrollo

Tal y como se explicó en la sección 2.4, el proceso de desarrollo del software ha seguido una metodología basada en Scrum, pero algo más laxa.

A lo largo del proyecto se ha mantenido un listado de tareas globales, o *product backlog*. Este listado de tareas aglutina tanto los requisitos funcionales como los no funcionales de la aplicación, así como otro tipo de tareas que no caen dentro de estas dos categorías.

En la figura 8.1 se muestra una gráfica que representa el número de tareas sin resolver del *product backlog* frente al número de semana de desarrollo (periodo que se extiende desde la semana 40 de 2008 hasta la semana 27 de 2009).

Analizando la gráfica se pueden extraer varias conclusiones que explican, de forma objetiva, cómo ha procedido el desarrollo del proyecto.

Durante las primeras semanas de desarrollo se llevó a cabo, principalmente, el proceso de aprendizaje de las arquitecturas Eclipse RCP así como de la biblioteca de procesamiento de imágenes JAI. Durante estas semanas, por tanto, el número de tareas globales siempre se mantuvo bajo, sin una tendencia claramente decreciente, que es lo que cabe de esperar durante el desarrollo de software.

A partir de la semana 48 de 2008, sin embargo, se observa un aumento importante en el número de tareas. Esto se debió, fundamentalmente, a la concreción de los requisitos funcionales

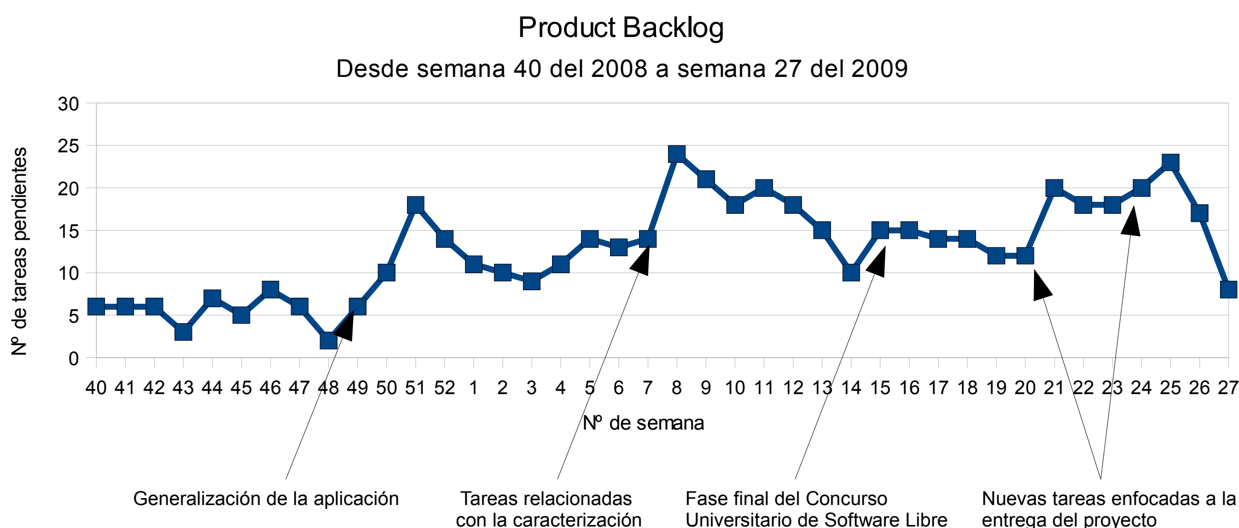

Figura 8.1: *Product Backlog* (número de tareas) del proceso de desarrollo

por parte del usuario. Si bien hasta entonces se había tenido una idea general acerca de lo que se esperaba de Cool Imaging, fue a partir de esa semana que dicha idea se plasmó en requisitos funcionales tangibles, motivo por el cual se produjo un fuerte incremento del número de tareas.

Es interesante observar una tendencia periódica en el ascenso y descenso del número de tareas. A partir del pico de la semana 51 de 2008, el número de tareas decrece de forma progresiva hasta entrar en otra fase de crecimiento de tareas, que culminó en la semana 8 de 2009. Alrededor de esa semana se especificaron los requerimientos funcionales relacionados con la caracterización de imágenes, motivo por el cual el número de tareas volvió a incrementarse en gran medida. El número de tareas vuelve a decrecer de forma progresiva hasta la semana 14, momento en el que, por motivo de la presentación de este proyecto en el Concurso Universitario de Software Libre, se decidió dar un pequeño empujón extra, a raíz del cual se incrementó ligeramente el número de tareas total. De nuevo, se entró en una fase decreciente, que se vio alterada por la proximidad de la entrega del proyecto. A partir de entonces, nuevamente, se produjo una tendencia decreciente.

Esta tendencia periódica ascendente y descendente tiene sentido dentro del proceso del software, ya que las tareas pendientes se van resolviendo, decreciendo así en número, pero nuevas especificaciones por parte del cliente suponen un aumento del número total de tareas sin resolver. Sin embargo, el bajo número de desarrolladores implicados, sólo 2, ha provocado que la media del número de tareas sin resolver a lo largo de todo el proyecto haya sido relativamente alto, y que en general no haya decrecido con la suficiente rapidez. En el futuro, por tanto, sería necesario reconsiderar el incorporar colaboradores más activos que se dedicaran al desarrollo directo de Cool Imaging.

Bibliografía

[1] Librería OpenCV: http://sourceforge.net/projects/opencvlibrary.

[2] Librería Java Advanced Imaging: http://java.sun.com/javase/technologies/desktop/media/jai.

[3] Programming in Java Advanced Imaging: http://java.sun.com/products/java-media/jai/forDevelopers/jai1_0_1guide-unc .

[4] Java Media APIs: Cross-Platform Imaging, Media, and Visualization by Alejandro Terrazas, John Ostuni, Michael Barlow .

[5] Programmer's Guide to the Java 2D API, Sun Microsystems.

[6] Eclipse Rich Client Platform: http://wiki.eclipse.org/index.php/Rich_Client_Platform.

[7] Professional Eclipse 3 for Java Developers, Berthold Daum.

[8] Eclipse: Building Commercial-Quality Plug-ins, Second Edition by Eric Clayberg & Dan Rubel.

[9] The Definitive Guide to SWT and JFace by Robert Harris and Rob Warner.