INGENIERÍA INFORMÁTICA
Curso Académico 2007/2008

Proyecto de Fin de Carrera

RZ-LAYOUT: Generalización de una técnica de visualización

Autor: Francisco Javier Almeida Martínez
Tutor: Jaime Urquiza Fuentes
Índice general

Índice general

Índice de figuras

Resumen

1. Introducción 1

2. Objetivos 5

3. Cuestiones de diseño 7
   3.1. Elección del lenguaje de programación 7
   3.2. Elementos gráficos en Java 8
   3.3. Elección del elemento responsable de la visualización 9
   3.4. Características de los layouts de Java 11
   3.5. Características del layout que se necesita 13
   3.6. Características del componente que se necesita 14
ÍNDICE GENERAL

3.7. Características de la API ........................................... 15
  3.7.1. Distancia inicial de los elementos en el espacio de trabajo ..... 15
  3.7.2. Cambio de tamaño del espacio de trabajo ....................... 16
  3.7.3. Selección de una miniatura ................................... 18

4. Descripción informática ........................................... 21
  4.1. Casos de uso de RZ-Layout ...................................... 22
  4.2. Desarrollo de un nuevo layout en Java:
        RZ-Layout ....................................................... 23
  4.3. Desarrollo del nuevo tipo de componente:
        RZ-Component .................................................. 24
  4.4. Especificación detallada de RZ-Layout ............................ 26
    4.4.1. Crear un nuevo RZ-Layout .................................. 29
    4.4.2. Encapsulación de contenedores: barras de scroll .............. 29
    4.4.3. Nuevo tipo de componente: RZ-ComponentInformado .......... 30
    4.4.4. Transparencia de uso de RZ-Layout .......................... 31
    4.4.5. Distribución total de componentes ............................ 33
    4.4.6. Resize: recolocación de componentes .......................... 34
    4.4.7. Averiguar el componente seleccionado ........................ 40
    4.4.8. Recolocación R-Zoom ......................................... 42
    4.4.9. Eventos de teclado .......................................... 49
ÍNDICE GENERAL

4.5. Jerarquía de clases ................................................. 53

4.6. Generalización y parametrización de
RZ-Layout ............................................................... 56

  4.6.1. Recolocación opcional 1: RZ-Centrado ................. 59

  4.6.2. Recolocación opcional 2: Centrar-Fila ............... 61

  4.6.3. Recolocación opcional 3: vista Global + Detalle .... 62

4.7. Problemas durante la implementación .................... 66

  4.7.1. Identificación de los elementos ....................... 66

  4.7.2. Conversión de Component a RZ-ComponentInformado .... 68

  4.7.3. Señalar el elemento seleccionado ..................... 68

4.8. Pruebas y evaluación de RZ-Layout ..................... 70

  4.8.1. Ejemplo de utilización 1: visualización simple de imágenes ... 70

  4.8.2. Ejemplo de utilización 2: visualización de imágenes .... 74

  4.8.3. Ejemplo de utilización 3: visualización de páginas Web .... 76

5. Conclusiones y trabajos futuros .............................. 79

Glosario ................................................................. 82

Bibliografía .......................................................... 83
Índice de figuras

<table>
<thead>
<tr>
<th>Sección</th>
<th>Título</th>
<th>Página</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Esquema de carga de RZ-Layout</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Cambio de tamaño del espacio de trabajo. A la izquierda se puede ver el estado anterior a la ejecución del método resize. A la derecha está el estado después de la ejecución del método resize.</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Esquema del caso básico. La figura de la izquierda hace referencia al estado anterior, en el que no hay ningún elemento seleccionado. La figura de la derecha muestra que se ha asignado el foco al tercer elemento de la primera fila</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>Esquema del caso alternativo I. A la izquierda tenemos el esquema inicial antes de asignar el foco. La imagen de la derecha muestra los cambios que se han hecho para asignar el foco</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Casos de uso de RZ-Layout</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Especificación de la interfaz LayoutManager</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>Especificación de la interfaz RZ-Component</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>Diagrama de la clase RZ-Layout</td>
<td>27</td>
</tr>
<tr>
<td>4.5</td>
<td>Encapsulación de contenedores del RZ-Layout</td>
<td>30</td>
</tr>
<tr>
<td>Número</td>
<td>Descripción</td>
<td>Página</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>4.6.</td>
<td>Diagrama de colaboración de asignación de evento de resize</td>
<td>35</td>
</tr>
<tr>
<td>4.7.</td>
<td>Diagrama de colaboración de registros de eventos de resize</td>
<td>36</td>
</tr>
<tr>
<td>4.8.</td>
<td>Diagrama de colaboración para la optimización de resize</td>
<td>39</td>
</tr>
<tr>
<td>4.9.</td>
<td>Diagrama de colaboración para la selección de componentes</td>
<td>42</td>
</tr>
<tr>
<td>4.10.</td>
<td>Imagen de la carga inicial del RZ-Layout</td>
<td>44</td>
</tr>
<tr>
<td>4.11.</td>
<td>Imagen del caso básico con foco</td>
<td>46</td>
</tr>
<tr>
<td>4.12.</td>
<td>Esquema del cálculo de la nueva coordenada Y</td>
<td>48</td>
</tr>
<tr>
<td>4.13.</td>
<td>Imagen del caso alternativo I con foco</td>
<td>49</td>
</tr>
<tr>
<td>4.14.</td>
<td>Esquema R-Zoom para el caso del keyUp: El elemento seleccionado es el 5, el de</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>la misma fila es el 2, pero el más cercano visualmente es el 1.</td>
<td></td>
</tr>
<tr>
<td>4.15.</td>
<td>Diagrama de integración del RZ-Layout en la jerarquía de los layouts de Java</td>
<td>53</td>
</tr>
<tr>
<td>4.16.</td>
<td>Diagrama de clases que implementan el RZ-Layout</td>
<td>54</td>
</tr>
<tr>
<td>4.17.</td>
<td>Diagrama de clases de utilización del RZ-Layout</td>
<td>55</td>
</tr>
<tr>
<td>4.18.</td>
<td>Esquema de recolocación de los distintos métodos</td>
<td>57</td>
</tr>
<tr>
<td>4.19.</td>
<td>Funcionamiento RZ-Centrado. Como se puede ver se ha seleccionado el segundo</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>elemento de la primera fila, lo que ha provocado que se divida la fila en dos</td>
<td></td>
</tr>
<tr>
<td></td>
<td>y se sitúe por un lado el elemento que tiene el foco en el centro del</td>
<td></td>
</tr>
<tr>
<td></td>
<td>contenedor y el resto de elementos a su derecha han conservado su posición en</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X pero se han situado debajo del elemento seleccionado</td>
<td></td>
</tr>
</tbody>
</table>
ÍNDICE DE FIGURAS

4.20. En la figura de la izquierda se puede ver la carga inicial, en la que no se ha seleccionado ningún elemento. A la derecha vemos cómo se ha seleccionado el tercer elemento de la primera fila. Como resultado, se marca dicho elemento y se muestra su vista detallada . . . . . . . 62

4.21. Se encuentra seleccionado el tercer elemento de la segunda fila . . 63

4.22. Visualización vista Global + Detalla. En la figura de la izquierda se muestra la carga inicial. Se puede observar que el ScrollPane se encuentra desplazado en altura. En la figura de la derecha se ha seleccionado el quinto elemento de la segunda fila . . . . . . . . . 66

4.23. Unificación de Ejemplos . . . . . . . . . . . . . . . . . . . . . . . . . . 70

4.24. Visualización simple de imágenes . . . . . . . . . . . . . . . . . . . . . . . 73

4.25. Esquema visualización compleja de imágenes . . . . . . . . . . . . . . 74

4.26. Visualización de páginas Web . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
Agradecimientos

No existen palabras suficientes para agradecer el apoyo que durante estos cin-co años me han ofrecido todos los míos. En especial quiero dar las gracias a mis padres por el esfuerzo que han tenido que realizar para que yo pudiera terminar mis estudios en Madrid, apoyándome siempre en todas mis decisiones. Quiero dar las gracias también a mi novia por haberme comprendido, animado y apoyado en todo momento durante todo este tiempo. Agradecer también de forma especial a mi tutor de proyecto Jaime Urquiza Fuentes, por la ayuda prestada durante todo el proyecto y por guiarme como lo ha hecho.

Dedicado a: César Martínez Hernández
Resumen

La visualización de información se puede definir como el uso de medios informáticos, interactivos y visuales para la representación de datos abstractos, de forma que facilite el proceso cognitivo [14]. El objetivo de la visualización de información es convertir los datos a una representación visual para mejorar la percepción humana. Dentro de la visualización de información, nos centraremos en la manipulación de espacios de trabajo a gran escala. Este PFC se centra en R-Zoom, una técnica Foco+Contexto que actualmente se encuentra implementada en Delphi e integrada específicamente en un entorno de programación.

El objetivo de este PFC es realizar una API genérica que facilite el uso de R-Zoom, y que permita la visualización de cualquier tipo de componente. Para la nueva implementación se utilizará Java, ya que es uno de los lenguajes de programación más extendidos en la actualidad y posee una estructura interna que facilita la reutilización. Además, se desarrollará un tipo de componente genérico que permitirá visualizar cualquier tipo de elemento. Una vez que se tiene implementado R-Zoom, se plantea la generalización de la API para permitir otros tipos de visualizaciones diferentes.

Por último, se realizarán unas implementaciones utilizando la API desarrollada, que servirán para probarla y evaluar su facilidad de uso desde el punto de vista del programador.
Capítulo 1

Introducción


El crecimiento de la cantidad de información, complica enormemente que una persona pueda extraer conclusiones, tendencias y patrones a partir de los datos. En este contexto es donde tiene sentido hablar de visualización de información. El principal objetivo de la visualización de información es la representación perceptual adecuada de datos [14]. Su propósito no es la creación de las imágenes en sí mismas, sino mejorar la asimilación rápida de información. La cantidad de aplicaciones en visualización de información es grande y crece constantemente siendo actualmente un área de activo desarrollo. También cabe señalar que un requisito fundamental de la visualización de información, es que proporcione una respuesta en tiempos interactivos. Los métodos utilizados en las distintas ramas de la visualización son, en su mayoría, de gran costo computacional y es por ello que, para lograr una visualización en tiempos interactivos, es necesario contar con métodos adecuados para los distintos conjuntos de datos [17].
Dentro de la visualización de la información, nos centraremos en las técnicas de manipulación de espacios de trabajo a gran escala. Estas técnicas son: Zoom+Pan, Vista Global+Vista Detallada y Foco+Contexto.

**Técnicas de Zoom+Pan.** Este tipo de técnicas organizan la información distribuyéndola en el espacio de trabajo, permitiendo al usuario manipularla interactuando directamente con ella. Como el espacio de trabajo es más grande que la pantalla, el usuario sólo puede acceder a una parte de él (zona visible) en cada momento, sin embargo, podrá desplazarse por el espacio de trabajo (panning), cambiando la zona visible, así como acercarse o alejarse (zooming) [7].

**Técnicas de Vista Global+Vista Detallada.** Estas técnicas se caracterizan por dividir el espacio de trabajo en dos zonas bien diferenciadas: la zona de vista global y la zona de vista detallada. La vista global muestra al usuario una visión general de la información con la que debe trabajar, la vista detallada muestra, con mayor grado de detalle, una parte de la información previamente seleccionada en la vista global. Así, ambas vistas están sincronizadas en cuanto a contenidos [13].

**Técnicas de Foco+Contexto.** Al igual que las técnicas de Vista Global+Detalle, estas técnicas muestran simultáneamente vistas globales y detalladas en el espacio de trabajo. La diferencia radica en que las técnicas Foco+Contexto integran en una sola vista todo. Así, el espacio de trabajo es ocupado simultáneamente por los elementos de interés (foco) y por el resto de los elementos (contexto) [13].

R-Zoom [16] es una técnica de la familia Foco+Contexto, que actualmente se encuentra integrada en un entorno de programación funcional. En esta técnica los elementos tienen dos aspectos: vista detallada y vista en miniatura. Normalmente los elementos muestran su vista en miniatura, pero al seleccionar uno, éste (foco) cambia su aspecto y muestra su vista detallada. Al producirse este cambio, como el espacio de trabajo sigue siendo el mismo, es necesario recolocar las miniaturas para que sean visibles. La forma de redistribuir los elementos dependerá del foco y del tamaño del espacio de trabajo (véase 4.4.8).
1. *Introducción*

El resto de la memoria se estructura como sigue. En el capítulo 2 se hace una exposición de los objetivos de este PFC. En el capítulo 3 se realiza un análisis de los aspectos del diseño y cuestiones previas. La existencia de este capítulo se debe a que se necesitan tomar decisiones previas a la implementación, que condicionarán todo el desarrollo. En el capítulo 4 se realiza la descripción informática detallada. Y finalmente, en el capítulo 5 se exponen las conclusiones obtenidas del desarrollo del PFC.
Capítulo 2

Objetivos

R-Zoom es una técnica de visualización del tipo Foco+Contexto, que se caracteriza por mantener un nivel de comprensión de los elementos del contexto y minimizar los cambios en su localización. Al comienzo de este trabajo R-Zoom estaba implementado en Delphi e integrado en un entorno de programación+animación llamado WinHIPE [12].

El objetivo de este PFC es extraer el funcionamiento de R-Zoom de la aplicación en la que se encuentra e implementar una API genérica. Esta API debe permitir utilizar R-Zoom en cualquier contexto. Además, el hecho de ser genérica quiere decir que se debe permitir utilizar la API con cualquier tipo de componente. También será necesario extender la API para ofrecer otras técnicas de visualización de información diferentes a R-Zoom.

Finalmente evaluaremos la facilidad de uso de la API desarrollada, creando una serie de aplicaciones. Para esta tarea será necesario ponernos en el papel del programador e ignorar la implementación interna de la API.
Capítulo 3

Cuestiones de diseño

En este capítulo vamos a analizar algunos aspectos, que aunque no son requisitos, van a condicionar completamente el desarrollo de la API. Entre ellos, está la elección del lenguaje de programación para el desarrollo del PFC. Luego se realizará una descripción detallada del funcionamiento que debe tener la API que queremos desarrollar.

3.1. Elección del lenguaje de programación

Hoy por hoy existe un gran abanico de lenguajes de programación en el que cada uno tiene unas características que lo hace ideal para una determinada tarea. Java, es uno de los lenguajes de programación más utilizados en la actualidad, debido principalmente a la independencia de plataforma que proporciona, lo que se transforma en una alta portabilidad. Sin embargo, desde el punto de vista del rendimiento, existen determinadas características del lenguaje que hacen que las aplicaciones resultantes no sean tan veloces como las creadas con otros lenguajes. Existen datos comparativos en los que se muestra que Java puede llegar a ser entre 2 y 5 veces más lento que C++ [1].
3.2. Elementos gráficos en Java

El motivo de la disminución de rendimiento de Java se debe a la forma en la que se consigue la gran portabilidad del lenguaje. En este punto, cabe destacar que para poder ejecutar aplicaciones creadas en Java es necesario instalar su máquina virtual, que es la que va a interpretar el lenguaje. Así, según estemos en una plataforma u otra, se instalará siempre la versión de la máquina virtual que el fabricante proporcione para dicha plataforma. El hecho de tener una máquina virtual para cada plataforma y además que cada una de ellas ejecute el mismo código, evita al programador tener que re-compilar los programas según la plataforma.

El desarrollo de este proyecto requiere de un lenguaje de programación que sea ampliamente utilizado y que tenga una alta portabilidad, por ello, la elección de Java está justificada. En conclusión, vamos a sacrificar un poco el rendimiento a cambio de permitir una mayor distribución de la API creada.

3.2. Elementos gráficos en Java

Una vez decidido el lenguaje de programación a utilizar y justificada dicha elección, hay que centrarse en los elementos que nos interesan del mismo. Los elementos gráficos, es decir, aquellos que pueden tener una visualización gráfica, se suelen denominar componentes (objetos de la clase Component). Ahora bien, existen diferentes tipos de componentes según lo que se quiera visualizar. Estos componentes, que tienen su nombre propio, por ejemplo Button, Label, ..., deben heredar de la clase Component de Java para que puedan ser visualizados. Además, Java distingue otro tipo de elemento gráfico muy importante: el contenedor. Un contenedor es un objeto que puede albergar en su interior componentes o incluso otros contenedores. Al igual que sucede con los componentes, existen también diferentes tipos de contenedores, por ejemplo, un panel es un contenedor que nos permite agrupar componentes. Lo que es imprescindible tener claro, es que debido a la herencia de Java, un contenedor es también un componente pero no viceversa, por esta razón a un contenedor se pueden añadir tanto componentes como otros contenedores.

Por otro lado, además de los contenedores y los componentes, hay que destacar
3. Cuestiones de diseño

otro objeto muy importante en cuanto a aspectos gráficos se refiere, el Layout. La utilización de los layouts que proporciona Java es quizás uno de los aspectos de la representación gráfica que más difiere con otros lenguajes. Cuando utilizamos un contenedor, sea de la clase que sea, y añadimos componentes, hay que especificar cómo queremos que se distribuyan dichos componentes dentro del contenedor. Para determinar dicha distribución se utilizan los layouts, que según su naturaleza realizarán una distribución u otra.

Por último, ya elegido el lenguaje, y destacado las principales características que nos interesan de él, nos centraremos en el desarrollo del PFC. Este trabajo requiere de todos los elementos del lenguaje mencionados anteriormente: componentes, contenedores y layouts.

3.3. Elección del elemento responsable de la visualización

Para el desarrollo del proyecto se ha realizado un proceso de investigación sobre el lenguaje Java que permita averiguar el punto de partida del desarrollo. En primer lugar se tendrá que seleccionar el objeto a implementar, para a continuación saber si se puede o no aprovechar alguno de los que proporciona el lenguaje.

El principal problema que se nos plantea en estos momentos es poder distinguir qué componente es el que asumirá toda la responsabilidad de visualización, o bien si ésta debe recaer sobre varios elementos. En la visualización debemos distinguir dos aspectos:

1. Aquel que se encarga de situar los elementos cargados en el sistema correctamente, es decir, todo lo que tiene que ver con su distribución en la ventana de la aplicación.

2. Aquel que se encarga de la representación de los elementos en un momento determinado, es decir, un elemento en un momento determinado puede cam-
3.3. Elección del elemento responsable de la visualización

biar su apariencia, por lo que tiene que almacenar al menos dos formas de mostrarse.

En primer lugar se estudiará la posibilidad de utilizar los componentes como elemento responsable de visualización, es decir, el componente es el responsable de decidir dónde situarse y cómo mostrarse. En cuanto a la colocación del componente, esta solución sería equivalente a no utilizar ningún layout de los que proporciona el lenguaje. Desde el punto de vista de la distribución no sería una buena solución ya que en nuestro caso interesa que al componente se le ubique automáticamente en un punto determinado. Ahora bien, desde el punto de vista de la apariencia del componente, consistiría en que el componente guardara dos aspectos de visualización y que en un momento determinado cambie de un aspecto a otro. Teniendo en cuenta estos aspectos, desde el punto de vista de la colocación del elemento, podemos descartar al componente, sin embargo, desde el otro punto de vista, parece ser que el componente es ideal para guardar los aspectos que se consideren oportunos.

Una vez elegido el componente como responsable de su propia apariencia, debemos elegir el responsable de la distribución de los propios componentes. La elección se tendrá que realizar entre layouts y contenedores. En cuanto a los contenedores, estos no tienen ninguna responsabilidad sobre la colocación de los elementos en su interior, de hecho cuando no se utiliza ningún layout, lo que se hace es colocar los elementos en los puntos que nos interese dentro del contenedor sin que éste último tenga nada que ver. Por último, ya sólo queda estudiar los layouts. Un layout no es más que un elemento que se encarga de distribuir los componentes automáticamente en el interior de un contenedor, por ello a un contenedor debe asignársele un layout para que éste haga su tarea de distribución.

En definitiva, se va a elegir al componente como responsable para cambiar de apariencia y al layout como responsable para distribuir los elementos en los contenedores. A continuación se estudiarán tanto los layouts que proporciona Java como la posibilidad de utilizar alguno de los componentes que ofrece el lenguaje.
### 3. Cuestiones de diseño

#### 3.4. Características de los layouts de Java

Los distintos layouts que ofrece Java son: FlowLayout, GridLayout, GridBagLayout, CardLayout, BorderLayout, BoxLayout, OverlayLayout, etc. A continuación se hará una breve descripción de las principales características de cada uno de ellos.

- **FlowLayout**: es un layout bastante sencillo de usar. Coloca todos los componentes que se añaden al contenedor alineados de izquierda a derecha, haciendo que cada uno ocupe lo que necesita. Si hay hueco de sobra en horizontal, los componentes aparecerán centrados. Si falta hueco, los componentes se partirán automáticamente en varias filas. Es adecuado para barras de herramientas [9].

- **GridLayout**: este layout pone los componentes en forma de matriz estirándolos para que todos tengan el mismo tamaño. Es adecuado para hacer tableros, por ejemplo calculadoras en que todos los botones son iguales.

- **GridBagLayout**: es de los layouts más versátiles y complejos de usar. Es como el GridLayout, pone los componentes en forma de matriz (cuadrícula), pero permite que las celdas y los componentes tengan tamaños variados. Es posible hacer que un componente ocupe varias celdas.

- **CardLayout**: este layout hace que los componentes recibidos ocupen el máximo espacio posible, poniendo unos encima de otros. Sólo es visible uno de los componentes, los otros quedan detrás. Tiene métodos para indicar cuál de los componentes es el que debe quedar encima y verse.

- **BorderLayout**: divide la ventana en 5 partes: centro, arriba, abajo, derecha e izquierda. Hará que los componentes que pongamos arriba y abajo ocupen el alto que necesiten, pero los estirará horizontalmente hasta ocupar toda la ventana. Los componentes de derecha e izquierda ocuparán el ancho que necesiten, pero se les estirará en vertical hasta ocupar toda la ventana. El componente central se estirará en ambos sentidos hasta ocupar toda la ventana.
3.4. Características de los layouts de Java

Es un layout adecuado para ventanas en las que hay un componente central importante (una tabla, una lista) y tiene menús o barras de herramientas situados arriba, abajo, a la derecha o a la izquierda.

- **BoxLayout**: es como un `FlowLayout` aunque un poco más completo. Nos permite decidir cómo colocar los elementos en vertical o en horizontal aunque todos los componentes van a tener las mismas dimensiones.

- **OverlayLayout**: es un layout que se incorpora en Java pero con la biblioteca Swing. El layout se encarga de hacer visible sólo el componente que tenga mayor tamaño, de tal forma, que superpone dicho componente al resto.

- **NullLayout**: se caracteriza por no utilizar ningún layout. La manera de situar los elementos es utilizando el posicionamiento absoluto. Aunque algunas fuentes lo pueden tachar de una mala solución [9], en ocasiones puede darse que nuestras necesidades no se ajusten a ninguno de los layouts proporcionados. En el caso de no utilizar ningún layout lo que se hace es indicar a cada componente el lugar exacto en el que debe situarse en el interior del contenedor, de ahí que también se conozca con el nombre de posicionamiento absoluto.

Aunque no se añadan a la lista, existen más tipos de layouts, cada uno con sus propias características, incluso se da el caso de que el fabricante proporciona diferentes tipos de layouts, los cuales se pueden descargar desde la web. En este caso exponemos el ejemplo del `TabletLayout`, el cual se concibe como una alternativa al `GridLayout`. Este layout divide el contenedor en un conjunto de filas y columnas. La intersección de una fila y una columna se denomina celda. Los componentes que se añaden al contenedor, se sitúan en una celda. Cuando el contenedor cambia de tamaño, la celda también lo hace, en consecuencia, los componentes de su interior cambian de tamaño [6].

Al igual que sucede con este layout, existen muchas otras implementaciones de layouts diferentes en la web. Algunos ejemplos de estos layouts son: `ConstraintLayout`, `CentreLayout`, `StackLayout`, `BasicGridLayout`, `GridLayoutPlus` y `Paragraph Layout`[10].
3. Cuestiones de diseño

3.5. Características del layout que se necesita

A continuación se hará una enumeración de las características del layout que se necesita. De esta forma se podrá comparar nuestro layout con los que proporciona Java y así decidir si podemos o no aprovechar alguno de los que proporciona el lenguaje. El layout debe:

1. Ser capaz de tratar componentes de diferentes tamaños y conservar el aspecto de los componentes.

2. Ofrecer la posibilidad de determinar la separación que deseamos entre los componentes en ambos ejes.

3. Ser capaz de recolocar los elementos cuando el tamaño de la ventana cambia manteniendo siempre las distancias acordadas en un principio y el aspecto de los elementos.

4. Trabajar con un tipo de componente que pueda almacenar al menos dos tipos de aspectos.

5. Ser capaz de proporcionar los métodos necesarios para reducir los componentes que se añadan al contenedor. Es decir, el usuario del layout puede especificar qué reducción quiere que se aplique a los componentes que se añadan al contenedor.

6. Ofrecer total transparencia al usuario, es decir, que se pueda usar de la misma manera que los que proporciona el lenguaje.

7. Ser capaz de distinguir qué elemento está seleccionado en cada momento.

8. Ser capaz de indicar a un elemento previamente seleccionado, que debe cambiar su aspecto.

9. Recolocar los elementos que no cambian su visualización atendiendo a las dimensiones que tiene el elemento que ha cambiado su aspecto.
3.6. **Características del componente que se necesita**

10. Mostrar automáticamente una barra de desplazamiento para desplazarse por el contenedor.

En la tabla 3.1 se muestra un resumen de qué layout cumple qué requisito de los que se ha especificado anteriormente.

<table>
<thead>
<tr>
<th>Layout</th>
<th>Requisito</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlowLayout</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>GridLayout</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>GridBagLayout</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>CardLayout</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>BorderLayout</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>BoxLayout</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>OverlayLayout</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>NullLayout</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
</tbody>
</table>

Cuadro 3.1: Tabla comparativa layouts

Como se puede ver en la tabla 3.1, ninguno de los layouts que proporciona el lenguaje cumple todos los requisitos que se necesitan. En el caso del requisito 1, algunos layouts lo cumplen, y en el caso del requisito 6, lo cumplen todos. Debido a esto, debemos implementar un nuevo layout que satisfaga todos los requisitos que se han expuesto. Atendiendo a las características que se le piden al layout, y teniendo en cuenta que va a ser una generalización de R-Zoom, lo llamaremos **RZ-Layout**.

**3.6. Características del componente que se necesita**

La característica más relevante del tipo de componente que se necesita, es su capacidad para cambiar su aspecto de visualización y que este cambio no esté condicionado por ningún otro elemento. Además, este componente debe poder añadirse a cualquier contenedor, independientemente del layout que se le haya asignado.
3. Cuestiones de diseño

Sin embargo, debido a las características del propio componente, éste no podrá ser visualizado si no se utiliza con *RZ-Layout*.

Debido a que los componentes que proporciona Java sólo ofrecen un tipo de visualización, debemos crearnos un tipo de componente nuevo. Este componente debe ser genérico, es decir, debe permitir la visualización de cualquier tipo de elemento. Al igual que sucede con el layout, al tratarse de una generalización de R-Zoom, denominaremos al componente *RZ-Component*.

3.7. Características de la API

Este proyecto está enfocado al desarrollo de una API genérica que implemente la visualización R-Zoom. A continuación se hará una descripción detallada de las características principales de R-Zoom atendiendo a la función que deben realizar los *RZ-Components* y el *RZ-Layout*.

La característica fundamental que tiene R-Zoom es que los *RZ-Components* deben ofrecer dos tipos de representaciones, esto es, una vista reducida y una vista ampliada. Al cargar el sistema, *RZ-Layout* debe distribuir los *RZ-Components* de izquierda a derecha aplicando sobre ellos una reducción especificada por el usuario del layout. El resultado de aplicar la reducción sobre cada uno de los *RZ-Components* será la vista reducida de cada uno de los componentes.

3.7.1. Distancia inicial de los elementos en el espacio de trabajo

Al iniciar la carga, los elementos se distribuirán por filas de izquierda a derecha, separados entre sí una distancia constante. Cuando no sea posible añadir más elementos a una fila, hay que crear una nueva. Puede darse el caso en el que no sea posible mostrar todas las filas, ya que se ha superado el alto máximo de la ventana. En esta situación, para poder desplazarse por todos los elementos se ha optado por
3.7. Características de la API

añadir una barra de desplazamiento. En la figura 3.1 se muestra un ejemplo de carga de RZ-Layout. En esta figura, los elementos que tienen el mismo color, tienen las mismas dimensiones. La separación constante que hay entre elementos de la misma fila, se representa con el valor de $a$. Los valores $b$ y $c$, hacen referencia a la separación inicial de los márgenes superior e izquierdo respectivamente. Por último, el valor de $d$ hace referencia a la separación que debe haber entre filas, cuando se crea una nueva.

![Diagrama de carga de RZ-Layout](image)

Figura 3.1: Esquema de carga de RZ-Layout

3.7.2. Cambio de tamaño del espacio de trabajo

Una vez explicada la carga de RZ-Layout, durante la ejecución pueden surgir dos eventos: que cambie el tamaño de la ventana o que se seleccione un elemento. Cuando se produce un cambio de tamaño en la ventana puede ocurrir que cambie el ancho, que cambie el alto o ambos. En el caso de cambiar sólo el alto de la ven-
3. Cuestiones de diseño

tana se deberán modificar las propiedades de la barra de desplazamiento, según sea necesario. En el caso de que cambie sólo el ancho tenemos dos alternativas, mostrar las barras de scroll para el desplazamiento horizontal o bien redistribuir los elementos por nuevas filas. En la implementación de *RZ-Layout* se ha optado por volver a redistribuir los elementos por filas. En este caso, el tamaño del scroll vertical debe actualizarse. Por último, en el caso de que se produzca un cambio tanto en anchura como en altura habrá que optar por realizar una solución híbrida entre las dos anteriores. En la figura 3.2 se muestra el resultado de aplicar un cambio en anchura. En estas figuras, los elementos que tienen el mismo color, significan que tienen las mismas dimensiones. Al producirse un cambio en anchura, el número de filas aumenta, por lo tanto es necesario actualizar las propiedades de la barra de desplazamiento.

Figura 3.2: Cambio de tamaño del espacio de trabajo. A la izquierda se puede ver el estado anterior a la ejecución del método *resize*. A la derecha está el estado después de la ejecución del método *resize*.
3.7. Características de la API

3.7.3. Selección de una miniatura

Cuando se selecciona una miniatura, ésta debe cambiar su aspecto, es decir debe mostrarse en su forma ampliada. Al ampliar un elemento se debe realizar una nueva distribución de las demás miniaturas para dar cabida al elemento que tiene el foco. El funcionamiento básico de R-Zoom implica dividir una fila es dos a partir del elemento seleccionado. Sin embargo, dependiendo del tamaño de dicho elemento pueden distinguirse tres situaciones. Al seleccionar un elemento:

Caso básico. Su vista detallada cabe en la fila en la que se encuentra. En consecuencia el resto de los elementos se redistribuyen en altura dentro de la misma fila. Las miniaturas no cambian su posición en el eje de las X, simplemente al valor de la coordenada Y que tiene cada elemento (al estar en la misma fila es igual para todos) se le añade la altura máxima de todas las vistas detalladas de los elementos que estén en la misma fila. De esta forma los elementos a la derecha de la vista detallada, poseen exactamente la misma X que antes de asignar el foco al elemento en cuestión, con la única diferencia de que se sitúan debajo de la vista detallada, alineadas por el margen superior de cada vista en miniatura. En lo referente a la altura, el resto de miniaturas a la derecha del elemento seleccionado la mantienen constante, ya que como se ha dicho, es la altura máxima de las vistas detalladas de los elementos de la misma fila. Teniendo en cuenta que la mayor parte de las veces tiene lugar esta situación, el objetivo que tiene R-Zoom es mostrar las vistas detalladas influyendo lo mínimo posible en la distribución de las demás vistas en miniatura. En la figura 3.3 se muestra un esquema de funcionamiento del caso básico.

Caso alternativo I. Su vista detallada no cabe en la fila en la que se encuentra. Debido a esto se crea una nueva fila con el elemento que tiene el foco y se recoloca el resto de elementos a su derecha alineados por el margen inferior de la vista detallada. En la figura 3.4 se muestra un ejemplo gráfico de este caso.
Figura 3.3: Esquema del caso básico. La figura de la izquierda hace referencia al estado anterior, en el que no hay ningún elemento seleccionado. La figura de la derecha muestra que se ha asignado el foco al tercer elemento de la primera fila.

**Caso alternativo II.** Debido al ancho de la ventana, su vista detallada no puede mostrarse, por lo tanto es necesario aplicarle una reducción.
Figura 3.4: Esquema del caso alternativo I. A la izquierda tenemos el esquema inicial antes de asignar el foco. La imagen de la derecha muestra los cambios que se han hecho para asignar el foco.
Capítulo 4

Descripción informática

En este capítulo se describe el desarrollo del proyecto. Además, se destacarán los principales problemas que se han encontrado en la etapa de desarrollo.

En un primer lugar se analizarán todos los casos de uso que debe tener RZ-Layout. Una vez vista toda la funcionalidad que debe ofrecer RZ-Layout, se estudiará cómo implementar un nuevo layout en el lenguaje elegido. A continuación habrá que centrarse en el desarrollo del RZ-Component. Puesto que ya se tiene la forma de implementar el layout y también la manera de crear un componente genérico, estamos en disposición de pasar directamente a la implementación de RZ-Layout.

Después de explicar en profundidad RZ-Layout, se estudiará su parametrización y generalización para implementar otros tipos de visualizaciones distintas a R-Zoom. Luego se describirán los problemas que se han encontrado en el desarrollo, y finalmente se propondrán algunos ejemplos de uso de RZ-Layout que incluirán algunas extensiones realizadas en el mismo.
4.1. Casos de uso de RZ-Layout

Desde el punto de vista del programador, que será el usuario final de RZ-Layout, éste tiene que ofrecer las funcionalidades que se expresan en el diagrama de casos de uso de la figura 4.1.

Figura 4.1: Casos de uso de RZ-Layout

A continuación se realizará una descripción de cada uno de los casos de uso:
4. Descripción informática

Crear RZ-Layout. Se refiere a que el programador puede crear un RZ-Layout en cualquier lugar del programa en el que se pudiera crear cualquiera de los layouts proporcionados por el lenguaje.

Añadir Componente. Se refiere a que el programador puede añadir los RZ-Component que desee al contenedor que se esté utilizando en cada momento. Aunque no es una funcionalidad directa de RZ-Layout, éste debe ofrecer la misma transparencia que el resto de los layouts de Java.

Preguntar TamañoMax, TamañoMin y TamañoRecomendado. Hace referencia a que el programador puede preguntar al layout el tamaño máximo, mínimo o recomendado que va a tener en cada momento según los componentes que se hayan añadido al contenedor.

Recolocar Componentes. Se refiere a que el layout debe poder recolocar los componentes que se le hayan añadido al contenedor asociado.

Decidir Selección. Se refiere a que el layout debe ser capaz de averiguar qué componente está seleccionado en cada momento. Una restricción que existe es que en cada momento sólo puede haber un elemento seleccionado.

Mover Izquierda, Derecha, Arriba y Abajo. Esta funcionalidad debe permitir mover el foco de selección del layout hacia la izquierda, derecha, arriba o abajo, según corresponda. En todos los casos, si no se ha seleccionado un elemento anteriormente, los eventos de teclado estarán desactivados.

4.2. Desarrollo de un nuevo layout en Java: RZ-Layout

Java permite crear nuevos layouts distintos a los que ya ofrece por defecto. Para ello proporciona una interfaz cuya implementación creará un layout nuevo. La interfaz, que recibe el nombre de LayoutManager, requiere cinco métodos que deben
4.3. Desarrollo del nuevo tipo de componente: 

**RZ-Component**

ser definidos para crear el layout nuevo. A continuación se muestra la especificación de dicha interfaz:

![Especificación de la interfaz LayoutManager](image)

Figura 4.2: Especificación de la interfaz LayoutManager

Hay que aclarar, que al tratarse de una interfaz estos métodos deben implementarse o al menos deben estar presentes en la implementación de *RZ-Layout*.

Como se ha mostrado, para crear un nuevo layout en Java, en teoría simplemente hay que realizar una implementación particular de la interfaz *LayoutManager*. Sin embargo, como se mostrará más adelante, en nuestro caso habrá que implementar algunos otros elementos debido a las necesidades que tiene *RZ-Layout*.

4.3. **Desarrollo del nuevo tipo de componente:**

**RZ-Component**

La idea principal es que el *RZ-Component* ofrezca dos tipos de visualizaciones distintas. Además interesa que se pueda añadir de forma transparente para el usuario a cualquier tipo de contenedor, ya sea un Frame, un Panel, etc.

Como ya se ha dicho anteriormente, a los contenedores sólo se pueden añadir elementos que sean componentes, por lo tanto, para lograr que un *RZ-Component* se pueda añadir a cualquier contenedor, hay que hacer que éste sea un componente. La manera de conseguir esto es haciendo que el *RZ-Component* herede de la clase
Component, que por defecto ofrece el lenguaje Java. El \textit{RZ-Component} se puede añadir a cualquier contenedor, sin embargo, sólo se podrá visualizar correctamente, si dicho contenedor tiene asignado el \textit{RZ-Layout}. El motivo de esto, es que sólo el \textit{RZ-Layout} sabe cómo visualizar correctamente un \textit{RZ-Component}.

Una vez logrado que el \textit{RZ-Component} sea un componente, aún queda pendiente la forma de hacer que éste ofrezca dos tipos de visualizaciones. Esto se podría conseguir fácilmente añadiendo métodos a la clase \textit{RZ-Component} que permitan cambiar su apariencia. Sin embargo, existen una serie de problemas a la hora de aplicar esta solución:

1. El \textit{RZ-Layout} va a trabajar con el \textit{RZ-Component}, que es un componente que no existe en el lenguaje, por lo tanto, a priori debe existir alguna clase que contenga dicho \textit{RZ-Component}.

2. El \textit{RZ-Component} debe ofrecer métodos para cambiar su apariencia, pero la implementación de estos va a depender del programador. Además, se añade el problema de que el programador puede llamar de forma diferente a estos métodos.

3. Este problema deriva del anterior, y se plantea cómo recuperar el método que cambia la visualización de un \textit{RZ-Component}, si no existe un nombre único.

4. \textit{RZ-Layout} debe poder cargar diferentes tipos de \textit{RZ-Components} a la vez.

Para resolver estos problemas hemos decidido convertir el \textit{RZ-Component} en una interfaz, de tal manera que los métodos que necesita \textit{RZ-Layout} siempre estén presentes y además con el mismo nombre. Desde el punto de vista del programador, para crear un \textit{RZ-Component} se debe heredar de la clase Component de Java e implementar la interfaz \textit{RZ-Component}. En la figura 4.3 se muestra dicha interfaz.

Con la solución propuesta se ha conseguido una serie de ventajas:

1. Lograr la independencia entre \textit{RZ-Layout} y el funcionamiento de los \textit{RZ-Component}.
4.4. Especificación detallada de RZ-Layout

2. Permitir la extensibilidad del sistema, ya que ahora la implementación que se haga del RZ-Component va a depender del programador y éste puede decidir los aspectos que puede tener el RZ-Component. Aunque en un principio se diseñó el sistema para que ofreciera dos vistas y que éstas fueran vista resumida y vista detallada, una implementación también válida del RZ-Component sería por ejemplo ofrecer un cambio de color en sus visualizaciones.

3. Permite que se carguen a la vez distintos RZ-Components ya que todos implementan la misma interfaz.

4. Desde el punto de vista de RZ-Layout, permite saber con qué métodos se puede trabajar.

4.4. Especificación detallada de RZ-Layout

En esta sección explicamos de forma detallada los aspectos más relevantes de RZ-Layout. Además, para ofrecer una mejor visión del sistema se añadirá cuando sea posible el diagrama de colaboración oportuno para aclarar aquellos aspectos
4. Descripción informática

que puedan resultar confusos. En la figura 4.4 se muestra la especificación de la clase RZ-Layout.

| RZ展演.
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RZ展演(p_x : Integer, y_sep : Integer, x_max : Integer, f_rec : Double)</td>
</tr>
<tr>
<td>RZ展演(sep_x : Integer, sep_y : int, x_max : int, f_rec : Double, opcion : int)</td>
</tr>
<tr>
<td>actualizar_Scroll() : void</td>
</tr>
<tr>
<td>situarScroll() : void</td>
</tr>
<tr>
<td>Buscar(id : Integer) : Integer</td>
</tr>
<tr>
<td>ControlarEventosRaton() : void</td>
</tr>
<tr>
<td>calcularTamañoMinimo(parent : Container) : Dimension</td>
</tr>
<tr>
<td>keyDown() : void</td>
</tr>
<tr>
<td>keyLeft() : void</td>
</tr>
<tr>
<td>situarElementos(parent : Container) : void</td>
</tr>
<tr>
<td>keyUp() : void</td>
</tr>
<tr>
<td>keyRight() : void</td>
</tr>
<tr>
<td>nuevoFactorReduccion()</td>
</tr>
<tr>
<td>preferidoLayoutSize(parent : Container) : Dimension</td>
</tr>
<tr>
<td>recolocarElementos(parent : Container, comienzo : Integer) : void</td>
</tr>
<tr>
<td>resize(parent : Container) : void</td>
</tr>
<tr>
<td>setPosicionInicio(x : Integer, y : Integer) : void</td>
</tr>
<tr>
<td>setSeleccionado(id : Integer) : void</td>
</tr>
<tr>
<td>localizar_ventana() : void</td>
</tr>
<tr>
<td>recolocarQuitarFoco() : void</td>
</tr>
<tr>
<td>recolocarPonerFoco() : void</td>
</tr>
<tr>
<td>addListenerResize(contenedor : Container, layout : RZ-Layout)</td>
</tr>
<tr>
<td>recolocarElementos()</td>
</tr>
</tbody>
</table>

Figura 4.4: Diagrama de la clase RZ-Layout

En primer lugar nos centraremos en describir los métodos que ofrece la interfaz LayoutManager, es decir, para qué sirven y cuándo son invocados y luego nos centraremos en los demás métodos que se han tenido que implementar y que permiten el funcionamiento específico de RZ-Layout.

layoutContainer. Es quizás el método más importante de la interfaz ya que se invoca automáticamente cuando se produce un cambio en el tamaño del contenedor o bien cuando cambia la visibilidad del mismo, es decir, que se oculte
4.4. Especificación detallada de RZ-Layout

o se haga visible. Su cometido es colocar los componentes de una forma específica dentro del contenedor. Variando la forma de posicionar los componentes en este método varía totalmente la apariencia que tendría el contenedor.

**addLayoutComponent y removeLayoutComponent.** Estos métodos se utilizan para añadir o eliminar componentes al layout. Hay que tener en cuenta que ésta no es la forma habitual de trabajar con un layout, ya que normalmente esta tarea se realiza utilizando el contenedor no el layout. Desde el punto de vista de RZ-layout no interesa que los componentes se puedan añadir o eliminar directamente, por ello estos método no se han implementado.

**preferredLayoutSize.** Este método es el encargado de calcular el tamaño óptimo del contenedor. En el caso de RZ-Layout el tamaño óptimo es el tamaño mínimo que debe tener el contenedor, por lo tanto el tamaño mínimo y el óptimo van a coincidir. Para calcular el tamaño óptimo tenemos que utilizar un ancho máximo como referencia, éste último se le solicita al usuario cuando se crea el RZ-Layout. Basándose en el ancho máximo se simula una distribución por filas de izquierda a derecha. Cuando se llegue a la situación en la que se cumpla \( PosXComponente + anchoComponente + separacion_x > anchomaximo \), se debe simular la creación de una fila nueva. Para situar al primer elemento a partir de la segunda fila, ésta inclusive, es necesario calcular la altura máxima de las vistas en miniatura de los elementos de la fila anterior, ya que se permite que los elementos tengan diferentes tamaños. Una vez que tenemos la altura máxima, los elementos restantes se colocarán a la altura indicada por la expresión \( altura = altura + maximoAlturaFilaAnterior + separacion_y \). Finalmente, mientras queden componentes en el contenedor será necesario repetir estos pasos. Como se puede ver, lo que se hace es distribuir los elementos en anchura de tal forma que se aproveche al máximo el ancho dado por el usuario, de esta forma la altura que tenga el contenedor dependerá de dos factores: del ancho máximo que se indique y del número de componentes que tenga el contenedor.

**minimumLayoutSize.** Este método es el encargado de calcular el tamaño mínimo del contenedor. La forma de calcularlo es igual que en el método anterior.
4. Descripción informática

4.4.1. Crear un nuevo RZ-Layout

Cuando se crea un RZ-Layout, es necesario suministrarlo a la clase una serie de datos para el correcto funcionamiento del layout. A continuación se especifican los datos necesarios para crear un nuevo RZ-Layout.

- **Separación en horizontal y vertical**: es necesario saber la distancia de separación entre elementos de la misma fila, y entre diferentes filas.

- **Ancho máximo**: indica el ancho máximo que puede tener el contenedor.

- **Factor de reducción**: se le solicita al usuario un factor de reducción que se aplicará sobre todos los elementos que se añadan al contenedor.

Aparte de guardar estos datos, al crear el RZ-Layout se crean los listener necesarios para recibir los eventos de ratón y se procede a formar una encapsulación de contenedores que permitan la visualización de manera correcta y que cumpla todos los requisitos necesarios.

4.4.2. Encapsulación de contenedores: barras de scroll

Al crear el RZ-Layout se realiza una inicialización de contenedores, en concreto se crea un panel que se inserta dentro de un ScrollPane. El ScrollPane es el elemento encargado de mostrar las barras de desplazamiento cuando sea necesario. Desde el punto de vista de RZ-Layout, los elementos se añadirán al panel, éste a su vez se añadirá al ScrollPane y éste último al contenedor sobre el que se aplicó el layout.

Cuando se añade un RZ-Component al contenedor, RZ-Layout toma el control y añade el componente a su panel interno. Como el panel se encuentra en el interior de un ScrollPane, al variar su tamaño se actualizarán las barras de desplazamiento.
4.4. Especificación detallada de RZ-Layout

Aún queda pendiente cómo se van a mostrar las barras de desplazamiento. El ScrollPane va tener siempre las mismas dimensiones que el contenedor sobre el que se aplica RZ-Layout. En cuanto al panel, en él se van a añadir los componentes que se añadan al contenedor. El tamaño del panel va a coincidir con el tamaño óptimo que debería tener el contenedor. Si se cumple TamPanel > TamScrollPane entonces deben mostrarse las barras de desplazamiento.

En la figura 4.5 se muestra un gráfico que representa la encapsulación de contenedores.

![Encapsulación de contenedores del RZ-Layout](image)

Figura 4.5: Encapsulación de contenedores del RZ-Layout

4.4.3. Nuevo tipo de componente: RZ-ComponentInformado

En los apartados anteriores se ha dicho que el RZ-Layout va a trabajar directamente con el RZ-Component. Sin embargo, para ocultar ciertos aspectos de la
4. Descripción informática

Implementación, \textit{RZ-Layout} va a trabajar con un componente que contiene mayor información que el \textit{RZ-Component}, se trata del \textit{RZ-ComponentInformado}.

En definitiva, el \textit{RZ-Component} va a ser el componente con el que debe trabajar el programador, es decir, debe implementar dicho componente, y añadirlo al contenedor. Cuando el \textit{RZ-Layout} tome el control, recogerá todos los \textit{RZ-Component} y para cada uno de ellos creará un \textit{RZ-ComponentInformado}. Hay que tener en cuenta que esta acción sólo se realiza una vez, en concreto, la primera vez que se hace visible el contenedor que tiene asociado el \textit{RZ-Layout}. A continuación se muestran los campos que tiene el \textit{RZ-ComponentInformado}:

- **Fila y columna**: estos campos indican la fila y columna del panel interno del \textit{RZ-Layout} en la que se encuentra el componente.

- **Identificador**: cuando se añaden al panel los componentes, se les asigna un identificador único, el cual va a ofrecer una serie de ventajas a la hora de recolocar los elementos.

- **RZ-Component**: el \textit{RZ-ComponentInformado} debe guardar el \textit{RZ-Component} sobre el que se creó para poder acceder a los elementos de interés.

Cabe destacar que además de conseguir la ocultación de información, la utilización del \textit{RZ-ComponentInformado} optimiza el funcionamiento interno de \textit{RZ-Layout} (véase 4.4.8).

4.4.4. Transparencia de uso de \textit{RZ-Layout}

La transparencia de uso se refiere a que \textit{RZ-Layout} debe poder utilizarse de la misma forma que se utiliza cualquier otro layout de Java, además debe permitir que los componentes se puedan añadir al contenedor de la misma forma que se haría con otro layout, es decir, utilizando el método \texttt{add(Component)} de los contenedores.
4.4. Especificación detallada de RZ-Layout

En la sección 4.3 se ha explicado el motivo por el que el \textit{RZ-Component} debe heredar de la clase Component de Java. En este apartado se explicará cómo se obtiene un \textit{RZ-Component} a partir de los componentes que contiene el contenedor.

Aunque el \textit{RZ-Component} es un componente, cuando se añade a cualquier contenedor, éste lo transforma en un elemento de la clase \textit{Component}, es decir, ya no se conserva la estructura propia de un elemento \textit{RZ-Component}. Por esta razón, es necesario transformar dicho elemento a la clase \textit{RZ-Component}. \textit{RZ-Layout} va a tomar el control del contenedor cuando este último o en su defecto la ventana de la aplicación se hace visible. Justo en este momento se invoca el método \textit{LayoutContainer} de \textit{RZ-Layout}, el cual, si ve que es la primera carga del sistema, lo que hace es invocar a un método interno denominado \textit{add()}. A continuación se explicará el funcionamiento del método \textit{add()}, de una manera más detallada. Este método realiza las siguientes tareas para cada uno de los componentes que tiene el contenedor:

- Obtención de un \textit{RZ-Component}: una vez que el \textit{RZ-Layout} obtiene la referencia del contenedor sobre el que se ha aplicado, se puede acceder a los componentes que hay en su interior. Para cada componente hay que realizar una conversión de \textit{Component} a \textit{RZ-Component}.

- Obtención de la representación en miniatura: cuando se consigue tener un \textit{RZ-Component} se le comunica que debe reducirse pasándole el factor de reducción que se almacenó cuando se creó el layout.

- Creación del \textit{RZ-ComponentInformado}: al tener el \textit{RZ-Component} y una vez obtenida la versión reducida, ya se puede crear el \textit{RZ-ComponentInformado}, asignándole un identificador válido. Sin embargo, la información sobre la fila y la columna aún no se conoce, por lo que no tendrían un valor válido.

- Creación de la tabla Hash: hasta este punto no se ha comentado el uso de la tabla Hash. En concreto \textit{RZ-Layout} utiliza una tabla Hash en la que se almacenan los \textit{RZ-ComponentInformado}. La clave que usará la tabla Hash se obtiene del valor \textit{hashCode} que tiene la vista sin enfocar del \textit{RZ-ComponentInformado}. 
4. **Descripción informática**

Es necesario comentar que el valor que proporciona el `hashCode` es único, es decir, Java no genera dos `hashCode` iguales en la misma ejecución, por ello hay que tener en cuenta que en una hipotética implementación del `RZ-Component` puede provocar que los métodos `setFoco` y `removeFoco` puedan crear un nuevo componente, cambiando así el `hashCode` del componente. Debido a esto último, es necesario mantener actualizada en todo momento la tabla Hash, de lo contrario no se podrían recuperar los elementos en su interior.

- **Creación del vector de `RZ-ComponentInformado`:** como ya se dijo anteriormente, al `RZ-ComponentInformado` se le asignó un identificador, éste va a coincidir con su posición en el vector. Esto va a disminuir considerablemente el tiempo de acceso al vector en las situaciones de recolocación de elementos. La función principal que tiene este vector es almacenar los componentes en el orden en el que se han cargado en el sistema.

- **Inserción de `MouseListener` a los `RZ-ComponentInformado`:** es necesario añadir los eventos de ratón a cada uno de los componentes, ya que ellos serán los que recibirán estos eventos y le comunicarán al `RZ-Layout` que han sido seleccionados.

### 4.4.5. Distribución total de componentes

La distribución de componentes se encarga de colocar los componentes en el contenedor, independientemente del tipo de visualización que ofrezcan. Antes de empezar la colocación de componentes se deben eliminar todos los componentes que estén presentes en el panel de `RZ-Layout`, esto es muy costoso, por lo que habrá que evitar en la medida de lo posible realizar una distribución total de componentes. Los componentes se van a colocar de izquierda a derecha dejando entre sí la separación que se indica en la creación del layout. Además, la distribución debe tener en cuenta que si se sobrepasa el ancho máximo establecido para el layout será necesario crear una fila nueva. Suponiendo que todos los elementos se muestran con su visualización de miniatura, el espacio que hay que dejar entre filas es `separation_y + maximoAltoFilaAnterior`. 

33
4.4. Especificación detallada de RZ-Layout

La distribución de componentes es similar al cálculo del tamaño óptimo del contenedor que se ha explicado anteriormente, sin embargo, existe una diferencia notable, aparte del hecho de que en este caso sí se añaden los componentes al panel. Además de lo evidente, en la distribución no se va a utilizar el ancho máximo del layout para crear una nueva fila, sino el valor del ancho óptimo, lo que se conoce como \( x_{recomendada} \). Por lo tanto, se añadirá un elemento a una fila cuando se cumpla \( x_{Elemento} + \text{separacion}_x + AnchoElemento < x_{Recomendada} \), si no se cumple esto habrá que calcular la nueva altura a la que se deben situar los elementos restantes utilizando la expresión anterior. Este proceso se va a repetir hasta que no queden elementos.

Finalmente hay que destacar que a la hora de distribuir todos los elementos vamos a utilizar el vector creado en la ejecución del método \( \text{add} \). Tiene sentido usar el vector ya que se necesita acceder a todos los componentes, no a uno en concreto, en tal caso sería mejor utilizar la tabla Hash que también se creó en dicho método. Como se puede observar, el coste que va a tener la recolocación total de elementos será como mínimo igual a \( n \), siendo \( n \) el número de elementos que tiene el vector de \( RZ-ComponentInformado \). Aunque en una primera instancia parece que el coste no es muy elevado, puede degradar considerablemente el sistema. Los cambios que provocan los eventos que tienen lugar en la ejecución deben percibirse lo mínimo posible, por lo tanto, un coste polinómico puede ser muy elevado. La degradación del sistema se puede observar cuando empiezan a tener lugar una mayor cantidad de parpadeos.

4.4.6. Resize: recolocación de componentes

Cuando se produce un cambio en el ancho del contenedor sobre el que se ha aplicado \( RZ-Layout \) es necesario recolocar los componentes. En este apartado vamos a explicar cómo se consigue asignar el listener que provocará el resize del layout, y luego explicaremos cómo funciona la recolocación de componentes.

Como ya hemos visto, al producirse un cambio en la visibilidad del contenedor
4. Descripción informática

que tiene asociado **RZ-Layout**, en concreto la primera vez que se hace visible, se ponen en funcionamiento una serie de métodos. Entre todos ellos, existe uno que es el encargado de asignar el listener oportuno al contenedor para que éste sea capaz de recibir los eventos de resize. Hay que tener en cuenta que dicho listener debe asignarse al contenedor que tiene asociado el layout. Aunque el layout sería el componente ideal para tener asignado este listener, en la realidad vemos que el layout nunca se visualiza debido a que es un elemento que no tiene aspecto visual, por lo tanto, el usuario no puede tener interacción directa con él. Esto, va a impedir que el layout pueda tener un listener directo que permita saber cuándo se ha producido un cambio de tamaño. En nuestro caso, se ha optado, como ya se ha adelantado, por asignar el listener al contenedor que tiene asociado el layout. Por lo tanto, cuando se produzca un cambio de tamaño del contenedor, el listener que tiene asociado invocará al layout, indicándole que ha cambiado el tamaño y que debe actuar en consecuencia. Este proceso se plasma en el diagrama de colaboración que se muestra en la figura 4.6.

![Diagrama de colaboración de asignación de evento de resize](image.png)

**Figura 4.6: Diagrama de colaboración de asignación de evento de resize**

Un problema derivado del hecho de que el layout se pueda asignar a cualquier contenedor es que puede darse el caso en el que el layout no se asigne directamente sobre la ventana de aplicación, sino sobre un contenedor interno, sobre el cual no se permite que se produzca un cambio de tamaño por parte del usuario. En este caso, la ventana de aplicación puede tener asociado cualquier otro layout disponible en el lenguaje, sin embargo, cuando se produce un cambio de tamaño en la ventana de aplicación puede que no se genere una recolocación de componentes ya que,
4.4. Especificación detallada de RZ-Layout

dependiendo del layout del lenguaje que se utilice, puede que se produzca o no un cambio de tamaño en los componentes que contiene la ventana. Para intentar solucionar esto se plantea usar un layout que sea capaz de ajustar el tamaño de los componentes que tiene el contenedor al propio tamaño de éste.

Ahora nos centramos en el funcionamiento del propio resize. La forma estándar de realizar el resize, es simplemente actualizando el parámetro de ancho máximo con el valor de ancho actual del contenedor cuando se produce el resize. Una vez que se tiene este dato, habrá que calcular el tamaño óptimo del layout y luego recolocar de nuevo todos los elementos. En la figura 4.7 se expone el diagrama de colaboración del funcionamiento básico del resize.

**Figura 4.7: Diagrama de colaboración de registros de eventos de resize**

**Optimización de Resize**

La optimización que se propone del método resize no va orientada a mejorar el rendimiento del propio método, sino en reducir el número de veces que se invoca.

Durante la ejecución, el contenedor puede cambiar en anchura, en altura o bien en ambas dimensiones a la vez. A continuación se explicará lo que debe hacer el método resize en cada uno de estos casos:

- **Cambio en anchura**: cuando se detecta un cambio de este tipo hay que recolocar los elementos y actualizar las barras de scroll del ScrollPane. Para
actualizar dichas barras se utiliza el método `validate()` el cual introduce un poco de retardo en la actualización de la imagen. El tamaño del contenedor se debe fijar con los parámetros x e y-recomendada, así puede darse el caso de que el usuario amplíe el contenedor pero éste automáticamente se haga un poco más pequeño o más grande según sea más óptimo para encajar los elementos.

- Cambio en altura: en este caso no se realiza ninguna recolocación, lo único que se realiza es variar el tamaño del scroll de la misma manera que en el caso anterior. En este caso, al no realizarse una recolocación de elementos, sí es posible fijar el alto de la ventana como la requiere el usuario, en concreto se fija al tamaño `(xRecomendada, alturaActual)`.

- Cambio de anchura y altura: en anchura se mantiene la misma política que en el primer caso y en altura se sigue con el segundo punto.

Si se estudia en detalle la implementación propuesta del método `resize`, se puede ver que es muy costosa, esto se debe a tres razones:

- Al calcular el tamaño óptimo se debe recorrer todo el vector de componentes cargados en el sistema. Además hay que utilizar un vector para calcular el valor máximo de las alturas, lo que se hará por cada fila. Por otro lado, se utiliza otro vector para decidir la x-recomendada que debe tener el layout, esto se realiza una vez, ya que hay que tener en cuenta el ancho de todas las filas. El coste que tiene este método es polinómico, pero depende del número de elementos que haya en el sistema y del ancho del layout, ya que este valor va a influir en el número de elementos que haya por fila y a su vez esto influirá en el número de elementos del vector que almacena las alturas. Por último hay que añadir el coste de buscar el máximo en el vector de las x (almacena el ancho de la fila), el cual va a tener un tamaño que será igual al número de filas.

- A la hora de situar de nuevo los elementos también vamos a tener un coste polinómico, ya que prácticamente el método que realiza esto es igual al mé-
todo que calcula el tamaño óptimo del layout, con la diferencia de que cada vez que se distribuyen los elementos se borra completamente el panel interno, esto introduce el correspondiente parpadeo.

- Si se da el caso de que existe un elemento seleccionado cuando se produce el resize, habrá que volver a fijar el foco y además recolocar los elementos que se vean afectados. Después de hacer esto se tendrá que realizar lo indicado en los puntos anteriores.

Una vez estudiado el funcionamiento del resize, se puede observar que no siempre es necesario realizar una recolocación de los elementos aunque cambie el ancho del contenedor. Esto quiere decir que aunque cambie el tamaño del contenedor, puede que para dicho cambio no sea necesario realizar una recolocación de elementos, por lo que aunque se invoque al método resize, éste no debe realizar ninguna tarea.

La forma de especificar si un cambio de tamaño del contenedor implica ejecutar el resize en su totalidad, es fijando un umbral, por debajo del cual no se realiza ninguna tarea de recolocación. A continuación se explicará cómo y dónde se calcula dicho umbral y cómo se utiliza en el método resize.

Durante el estudio de esta optimización se observó que lo que interesaba era añadir el cálculo del umbral en un lugar donde el valor de \( x_{-}\text{recomendada} \) estuviera siempre actualizado. Por ello se llegó a la conclusión de que el mejor lugar para calcular el umbral era en el método \( \text{calcularTamanoMinimo()} \) que es invocado desde los métodos \( \text{preferredLayoutSize, minimumLayoutSize, layoutContainer y resize} \).

El cálculo del umbral requiere que se recorra una vez más el vector que almacena el ancho de cada una de las filas. Para cada valor del vector se debe calcular la siguiente expresión:

\[
diff = \abs(x_{\text{Recomendada}} - v(i))
\]
4. Descripción informática

Al concluir el recorrido del vector se debe seleccionar el valor mínimo de las diferencias, esto se debe a que se desea que siempre estén visibles todos los elementos. Si no escogiéramos el valor mínimo puede darse el caso de que en ocasiones al hacer el resize, no se recolocaran los elementos cuando sí que sería necesario que lo hiciesen. Para concluir con la forma de cálculo del valor del umbral, hay que comentar que si el valor de la diferencia mínima es menor que la separación-x, habrá que utilizar como valor de umbral la separación-x.

Ya calculado el umbral, sólo queda ver dónde se utiliza este valor. Como es lógico lo vamos a utilizar en el método resize, pero habrá que añadir una lógica que controle si se debe o no cambiar. Esto va a consistir simplemente en ver si se cumple la condición \( \text{abs}(x_{Recomendada} - x_{maxNuevo}) \geq \text{cambioMin} \), donde \( x_{maxNuevo} \) será el valor actual del ancho del contenedor. Si se cumple esta condición habrá que aplicar el resize, en caso contrario se habrá evitado una recolocación innecesaria. En la figura 4.8 se muestra el diagrama de colaboración de la optimización del resize.

![Figura 4.8: Diagrama de colaboración para la optimización de resize](image)

Como se puede ver lo único que cambia en el diagrama es que se ha añadido una condición para ver si es necesario realizar el resize o no.
4.4.7. Averiguar el componente seleccionado

El proceso para averiguar el componente que se encuentra seleccionado en cada momento, es otro de los aspectos fundamentales de RZ-Layout. Esta funcionalidad está muy relacionada con el método `add()` que se ejecuta al comienzo. Cuando se cargan los componentes en el sistema con el método `add()`, es necesario añadirles un listener de ratón. En una primera instancia se planteó añadir el listener simplemente al layout, pero esto no permitía distinguir qué elemento se encontraba seleccionado en cada momento. Además, aunque no se seleccionara un elemento se seguía recibiendo eventos, lo cual no interesaba. Por esta razón se optó por añadir a cada componente que estuviera en el layout un listener, de esta forma se conseguía que sólo se recibieran eventos cuando se seleccionaba un componente.

Para poder explicar correctamente este apartado hay que entrar en ciertos detalles. A la hora de asignar los listener a los componentes, estos no se añaden directamente sobre los RZ-Component sino que lo hacen sobre el objeto de la clase Component que devuelve el método `getElement()` del interface RZ-Component. Esto permite añadir unos listener sobre unos componentes que se encuentra en el interior de un RZ-Component que a su vez está encapsulado en un RZ-ComponentInformado. Existen dos razones para hacer esto:

- El layout debe trabajar con un tipo de componente genérico, que sea implementable por el programador y que además ofrezca una serie de métodos estándar.
- Como el RZ-Component es una interfaz no puede heredar de ninguna otra clase, por lo tanto un RZ-Component no pertenece a la clase Component. Quien sí pertenece a dicha clase es la implementación del RZ-Component.

Aunque se haya conseguido añadir un listener al componente de la forma comentada anteriormente, aún queda un problema pendiente. Cuando se produce un evento `mouseclick` se ejecuta el método `mouseClicked`, de la clase `eventosRaton`, al cual se le pasa como parámetro un objeto de la clase `MouseEvent`. Este objeto sólo
4. Descripción informática

permite obtener un elemento de la clase Component y resulta imposible convertirlo a RZ-Component, ya que verdaderamente se añadió el listener a un elemento de la clase Component. Por lo tanto, el problema en este caso está en obtener un RZ-Component o RZ-ComponentInformado a partir de un componente de otra clase, el cual no permite una conversión explícita. Como dijimos anteriormente, en el método add() se crea una tabla Hash con los valores de la forma (clave,elemento) donde la clave es un hashCode y el elemento es un RZ-ComponentInformado. Así pues, el valor de la clave, se obtiene a partir del hashCode de un elemento de la clase Component que devuelve el método getElement(). En cuanto al objeto de la clase RZ-ComponentInformado, éste se crea a partir de los componentes obtenidos del contenedor. De esta forma, se vincula un objeto de la clase RZ-ComponentInformado con una clave única en el sistema.

Cuando se produce un evento de clic sobre un elemento, se ejecutan una serie de acciones:

1. Al capturarse el evento de clic, en el interior del método mouseclicked se averigua el hashCode del componente.

2. Se invoca al método setSeleccionado() pasándole como parámetro el hashCode del elemento.

3. Usando el hashCode, se recupera de la tabla Hash la referencia al objeto de la clase RZ-ComponentInformado.

4. Si hay algún componente que se ha seleccionado anteriormente, hay que invocar al método recolocarQuitarFoco(). Este método se encargará de quitar el foco del componente antiguo y recolocar los elementos a su forma original.

5. Se le comunica al nuevo componente seleccionado que debe cambiar su aspecto visual.

6. Se invoca al método recolocarPonerFoco, el cual situará correctamente el elemento seleccionado y redistribuirá el resto de los elementos.
Hay que aclarar que otra solución posible, hubiera sido añadir a la interfaz *RZ-Component* un método para añadir eventos de ratón a los componentes, sin embargo, esto provocaría una pérdida de control sobre la selección de elementos y se perdería la ocultación de la información, ya que de algún modo habría que transmitir la información que necesita el layout. En la figura 4.9 se muestra el diagrama de colaboración que representa este proceso de selección.

Figura 4.9: Diagrama de colaboración para la selección de componentes

Este diagrama hace referencia a cuando se le indica a un componente que en ese momento tiene el foco, si quisiéramos indicar el caso en el que quitamos el foco a un componente simplemente habría que cambiar el método `setFoco()` por `removeFoco()`.

### 4.4.8. Recolocación R-Zoom

Una vez que hemos comentado las características más relevantes de *RZ-Layout* y en particular cómo se identifica el componente seleccionado, ya estamos en disposición de explicar en detalle el funcionamiento de R-Zoom.

Como ya se ha dicho, R-Zoom es una técnica que permite la visualización de elementos, y que entra dentro del tipo *Foco + Contexto*. La característica fundamental que tiene este tipo de visualización es que permite ver simultáneamente la
4. Descripción informática

vista detallada de un objeto y su contexto. En consecuencia, es fundamental que al mostrar una vista detallada, los cambios que se produzcan alrededor de dicha vista sean los mínimos posibles, de esta manera se conservará el contexto en el que está el elemento y además podremos visualizar la vista detallada de dicho elemento.

Para describir el funcionamiento de R-Zoom supongamos que hemos creado un RZ-Layout con los siguientes datos: separación con el margen superior igual a 50 píxeles, separación con el margen izquierdo igual a 10 píxeles, separación entre elementos de la misma fila igual a 25 píxeles y separación entre filas igual a 30 píxeles.

En la figura 4.10 se muestra el resultado de cargar un directorio de imágenes con estos datos iniciales.

Cuando se selecciona un elemento del contexto, para ser el nuevo foco, el comportamiento general de R-Zoom consiste en dividir una fila en dos. El punto de ruptura de la fila coincide con el elemento que tiene el foco, de esta forma los elementos anteriores a él no se ven afectados. Por lo tanto, sólo es necesario colocar correctamente el foco y recolocar los elementos que están a su derecha.

Teniendo en cuenta el comportamiento general de R-Zoom, atendiendo en todos los casos al esquema de carga inicial (figura 4.10) y suponiendo que a la hora de seleccionar un elemento no existe un elemento previo ya seleccionado, podemos describir a continuación cada una de las situaciones que pueden tener lugar en R-Zoom: caso básico, cuando la vista detallada del elemento cabe en la fila; caso alternativo I, cuando la vista detallada del elemento no cabe en la fila y es necesario crear una nueva; y caso alternativo II, cuando la vista detallada no solo no cabe en la fila sino que no cabe en el propio contenedor, por lo que hay que reducir el elemento.
4.4. Especificación detallada de RZ-Layout

Caso básico

Este caso tiene lugar cuando al asignar el foco a un elemento, éste cabe en la fila en la que se encuentra. Hablando en términos del algoritmo, este caso tiene lugar cuando la condición $\text{PosX} + \text{AnchoElemento} \leq x_{\text{Recomendado}}$ se cumple y en contrapartida, la condición $\text{AnchoElemento} > \text{AnchoContenedor}$ no se cumple. Con estos datos ya sabemos que nos encontramos en el caso básico. Aunque ya sabemos qué elemento ha sido seleccionado e incluso se le ha dicho al elemento que debe cambiar su apariencia, este cambio no se hará efectivo hasta que se lo comuniquemos al panel interno del layout. Por lo tanto, el algoritmo que implementa R-Zoom debe realizar las siguientes tareas:
4. Descripción informática

1. Eliminar la miniatura del panel interno y en su posición añadir la vista detallada del elemento que se encuentra seleccionado con sus dimensiones reales.

2. Redistribuir el resto de componentes a la derecha del elemento seleccionado. Al seleccionar un elemento, el resto de elementos a su derecha deben ser redistribuidos. La idea es dividir la fila del elemento seleccionado en dos, de tal forma que no sea necesario redistribuir los elementos a la izquierda del elemento con el foco. Los elementos que están a la derecha del elemento seleccionado y además están en la misma fila hay que desplazarlos simplemente en altura, es decir, estos elementos van a conservar su coordenada X, pero su coordenada Y se va a ver aumentada en la cantidad que resulte de averiguar la altura máxima de las vistas detalladas de los elementos de la misma fila. Con esto se asegura que mientras naveguemos por elementos de la misma fila, las miniaturas siempre se desplazarán en altura una cantidad constante, lo que va a contribuir a conservar el contexto de los elementos. En cuanto al resto de elementos a la derecha del elemento seleccionado, es necesario desplazarlos hacia abajo para evitar que se solapen con otros elementos. Gracias a que se utiliza un RZ-ComponentInformado, sabemos en qué posición del vector interno se encuentra el componente seleccionado, por ello para identificar los componentes a su derecha simplemente es necesario recorrer el vector a partir de la siguiente posición. Si no se realizara esto, sería necesario recorrer para todos los casos el vector interno, lo que lógicamente influye negativamente en el rendimiento del layout. Nótese que a medida que se selecciona un elemento que diste más del borde izquierdo superior, el tiempo que se tarda en redistribuir los elementos va a ser menor, asimismo, si se selecciona el primer elemento de todos, el coste que tiene la reordenación va a ser de O(n).

3. Actualizar el tamaño de las barras de desplazamiento. Como el tamaño del panel interno de RZ-Layout ha cambiado, es necesario actualizar el tamaño de las barras de desplazamiento para permitir navegar por todos los componentes.

4. Actualizar la posición de las barras de desplazamiento. Como el tamaño del elemento seleccionado ha cambiado y también lo ha hecho el tamaño del pa-
4.4. Especificación detallada de RZ-Layout

En la figura 4.11 se muestra un ejemplo de este caso.

Figura 4.11: Imagen del caso básico con foco

Caso Alternativo I

Este caso tiene lugar cuando al asignar el foco a un elemento, no es posible mostrarlo en su forma detallada, ya que no cabe en la fila en la que se encuentra. En
4. Descripción informática

términos del algoritmo, la condición $PosX + \text{anchoElemento} \leq x\text{Recomendada}$ no se cumple. Lo mismo sucede con la condición $\text{anchoElemento} > \text{AnchoContenedor}$. Al igual que en el caso anterior, aunque sabemos qué elemento ha sido seleccionado y le hemos indicado que debe cambiar su apariencia, este cambio no se va a reflejar hasta que actualicemos el panel interno del layout. El conjunto de pasos que realizamos en este caso es muy parecido a la situación anterior. En lo referente al elemento seleccionado, la expresión que configura su nueva coordenada $Y$ es $\text{AlturaFinal} = \text{alturaActual} + \text{AlturaMaximaFila}$. Además de este cambio, el valor de su coordenada $X$ se va a ver afectada. Si nos fijamos, esta situación no ha sucedido hasta ahora en ningún momento, ya que el objetivo fundamental de R-Zoom es variar lo mínimo posible la situación de los elementos visualizados. Así, el elemento seleccionado va a desplazarse hacia la izquierda una cantidad que no va a ser constante, es más, dicho valor va a ser el valor mínimo que permite que el elemento pueda mostrarse en su vista detallada. El resultado de esto, es que el elemento seleccionado va a ser desplazado de tal forma que pueda situarse en el contenedor. Por otro lado, los elementos que se encuentran a la derecha de la vista detallada, se van a recolocar en altura, pero, sin embargo, van a mantener su coordenada $X$. La expresión que configura su nueva coordenada $Y$ es $\text{AlturaFinal} = \text{alturaActual} + \text{MaximoAlturaFila} + \text{AlturaElementSeleccionado} - \text{AlturaComponente}_i$. Esta expresión se representa en la figura 4.12. El valor $\text{MaximoAlturaFila}$ corresponde a la altura máxima de las vistas detalladas de los elementos de la misma fila.

El objetivo de esta expresión, es alinear los elementos a la derecha del elemento seleccionado por su margen inferior. Antes de analizar la expresión en detalle, hay que tener en cuenta que el término $\text{AlturaComponente}_i$ se refiere a cada uno de los elementos que se encuentra a la derecha del elemento seleccionado y que están en su misma fila. Si sustituimos esta expresión por $\text{alturaFinal} = \text{alturaActual} + \text{MaximoAlturaFila}$, y hacemos que los elementos a la derecha se alinee a esta altura, lo que estaríamos haciendo es alinear los elementos por el margen superior. Al añadir el término $\text{AlturaElementSeleccionado}$ lo que se hace es situar los elementos que estén a la derecha, por debajo del elemento seleccionado. Por último, al añadir el término $\text{AlturaComponente}_i$, con signo negativo, lo que hacemos es que el elemento retroceda justo la cantidad que le permite alinearse
4.4. Especificación detallada de RZ-Layout

Figura 4.12: Esquema del cálculo de la nueva coordenada Y por el margen inferior. En la figura 4.13 se muestra un ejemplo de este caso.

Caso Alternativo II

Este caso tiene lugar cuando el elemento que se ha seleccionado no se puede mostrar completamente en su forma detallada. La principal diferencia de esta situación con la anterior, es que el elemento que tiene el foco no puede mostrarse en su forma detallada porque no cabe en el contenedor. Por esta razón para poder visualizar dicho elemento es necesario aplicarle una reducción que puede variar dependiendo de las dimensiones del contenedor y de las del propio elemento. En términos del algoritmo, lo que sucede es que la condición $PosX + \text{anchoElemento} \leq xRecomendada$ no se cumple, pero sí se cumple la condición
4. Descripción informática

Figura 4.13: Imagen del caso alternativo I con foco

anchoElemento > AnchoContenedor. Con esta última, sabemos que el elemento debe ser reducido. Para reducir un elemento dependemos de la implementación que se realice del método aplicarReducción(anchoDisponible,altoDisponible) del componente con el que estamos trabajando. A dicho método, se le va a informar del ancho y alto que tiene disponible el elemento, por lo tanto, será responsabilidad del programador del RZ-Component proporcionar al layout un elemento de dichas dimensiones si se desea que el elemento se visualice en su totalidad.

4.4.9. Eventos de teclado

Los eventos del teclado se refieren a la capacidad que tiene el layout de seleccionar un nuevo foco, utilizando para ello el teclado.

El problema fundamental que tiene la asignación de los eventos de teclado, es decidir qué componente va a ser el responsable de reaccionar a dichos eventos. En una primera instancia, decidimos que al igual que sucede con los eventos del ratón, sean los propios componentes los que reaccionen a estos eventos. Sin embargo, la práctica demuestra que los elementos son incapaces de reaccionar a dichos even-
tos. Debido a esto, hemos decidido que sea la ventana de aplicación la que reciba los eventos de teclado, y luego actúe en consecuencia. Para comprender esta decisión hay que tener en cuenta las diferencias existentes entre un evento de ratón y un evento de teclado. En el caso de los eventos de ratón, se puede asignar un listener a cada elemento porque sólo uno será el que reciba el evento en un momento determinado. Por ejemplo, cuando se produce un evento `mouseClick()`, será el elemento sobre el que se hizo clic y solamente este elemento, el que reciba el evento. Sin embargo, cuando se trata de un evento de teclado, si asignamos a cada elemento un listener para el teclado, cuando se produzca, por ejemplo, un evento `keyDown()`, reaccionarán todos los elementos. Este funcionamiento hace que sea imposible decidir qué nuevo elemento seleccionar, lo que provoca que el foco no se mueva. El hecho de tener un sólo listener de teclado asignado a la ventana de aplicación permite que el comportamiento sea el adecuado.

Una vez que hemos decidido el elemento que va a reaccionar a los eventos de teclado, aún queda decidir si esto va a ser responsabilidad del usuario del layout, o del propio layout. Se ha optado por una solución intermedia, es decir, por defecto el layout va a asignar los eventos al contenedor más externo, pero, se permite que el usuario pueda desactivar esta opción y añadir manualmente dichos eventos.

Entrando más en detalle en la asignación de eventos, podemos distinguir tres aspectos fundamentales del layout: acceder al contenedor más externo, decidir las características del listener de teclado por defecto, e implementar los métodos que permiten cambiar el foco.

El contenedor más externo de cualquier implementación va a ser la ventana de aplicación. Como necesitamos tener dicho contenedor para poder asignar el listener de teclado, tenemos que ser capaces de acceder a él a partir del contenedor sobre el que se aplica el layout. Este último nos va a permitir ir ascendiendo por la jerarquía de contenedores hasta que ya no podamos ascender más. Al producirse esta situación, se puede asegurar que hemos obtenido la ventana de aplicación. Para realizar esta tarea, existe un método en el layout llamado `TomarVentana()`, el cual se invoca antes de finalizar el método `add()`.
4. Descripción informática

Al crear el listener que asigna por defecto el layout, se va a limitar el uso del teclado a cuatro teclas, que corresponden al movimiento del cursor. Dependiendo de la tecla pulsada se enfocará el elemento que está encima, debajo, etc.

Finalmente nos queda explicar los métodos que permiten cambiar el foco de los elementos, que coinciden con los cuatro movimientos que se pueden hacer:

- **keyLeft()** y **keyRight()**: estos métodos se encargan de desplazar el foco hacia la izquierda y derecha respectivamente. La manera de realizar esto de una forma óptima es sabiendo en cada momento el identificador del elemento que está seleccionado, el cual coincide con la posición del elemento en el vector de componentes interno del layout. Al desplazar el foco hacia la izquierda, es como recorrer el vector en dicho sentido, por lo tanto el valor de los identificadores (posición del vector) debe ir disminuyendo. En el caso de que lleguemos al primer elemento del vector, habrá que empezar por el final del mismo. En el caso de desplazar el foco hacia la derecha, el funcionamiento es similar. La única diferencia que existe en este caso es que el número de los identificadores debe ir aumentando.

- **keyUp()** y **keyDown**: estos métodos se encargan de desplazar el foco arriba y abajo respectivamente. En ambos casos no podemos asignar un valor constante de desplazamiento en el vector, ya que las filas pueden no tener el mismo número de elementos. En el caso del **keyUp()**, una posible solución sería seleccionar el elemento de la fila anterior que esté situado en esa misma columna. Esto mismo sería válido para el **keyDown()**, pero con los elementos de la fila siguiente. Desde el punto de vista de la implementación esto sería lo más sencillo, ya que cada componente almacena la fila y columna en la que está, por lo que sería simplemente una búsqueda en el vector de componentes. Sin embargo, puede darse el caso de que el nuevo elemento a seleccionar, no sea el más adecuado, ya que visualmente puede estar muy distante (véase la figura 4.14).

El hecho de seleccionar el elemento de la fila anterior que esté más cercano visualmente, implica una mayor complejidad en el algoritmo de selección. En
4.4. Especificación detallada de RZ-Layout

Figura 4.14: Esquema R-Zoom para el caso del keyUp: El elemento seleccionado es el 5, el de la misma fila es el 2, pero el más cercano visualmente es el 1.

primero lugar, es necesario obtener todos los elementos de la fila anterior para luego realizar los cálculos oportunos. Para cada elemento de la fila anterior vamos a calcular el valor de la distancia que hay entre su centro y el centro del elemento seleccionado. Una vez que tenemos las distancias calculadas, simplemente queda elegir como nuevo foco al elemento cuya distancia sea la mínima. De esta manera, se logra que al seleccionar un nuevo elemento, el foco no se desplace demasiado desde su punto de origen.

El funcionamiento del keyUp() es semejante que el del keyDown(). Se diferencian en que el keyDown() toma la fila siguiente para el cálculo de las distancias mínimas.

Para todos estos métodos, una vez que se sabe qué elemento debe ser seleccionado, se invoca al método setSeleccionado(id), el cual se encargará de realizar los cambios oportunos. Con esto se concluye la asignación automática de eventos de teclado por parte del layout.

Como se ha expuesto anteriormente, otra posibilidad, es hacer que el usuario del layout pueda implementarse su propio listener. El único objetivo que tiene esto es que se puedan variar las teclas encargadas de desplazar el foco. Para permitir esto, lo que se ha hecho es que los métodos que controlan la selección del foco sean públicos. Además, se ha añadido una opción para desactivar la asignación
4. Descripción informática

de listener de teclado. Por lo tanto, lo único que tiene que hacer el usuario es crear un nuevo listener que invoque a los métodos de cambio de selección con los valores de teclas que crea oportuno.

4.5. Jerarquía de clases

En este apartado se hará una exposición de las clases que se han utilizado en el desarrollo del proyecto, así como la relación que existe entre ellas. Para ello nos apoyaremos en los diagramas de clases.

Todos los layouts proporcionados por el lenguaje implementan la interfaz LayoutManager, por lo tanto, para crear un layout nuevo es necesario realizar una nueva implementación de dicha interfaz. En la figura 4.15 se muestra el diagrama de clases de integración del RZ-Layout.

Figura 4.15: Diagrama de integración del RZ-Layout en la jerarquía de los layouts de Java

Para este desarrollo ha sido necesario crear una serie de clases que se comu-
4.5. Jerarquía de clases

tamiento deseado. La figura 4.16 refleja las clases que se utilizan en la implementación del **RZ-Layout**.

![Diagrama de clases que implementan el RZ-Layout](image)

**Figura 4.16: Diagrama de clases que implementan el RZ-Layout**

Finalmente, se muestra toda la jerarquía de clases que intervendrían en un ejemplo de uso del **RZ-Layout**. En este ejemplo se da por supuesto que la tarea de asignar los eventos de teclado es responsabilidad del layout. Todo esto se puede ver en el diagrama de clases de la figura 4.17.

En el diagrama de la figura 4.17 se puede ver que el usuario simplemente debe instanciar el **RZ-Layout** y por otro lado, desarrollar una implementación particular de la interfaz propuesta. Así, se demuestra que la utilización del **RZ-Layout** es inmediata, de tal forma que el usuario debe centrar los esfuerzos en desarrollar la implementación del **RZ-Component**.
Figura 4.17: Diagrama de clases de utilización del RZ-Layout
4.6. Generalización y parametrización de RZ-Layout

RZ-Layout se creó inicialmente para implementar la funcionalidad de R-Zoom, haciendo posible su utilización como cualquier otro layout. En este apartado estudiamos cómo extender el funcionamiento de RZ-Layout para realizar otro tipo de visualizaciones, realizando los mínimos cambios posibles. Cuando un elemento es seleccionado, el componente que recibe el evento indica al layout que lo han seleccionado pasándole su valor de hashCode. Este valor va a servir al layout para discriminar qué objeto RZComponentInformado ha sido seleccionado gracias a la asociación (hashCode, RZComponentInformado). En consecuencia, el layout indica a dicho componente que debe cambiar su apariencia, pudiendo darse dos situaciones:

- Hay un elemento seleccionado: en este caso hay que averiguar de qué elemento se trata para restaurarlo a su visualización normal, redistribuir los elementos a su posición original, luego proceder a cambiar el elemento que se acaba de seleccionar y redistribuir los otros elementos que no han cambiado su apariencia.
- No hay ningún elemento seleccionado: en este caso simplemente hay que cambiar la apariencia del elemento seleccionado y redistribuir el resto de elementos que se vean afectados.

Como la situación más genérica es aquella en la que hay un elemento seleccionado, vamos a centrarnos en ella. La generalización se basa en que para cambiar la visualización, simplemente hay que cambiar los métodos de redistribución cuando se selecciona un componente. Esto quiere decir, que la forma de deseleccionar un elemento es siempre la misma. Cuando se realice una nueva selección habrá que restaurar el elemento previamente seleccionado. Además, cuando se restaura un elemento, por un momento se vuelve a la visualización normal, es decir, aquella en la que no hay ningún otro elemento seleccionado. Teniendo en cuenta la idea principal de la generalización, todo el proceso de restaurar la selección es común.
4. Descripción informática

en todos los casos, independientemente de la redistribución que se hizo para seleccionar el elemento. Esto significa que si se cambia la forma en la que se distribuyen los elementos cuando uno es seleccionado, al quitar la selección, el estado al que se vuelve va a ser siempre el mismo. Vamos a plantear una serie de casos para ver si se cumple, en ellos, la idea básica de la generalización. De esta forma se podrá ver si es necesario o no la implementación de varios métodos de quitar el foco.

Supongamos que deseamos realizar dos recolocaciones distintas, llamadas RZ-Centrado y Centrar-Fila. Los esquemas de recolocación se muestran en la figura 4.18.

Figura 4.18: Esquema de recolocación de los distintos métodos

A continuación se hace una descripción de estas recolocaciones:

- RZ-Centrado: en este caso al seleccionar un elemento, al igual que sucede con R-Zoom, la fila se parte en dos a partir del elemento seleccionado. Sin embargo, dicho elemento debe mostrarse en el centro de la partición, es decir, los elementos a la izquierda de la selección no cambian, el elemento seleccionado se desplaza en altura y se sitúa en el centro del contenedor y por último los elementos a su derecha deben situarse justo debajo de la vista detallada. Si estudiamos en detalle esta situación, vemos claramente bien que la diferencia que existe con R-Zoom no es tan grande. La diferencia principal está a la hora de asignar el foco. A la hora de quitarlo, simplemente hay que volver a redistribuir los elementos como estaban en un principio.
4.6. Generalización y parametrización de RZ-Layout

- Centrar-Fila: en este caso al seleccionar un elemento, la fila a la que dicho elemento pertenece se desplaza completamente hacia abajo y este elemento se visualiza en su forma ampliada centrado en el hueco. Una diferencia con R-Zoom es que la miniatura no debe desaparecer del contenedor, sino que debe cambiar su color. De esta forma somos capaces de señalar qué miniatura es la que se está visualizando de forma detallada en cada caso. El estudio de este caso demuestra que la forma de conseguir que los elementos vuelvan a su colocación inicial es diferente que la del caso anterior. Esto se debe a que en este caso vamos a tener dos elementos, que son el mismo y de alguna forma hay que eliminar uno de ellos a la hora de volver a recolar los elementos.

En conclusión, la implementación de los métodos para devolver a los elementos a su posición original, va a depender fuertemente de la recolocación que se realice a la hora de asignar el foco. Para poder llegar a un consenso, se ha decidido que cada vez que se haga una recolocación nueva, se implemente el método de quitar el foco.

Por otro lado, tenemos que decidir si es necesario o no la implementación de varios métodos para realizar recolocaciones distintas a la hora de poner el foco. Atendiendo a los ejemplos que se han expuesto anteriormente, en este caso no existe la menor duda de que va a ser necesario implementar nuevos métodos que reubiquen los elementos atendiendo a las características de cada recolocación.

Una vez visto las características de la recolocación, hay que ver dónde realizar los cambios oportunos. En este proceso los métodos que se han visto involucrados son: setSeleccionado, recolarPonerFoco y recolarQuitarFoco. El método setSeleccionado() se encarga de quitar y poner el foco, para ello invoca a los métodos recolarQuitarFoco() y recolarPonerFoco() respectivamente. El método recolarPonerFoco() se encarga de redistribuir los elementos cuando uno tiene el foco, mientras que el método recolarQuitarFoco() se encarga de recolar los elementos cuando se quita el foco.

Como se puede ver, el problema para poder extender el layout se encuentra en estos tres métodos. A continuación se van a exponer una serie de cambios necesa-
4. Descripción informática

rios para poder extender el layout.

El primer cambio que se ha realizado es renombrar los métodos `recolocarPonerFoco` y `recolocarQuitarFoco` y cambiarlos para cada caso en particular, de la forma `recolocarPonerFocoXXXXX` y su análogo para quitar el foco, donde la `XXXXX`, se puede sustituir por el nombre que se desee. En el caso de R-Zoom se ha renombrado dicho método llamándose ahora `recolocarPonerFocoRZoom`. Además, en el método `setSeleccionado` se ha introducido un `switch` que permite elegir qué método hay que invocar para quitar y poner foco en cada caso. Finalmente, ha sido necesario sobrecargar el constructor del `RZ-Layout`, de forma que se le ha añadido un parámetro más que indica qué recolocación utilizar. Este valor será el que se utilice en el método `setSeleccionado` para invocar al método correspondiente. Por defecto, si no se especifica ningún valor para el parámetro, se utilizará la recolocación R-Zoom.

4.6.1. Recolocación opcional 1: RZ-Centrado

La característica fundamental de esta recolocación, es que cuando se selecciona un elemento se debe partir la fila, de tal forma que los elementos que se encuentran a la izquierda del elemento seleccionado, no alteran su colocación. Sin embargo, los elementos que se encuentran a la derecha, deben situarse, al igual que sucede con R-Zoom, debajo de la vista detallada. En cuanto al elemento que tiene el foco, éste debe mostrarse en el centro del contenedor, pero debe estar desplazado en altura, en concreto, debajo de los elementos que están a su izquierda. Véase figura 4.18.

Esta recolocación es muy semejante a la filosofía seguida por R-Zoom, ya que sigue el modelo de Foco + Contexto. La diferencia fundamental de esta recolocación con R-Zoom, es que la coordenada X del foco en este caso se ve afectada en todos los casos, ya que debe situarse en el centro del contenedor. Sin embargo, hay que recordar que en R-Zoom esto sólo sucede cuando el elemento no cabe en el contenedor o no cabe desde la posición en la que se encuentra.

Los métodos para poner y quitar el foco son `recolocarPonerFocoRZCentrado()` y `recolocarQuitarFocoRZCentrado()`, respectivamente. Para asignar el foco, lo funda-
4.6. Generalización y parametrización de RZ-Layout

mental es calcular el desplazamiento en altura de los elementos. Para ello hay que obtener el valor de altura máximo de las vistas detalladas de los elementos de la fila. Una vez que tenemos dicho valor, la altura a la que debe situarse el elemento seleccionado viene determinada por la expresión \( alturaFinal = alturaActual + maxAlturaDetalla + separacionAltura \). Por otro lado, el valor de la nueva coordenada X se obtiene con la siguiente expresión \( posX = anchoContenedor/2 - anchoElemento \). Como ya tenemos situado el elemento enfocado, sólo nos queda calcular la nueva posición de los elementos que están a su derecha. Estos elementos se van a ver desplazados simplemente en altura ya que el valor de su coordenada X no debe cambiar. Su nueva altura viene determinada por la expresión \( alturaFinal = AlturaActual + maxAlturaDetalla + alturaFoco + 2 \times separacionAltura \). En cuanto al método recolocarQuitarFocoRZCentrado(), su funcionamiento es igual al método recolocarQuitarFocoRZoom(). Véase 4.4. En la figura 4.19 se puede ver el funcionamiento de este tipo de recolocación.

Figura 4.19: Funcionamiento RZ-Centrado. Como se puede ver se ha seleccionado el segundo elemento de la primera fila, lo que ha provocado que se dividá la fila en dos y se sitúe por un lado el elemento que tiene el foco en el centro del contenedor y el resto de elementos a su derecha han conservado su posición en X pero se han situado debajo del elemento seleccionado.
4. Descripción informática

4.6.2. Recolocación opcional 2: Centrar-Fila

Al igual que sucede con la extensión anterior, la característica fundamental en este caso es que el elemento seleccionado se sitúa en el centro del contenedor. Sin embargo, existen algunas diferencias entre ambas recolocaciones. El aspecto visual que más difiere, es que al asignar el foco a un elemento se van a conservar en el contenedor tanto la vista en miniatura como la vista detallada. Véase la figura 4.18. Esta recolocación, sigue el modelo Foco + Contexto, pero difiere bastante del funcionamiento de R-Zoom.

Para permitir el funcionamiento correcto de esta extensión, existe una condición que debe cumplirse en la implementación del RZ-Component, y es que a la hora de asignar y quitar el foco deben crearse objetos nuevos, es decir, es obligatorio realizar un new, aunque la referencia se asigne al mismo elemento. El objetivo que tiene esto, es poder conservar los elementos que están en el panel interno del layout. En esta recolocación, el layout va a tener un comportamiento especial, ya que se encargará de cambiar el color de la miniatura que hace referencia al elemento que se muestra en detalle.

Otra diferencia con la recolocación anterior, es que en este caso la fila en la que está el elemento seleccionado no se va a dividir por ningún punto, sino que se va a desplazar completamente hacia abajo y va a ser el elemento que tiene el foco el que conservará siempre su coordenada Y. Los métodos encargados de poner y quitar el foco son recolocarPonerFocoCentrarFila() y recolocarQuitarFocoCentrarFila() respectivamente.

Las expresiones más relevantes del método recolocarPonerFocoCentrarFila() son las siguientes. El elemento que tiene el foco va a conservar su coordenada Y, sin embargo hay que recalcular su coordenada X a través de la expresión \(posix = \frac{anchoContenedor}{2} - anchoElemento\). Por otro lado, la expresión que calcula la nueva altura de los elementos de la fila es \(alturaFinal = alturaInicial + maximoAlturaFoco + separacion_y\).

Una vez que hemos comentado las acciones que tiene que realizar el método
4.6. Generalización y parametrización de RZ-Layout

de poner foco, aún nos queda comentar la funcionalidad que debe implementar el método de quitar el foco. En este caso, dicho método se va a encargar de realizar las mismas tareas que realizaba el `recolocarQuitarFocoRZoom`, con el añadido de que ahora debe eliminar uno de los dos componentes que tenemos. Como norma general, en el layout se van a eliminar ambos componentes y se le solicitará al `RZ-Component` que devuelva un nuevo elemento sin foco.

En la figura 4.20 vemos el funcionamiento de este tipo de recolocación.

Figura 4.20: En la figura de la izquierda se puede ver la carga inicial, en la que no se ha seleccionado ningún elemento. A la derecha vemos cómo se ha seleccionado el tercer elemento de la primera fila. Como resultado, se marca dicho elemento y se muestra su vista detallada

4.6.3. Recolocación opcional 3: vista Global + Detalle

La característica fundamental de este tipo de recolocación es que se va a reservar una zona común para visualizar todas las vistas detalladas de los elementos. Además, al igual que la recolocación anterior, se va a señalar en todo momento qué elemento se está visualizando. El esquema de funcionamiento de esta recolocación se muestra en la figura 4.21.
4. Descripción informática

![Figura 4.21: Se encuentra seleccionado el tercer elemento de la segunda fila](image)

Esta recolocación ya no sigue el modelo de *Foco + Contexto*, sino el modelo *Global + Detalle*. La diferencia de este tipo de recolocación es que no se va a guardar el contexto del elemento seleccionado. Hasta ahora, hemos visto que en todas las recolocaciones anteriores el elemento seleccionado se mostraba en su forma detallada, conservando en la medida de lo posible su posición en el entorno que le rodea. Sin embargo, esto no se va a cumplir en esta extensión, ya que se ha definido un área en la que los elementos se van a mostrar en su forma detallada. Los nombres asignados a estos métodos han sido *recolocarPonerFocoCentrar* y *recolocarPonerQuitarCentrar* respectivamente.

El principal problema que se nos plantea para la realización de esta extensión, es decidir dónde se va a situar el elemento seleccionado. Según la encapsulación de contenedores de la figura 4.5, vemos que tenemos un panel interno en el layout donde se realizan todos los cambios de recolocación del layout. Sin embargo, en este caso hay que mostrar el elemento seleccionado del panel en un contenedor externo. Otro problema añadido con el que nos encontramos, es que el tamaño del contenedor del layout va a coincidir con el tamaño del ScrollPane que muestra las barras de desplazamiento, por lo tanto, no podemos añadir el elemento a dicho contenedor sin variar la colocación del ScrollPane y por lo tanto del panel interno. La solución por la que se ha optado ha sido por modificar el método *layoutContainer*. El único cambio realizado, es la colocación y el tamaño del ScrollPane asociado al layout. En cuanto al panel interno, no ha sido necesario modificar ninguno de sus parámetros, ya que al variar la posición del ScrollPane, como el panel se encuentra en
4.6. Generalización y parametrización de RZ-Layout

su interior, también se va a desplazar. A continuación se muestra un esquema del algoritmo seguido para colocación del ScrollPane.

si alturaMaximaFoco < AlturaContenedor
{
    alturaMaxMiniatura=buscarMaximo(vector)
    si alturaScrollPanel-alturaMaximaFoco < alturaMaxMiniatura
    {
        ScrollPanel.colocar(0,AlturaContenedor-maxMiniatura ,anchura,maxMiniatura)
    }
    sino
    {
        ScrollPanel.colocar(0,alturaMaximaFoco,anchura, altura-alturaMaximaFoco)
    }
}
sino
{
    ScrollPanel.colocar(0,AlturaContenedor/2,anchura, altura-AlturaContenedor/2)
}

Como queremos que en el espacio común en el que se muestran los elementos seleccionados pueda situarse sin ningún problema cualquiera de ellos, debemos realizar una búsqueda en el vector interno que nos devuelva el elemento que tiene mayor altura, esto se representa con la variable alturaMaximaFoco. Una vez que tenemos el valor máximo de altura, hay que comprobar si dicho valor es mayor o menor que la altura del contenedor. Si resulta que el valor es mayor o igual que la altura del contenedor, la nueva situación del ScrollPane es la que figura en el algoritmo. En este caso para poder situar los elementos que superen AlturaContenedor/2 va a ser necesario aplicar una reducción. En el caso contrario, es decir, que la altura máxima sea menor que la del contenedor, habrá que calcular también el tamaño
máximo de las vistas en miniatura, esto se representa en el algoritmo con la variable \textit{alturaMaximaMiniatura}. Al tener ambos valores, ya podemos fijar la altura y la posición del ScrollPane, sin embargo, aún queda el caso de que la altura de las vistas detalladas sea bastante grande, pero no lo suficiente como para que superen la altura del contenedor. En este caso, si dejáramos directamente este espacio para representar los elementos enfocados podría ocurrir que el tamaño del ScrollPane disminuyera tanto que fuera imposible navegar por él. Por lo tanto, para garantizar que siempre se pueda navegar por el ScrollPane y así poder ver las miniaturas, va ser necesario introducir una condición. Para ello lo que se hace es comparar la altura que tendría el ScrollPane con la altura máxima de las miniaturas. Si resulta que la nueva altura del ScrollPane va a ser menor que el máximo de las miniaturas, se obligará al ScrollPane a tener como mínimo esta última altura. Esto va a obligar a que se tengan que reducir algunas vistas detalladas. Sin embargo, si resulta que el nuevo tamaño del ScrollPane es mayor que la altura máxima de las miniaturas, se realizarán los cambios sin tener en cuenta nada más. En este último caso, si no cambian las dimensiones del contenedor, nunca va a ser necesario aplicar una reducción a una vista detallada.

Una vez que hemos explicado los cambios que sufre el \textit{layoutContainer}, sólo quedan ver las características más relevantes del método de recolocación. Para recolocar un elemento, al igual que antes, se debe cumplir la condición en la implementación del \textit{RZ-Component}, es decir, al asignar y quitar el foco deben crearse objetos nuevos. Esto es necesario exactamente por los mismos motivos que en el caso anterior. La única diferencia que existe en este caso a la hora de asignar el foco es que el elemento enfocado no se va a añadir al panel del layout, sino que se va añadir directamente al contenedor al que se le ha asignado el layout en el espacio resultante de desplazar el ScrollPane. Como al poner el foco, el elemento se añade directamente al contenedor, necesitamos tener un método concreto para quitar el foco. Este método simplemente debe encargarse de eliminar el elemento seleccionado del contenedor y realizar los cambios de color en la miniatura seleccionada. En la figura 4.22 se muestra un ejemplo de funcionamiento.
4.7. Problemas durante la implementación

Figura 4.22: Visualización vista Global + Detalla. En la figura de la izquierda se muestra la carga inicial. Se puede observar que el ScrollPane se encuentra desplazado en altura. En la figura de la derecha se ha seleccionado el quinto elemento de la segunda fila.

4.7. Problemas durante la implementación

Los principales problemas que hemos encontrado durante el desarrollo del proyecto se deben principalmente a la relación existente entre el layout y el RZ-Component. Estos problemas fueron: identificación de los elementos, conversión de Component a RZ-ComponentInformado y señalar elemento seleccionado.

4.7.1. Identificación de los elementos

Este aspecto se refiere a la necesidad de obtener un valor que sirva para poder distinguir fácilmente un elemento de otro. Hasta ahora hemos comentado que el layout utiliza el valor de los hashCode de los elementos que devuelve el método...
**4. Descripción informática**

`getElement()` de la interfaz `RZ-Component`. Una primera solución que se planteó para identificar a los elementos, consistía en utilizar un campo de la interfaz `RZ-Component` para asignar un nombre a cada elemento. Sin embargo, esta solución planteaba dos problemas principalmente. Por un lado, se dejaba como tarea del usuario que asignara de forma manual los nombres a cada uno de los componentes. Esto tenía que realizarse utilizando el método `getElement()` que devuelve un elemento de tipo `Component`. A su vez, este tipo de objeto proporciona un método denominado `setName()`, el cual debía utilizar el usuario para asignar el nombre al componente. Claramente esta solución implica una serie de limitaciones:

- Es responsabilidad del usuario asignar correctamente los nombres a los elementos, ya que debe asegurarse que dos elementos no pueden tener el mismo nombre.
- Traslada un aspecto que debería ser interno del layout, hacia el usuario del mismo, afectando a la transparencia.
- Requiere que se implementen métodos que mantengan actualizados los nombres en el layout.

Por otro lado, el identificador que se utilizaba para distinguir un elemento de otro, era de tipo `String`. Como este identificador se va utilizar luego en el layout como clave en una tabla hash, a la hora de realizar una búsqueda el coste es más elevado que si fuera simplemente un valor entero.

Debido a estos problemas, se empezó a buscar una solución que evitara trasladar el problema al usuario permitiendo que se pudiera cambiar el identificador. El resultado fue la utilización del valor del `hashCode`. La utilización de este valor implicó que:

- Se eliminara el campo para almacenar el nombre de cada elemento.
- Se quitara responsabilidad al usuario del layout, ya que en este caso no debe preocuparse de asignar los nombres a los componentes.
4.7. Problemas durante la implementación

- Se ocultará la forma en la que se identifican los elementos en el funcionamiento del layout.

- Se obtuviera un valor entero para indexar en las tablas hash.

Cuando se realiza un new dentro de la implementación del RZ-Component debe comunicarse el nuevo valor del hashCode al layout. Esto puede parecer un inconveniente, pero como los únicos lugares en los que debería crearse un objeto nuevo es en el constructor y en los métodos setFoco y RemoveFoco, no se requiere que el programador sea consciente de ello, ya que en el método setSeleccionado del layout, tiene en cuenta esta situación.

4.7.2. Conversión de Component a RZ-ComponentInformado

Este problema surge por la decisión de añadir los eventos de ratón a objetos de la clase Component, por lo tanto, al producirse un evento mouseclick se recupera un objeto de dicha clase. Para obtener un elemento de la clase RZ-ComponentInformado no tiene sentido una conversión implícita ni explícita entre estos dos tipos de datos. Por esta razón fue necesario implementar una relación que permitiera obtener un RZ-ComponentInformado a partir de un objeto de la clase Component. En particular se optó por realizar una relación entre el hashCode del componente que devuelve el método getElement con el RZ-ComponentInformado que se crea al ejecutar el método add(). Esta asociación se guardaría en la tabla Hash del layout. Sin embargo, la asociación sólo sirve para consulta interna del propio layout, pues el usuario no va a tener acceso a ella.

4.7.3. Señalar el elemento seleccionado

Este problema surge cuando se plantea la extensión del layout y empiezan a surgir tipos de recolocaciones que mantienen la miniatura, la cual debe aparecer resaltada, mientras se visualiza la vista detallada. El problema surge cuando nos
4. Descripción informática

planteamos si debe ser el layout o el programador el que debe señalar el elemento. Se ha optado por una solución mixta, es decir, el layout sólo va a resaltar automáticamente los elementos que pertenezcan a determinadas clases. Sin embargo, el usuario va a ser el responsable de resaltar el resto de elementos cuando implemente el método `setFoco`. El objetivo que tiene esta solución es permitir una mayor flexibilidad a la hora de elegir cómo se quiere que se realice la selección, por ejemplo, de esta forma se va a permitir al usuario elegir el tipo de borde a utilizar.
4.8. Pruebas y evaluación de RZ-Layout

Tras finalizar el desarrollo del RZ-Layout, hemos comprobado que se han cumplido correctamente todos los requisitos descritos al comienzo del proyecto.

En esta sección vamos a evaluar el uso genérico del RZ-Layout desarrollando varias interfaces. En concreto, se han desarrollado tres interfaces que permiten mostrar el funcionamiento del layout, así como demostrar las posibles implementaciones que puede tener la interfaz RZ-Component. Hay que tener en cuenta que aunque las recolocaciones que se utilizan en los ejemplos son R-Zoom y vista Global + Detalle, es posible cambiar cualquiera de ellas para utilizar la recolocación que se desee, simplemente hay que indicar al layout qué recolocación queremos utilizar. Para cada una de las implementaciones, hemos estudiado los tiempos dedicados a cada tarea para poder comparar la facilidad de uso del layout y del RZ-Component en cada caso.

Por último, hemos realizado una interfaz que unifica los tres ejemplos desarrollados y que permite ejecutarlos con cualquier tipo de las visualizaciones disponibles. Véase la figura 4.23.

Figura 4.23: Unificación de Ejemplos

4.8.1. Ejemplo de utilización 1: visualización simple de imágenes

Esta visualización consiste en una aplicación en la que el RZ-Layout se asigna directamente a la ventana de aplicación. De esta forma se evita que el RZ-Layout
4. Descripción informática

trabaje en conjunto con otros elementos a su alrededor. Además, para obtener las miniaturas de los elementos, se van a utilizar los métodos que proporciona el lenguaje para obtener imágenes reducidas. La aplicación va a consistir en la visualización de un directorio cuyo contenido son imágenes, en concreto archivos gif y jpg. En esta aplicación se va a permitir realizar diferentes tareas como: cambio de la separación de los elementos, cambio de la separación de las filas, cambiar el factor de reducción y cambiar el directorio de imágenes a mostrar.

La utilización del layout se limita simplemente a crear el layout con los parámetros deseados, asignarlo al contenedor y a añadir a este último componentes. Todo esto se muestra en el siguiente pseudocódigo:

```java
milayout=nuevo RZ-Layout(...) 
ventana.layout=milayout
ventana.add(nuevo Implementacion_RZ_Component)
```

La utilización del layout es bastante sencilla, y además su funcionamiento es siempre el mismo. Sin embargo, aún queda un aspecto fundamental, la implementación del RZ-Component. En este ejemplo, las características que debe tener el RZ-Component son:

- **Construcción**: a la hora de crear un componente es necesario proporcionarle al constructor un objeto de la clase `Image`. En su interior se va a guardar dicho objeto y además se realizará una copia que será la que el layout deberá reducir cuando se aplique el método `obtenerMiniatura()`.

- **Miniatura**: para obtener la miniatura se va a utilizar la copia generada cuando se construyó el componente. Para obtener la miniatura se utiliza un método de los que proporciona la clase `Image`.

- **Elemento sin Foco**: el elemento que obtendrá el layout cuando el componente no esté seleccionado será la imagen en miniatura.
4.8. Pruebas y evaluación de RZ-Layout

- Elemento con Foco: el elemento que obtendrá el layout cuando el componente esté seleccionado será la imagen original, es decir, aquella que no se ha reducido.

- Reducción: cuando se selecciona un elemento éste devuelve la imagen original, pero puede darse el caso de que dicha imagen no pueda visualizarse en el contenedor, por lo que habrá que reducir el elemento. En esta implementación se ha optado por realizar una reducción en anchura y en altura de la imagen.

Como podemos observar la implementación del *RZ-Component* depende de lo que se quiera visualizar. Hemos realizado un estudio para ver cuánto tiempo dedicamos a la implementación del componente y cuánto a la utilización correcta del layout. Los resultados se muestran en la tabla 4.1.

<table>
<thead>
<tr>
<th></th>
<th>Tiempo Total</th>
<th>2 h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiempo dedicado al RZ-Component</td>
<td>37 min</td>
<td></td>
</tr>
<tr>
<td>Tiempo dedicado a la creación del Layout</td>
<td>2 min</td>
<td></td>
</tr>
<tr>
<td>Esfuerzo dedicado a la impl. del componente</td>
<td>30%(37min/120min)</td>
<td></td>
</tr>
<tr>
<td>Esfuerzo dedicado a la creación del layout</td>
<td>1,7%(2min/120min)</td>
<td></td>
</tr>
<tr>
<td>Esfuerzo dedicado a otras tareas</td>
<td>67,5%(81min/120min)</td>
<td></td>
</tr>
</tbody>
</table>

Cuadro 4.1: Tabla de tiempos para el desarrollo de la visualización simple de imágenes. Otras tareas se refiere al tiempo invertido en el diseño de la aplicación, asignación de eventos, creación de menús, etc.

Como se puede ver en la tabla, el tiempo dedicado a la utilización del layout es prácticamente insignificante comparado con el total. También podemos ver que una fracción muy importante de tiempo se dedica a la implementación del componente y además dicha fracción es mayor que la dedicada a la utilización del layout.

En la figura 4.24 se muestra esta aplicación.
Figura 4.24: Visualización simple de imágenes
4.8. Pruebas y evaluación de RZ-Layout

4.8.2. Ejemplo de utilización 2: visualización de imágenes

La visualización de imágenes difiere de la visualización simple únicamente en dos aspectos: asignación de RZ-Layout a un contenedor que no es la ventana de aplicación e implementación del RZ-Component. Esta interfaz podrá cargar un directorio de imágenes. Luego, a través del menú podremos cambiar la separación de los elementos, el factor de reducción y también podremos indicar el directorio a importar.

En este caso se ha realizado una distribución especial de la ventana de aplicación asignándole el GridBagLayout. A la izquierda se va a colocar un contenedor que será al que se le asigne el RZ-Layout, y a la derecha se colocará otro contenedor que será el que contenga cierta información sobre los componentes seleccionados. En la figura 4.25 se muestra un esquema de la distribución realizada.

![Esquema visualización compleja de imágenes](image)

**Figura 4.25: Esquema visualización compleja de imágenes**

Por otro lado, en cuanto a la implementación del RZ-Component podemos destacar:

- **Construcción**: al igual que en el caso anterior, al construir un RZ-Component, hay que proporcionar un elemento de la clase `Image`, del cual se generará una copia para luego generar la miniatura. Además hay que comunicarle cuál es el panel en el que se quiere mostrar la información del elemento seleccionado.
4. Descripción informática

- **Miniatura**: para obtener la miniatura se va a utilizar la copia generada cuando se construyó el componente. La diferencia en este caso es que para obtener la miniatura no se va a utilizar el método de la clase *Image* sino que se ha utilizado un método que obtiene miniaturas con una mejor calidad [8].

- **Elemento con Foco**: el elemento que obtendrá el layout cuando el componente esté seleccionado será la imagen a tamaño original. Además, se obtendrá la información sobre el nombre del fichero, resolución ampliada y resolución reducida, que se insertará en el panel que se le comunicó cuando se creó el componente.

- **Elemento sin Foco**: el elemento que obtendrá el layout cuando el componente no esté seleccionado será la imagen en miniatura, además se tendrá que eliminar del panel la información que se insertó cuando se seleccionó.

En la tabla 4.2 podemos observar los tiempos empleados para cada parte de la implementación.

<table>
<thead>
<tr>
<th>Tiempo Total</th>
<th>2:30 horas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiempo dedicado al RZ-Component</td>
<td>1:15 h</td>
</tr>
<tr>
<td>Tiempo dedicado a la creación del Layout</td>
<td>2 min</td>
</tr>
<tr>
<td>Esfuerzo dedicado a la impl. del componente</td>
<td>50%(75min/150min)</td>
</tr>
<tr>
<td>Esfuerzo dedicado a la creación del layout</td>
<td>1,3%(2min/150min)</td>
</tr>
<tr>
<td>Esfuerzo dedicado a otras tareas</td>
<td>49%(73min/150min)</td>
</tr>
</tbody>
</table>

Cuadro 4.2: Tabla de tiempos para el desarrollo de la visualización compleja de imágenes. Otras tareas se refiere al tiempo invertido en el diseño de la aplicación, asignación de eventos, creación de menús, etc

Como se puede ver en la tabla, el tiempo dedicado a la utilización del layout es similar. También podemos ver que una fracción muy importante de tiempo, en este caso mayor que el primer ejemplo, se dedica a la implementación del componente y además dicha fracción es mayor que la dedicada a la utilización del layout. Esto se debe a que la complejidad del RZ-Component ha aumentado. En este caso tenemos la necesidad de comunicar cierta información al panel que va a contener la información del elemento seleccionado. Además, otra tarea que ha requerido bastante
tiempo ha sido obtener la miniatura de cada imagen, ya que este caso no utilizamos los métodos de Java.

4.8.3. Ejemplo de utilización 3: visualización de páginas Web

En este ejemplo hemos decidido utilizar el layout para visualizar páginas web. La aplicación va a permitir realizar la carga de ciertas páginas web y visualizarlas utilizando un RZ-Component. El objetivo de esta visualización es demostrar la versatilidad del RZ-Component.

Básicamente lo que hará esta aplicación será conectarse a varias URLs ya proporcionadas y para cada una de ellas realizará un parseo. Este proceso nos permitirá obtener la versión reducida de las páginas, que consiste en otro documento con parte del documento original. En concreto, la versión reducida consiste en: título de página, color de fondo, primera imagen y cabeceras h1, h2 y h3.

En una primera versión de este ejemplo se utilizó el componente JEditorPanel[15] para visualizar las páginas web. Sin embargo, se desechó por tener ciertos problemas en la visualización de algunos elementos, a lo que se le añade la dificultad para parsear el documento html correctamente. En consecuencia, hemos probado una serie de componentes que permiten la visualización web. Finalmente hemos decidido utilizar el componente Cobra[11].

Con esta información ya somos capaces de crear un RZ-Component. Las características más relevantes en este caso de la implementación del componente son:

- Construcción: a la hora de construir el componente hay que transmitirle dos objetos de la clase Document. Uno de ellos contendrá el documento original, es decir, aquel que contiene la página original, el otro será el que hemos creado con las partes de nos interesaban a través del parseo, es decir, la versión reducida. Además se va a crear un ScrollPane que será donde se sitúen tanto la versión original como la reducida.
4. Descripción informática

- Miniatura: la obtención de la miniatura en este caso es distinto a los dos casos anteriores. Como el tamaño del ScrollPane es común para todos los RZ-Components, el tamaño de las miniaturas va a ser siempre el mismo.

- Elemento con Foco: el elemento que obtendrá el layout cuando el componente está seleccionado será el mismo que cuando no lo está, es decir, el ScrollPane pero con el tamaño cambiado. Para permitir que sólo sea necesario utilizar un componente, hay que crear un nuevo objeto y posteriormente cambiarle el tamaño.

- Elemento sin Foco: el elemento que obtendrá el layout cuando se quite el foco será de nuevo el ScrollPane al que se le ha cambiado otra vez el tamaño.

En la tabla 4.3 podemos observar los tiempos empleados para cada parte de la implementación:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiempo Total</td>
<td>7:30 h</td>
</tr>
<tr>
<td>Tiempo dedicado al RZ-Component</td>
<td>2:20 h</td>
</tr>
<tr>
<td>Tiempo dedicado a la creación del Layout</td>
<td>2 min</td>
</tr>
<tr>
<td>Esfuerzo dedicado a la impl. del componente</td>
<td>30 % (140 min / 450 min)</td>
</tr>
<tr>
<td>Esfuerzo dedicado a la creación del layout</td>
<td>0,4 % (2 min / 450 min)</td>
</tr>
<tr>
<td>Esfuerzo dedicado a otras tareas</td>
<td>68 % (308 min / 450 min)</td>
</tr>
</tbody>
</table>

Cuadro 4.3: Tabla de tiempos para el desarrollo de la visualización páginas web. Otras tareas se refiere al tiempo invertido en el diseño de la aplicación, asignación de eventos, creación de menús, etc

Como podemos ver, el tiempo dedicado a la utilización del layout es mínimo respecto al resto. Sin embargo, el tiempo para la implementación del componente es mayor. Pero sin duda el tiempo que destaca por encima de todos, es el tiempo total dedicado a la implementación del ejemplo. Este tiempo se debe a que, como ya se ha dicho, ha sido necesario probar distintos componentes que permitan la visualización web. En la imagen 4.26 se muestra una captura de pantalla de esta aplicación.
Figura 4.26: Visualización de páginas Web
Capítulo 5

Conclusiones y trabajos futuros

El objetivo de este proyecto era extraer el funcionamiento de la técnica R-Zoom de la aplicación en la que se encontraba para construir una API que permitiera su utilización fuera de ese contexto. Se requería que la API fuera genérica, por lo que tenía que permitir la visualización de cualquier tipo de componente. Además, la API debía ser ampliable, es decir, que se pudiera aumentar fácilmente el número de técnicas de visualización de información disponibles.

A lo largo del PFC hemos creado una API que satisface todos los requisitos aquí expuestos. Se ha creado un tipo de componente llamado *RZ-Component* que permite que la API sea genérica. Además, se ha creado un diseño que permite ampliar fácilmente el número de técnicas de visualización de información. Aparte de todo esto, se ha realizado una parametrización de la API que permite al usuario elegir algunas opciones de configuración como: factor de reducción, activación de eventos, etc. Aunque no fue un requisito inicial, se ha conseguido que la API tenga un rendimiento óptimo, de forma que los cambios que tienen lugar en la ejecución se hacen de forma eficiente.

Al concluir la implementación de la API se ha realizado una evaluación de la misma. El objetivo de esta evaluación era medir la facilidad de uso de la herramienta creada. Los resultados han sido satisfactorios, ya que la principal carga de traba-
jo (en el uso de la API) radica en la implementación del componente que se quiera visualizar.

Por último, como línea de trabajo futura identificamos la extensión del número de técnicas de visualización de información disponibles en la API y su generalización de manera semejante a lo que se ha hecho con el RZ-Component. Esta última tarea permitiría al propio usuario crear nuevas técnicas de visualización e incorporarlas a la API. Otra extensión que se plantea es mejorar las técnicas de asignación de foco, por ejemplo añadir transiciones animadas.
Glosario

**API**: Application Programming Interface. Grupo de rutinas (conformando una interfaz) que provee un sistema operativo, una aplicación o una biblioteca, que definen cómo invocar desde un programa un servicio que éstos prestan. En otras palabras, una API representa una interfaz de comunicación entre componentes software. El software que provee la funcionalidad descrita por una API se dice que es una implementación de la API. La API en sí misma es abstracta, en donde especifica una interfaz y no da detalles de implementación [2], 5

**Contenedor**: un contenedor pertenece a la clase Container la cual es una clase abstracta derivada de Component, que representa a cualquier componente que pueda contener otros componentes. Se trata, en esencia, de añadir a la clase Component la funcionalidad de adición, sustracción, recuperación, control y organización de otros componentes. Existen diferentes tipos de contenedores: Frame, Window, Dialog o Panel. La forma de añadir componentes a un contenedor es utilizando el método **add (Component)** de la clase Container [3], 8

**Foco**: se dice que un elemento tiene el foco cuando se selecciona para mostrarse con mayor detalle. La expresión asignar el foco equivale a seleccionar un elemento. Así, quitar el foco, es deseleccionar un elemento, 18

**LayoutManager**: se trata de una interfaz que proporciona Java para implementar nuevos layouts, 23
**Layout**: elemento del lenguaje Java que permite situar y distribuir los componentes añadidos a un contenedor de forma automática, 9

**Listener**: método o abstracción que sirve para conocer la existencia de eventos que tienen lugar en el sistema y actuar en consecuencia, 29

**R-Zoom**: se trata de una interfaz para la visualización del tipo *Foco+ Contexto*, 15
Bibliografía


