Framework de Software

Para el Desarrollo de Aplicaciones Gráficas en Java

TESIS DE MAGISTER

REINALDO ESPEJO MATTE

Mayo 2011

Tesis de Grado

Magíster en Ingeniería Informática

Escuela de Ingeniería Informática

Pontificia Universidad Católica de Valparaíso

Agradecimientos

Al profesor que me dio la idea de realizar este tema, y me apoyó durante los dos años de desarrollo del proyecto.

Al profesor que decidió apoyarme pese a que no era su tema favorito.

Resumen

El desarrollo de aplicaciones con representaciones gráficas de modelos de datos es complejo. El gran problema en los proyectos de desarrollo es el tener que comenzar desde cero con la construcción de las bases necesarias para lograr obtener dichas aplicaciones. En esta tesis se presenta el diseño y desarrollo de un framework de software para el desarrollo de aplicaciones gráficas en Java, el cual apunta a solucionar este problema. El trabajo consistió en un desarrollo preliminar de aplicaciones en Java con alto contenido gráfico. De estos desarrollos nacieron las primeras versiones del framework. Para esta tesis se formalizó el diseño de este framework, se finalizó su desarrollo, y se realizaron pruebas. Las pruebas sobre el framework ayudaron a comprobar su utilidad y facilidad de uso, comparado con la manera actual de realizar aplicaciones gráficas, utilizando Java Swing.

Palabras Claves

Framework de software, Java AWT, Java Swing, UML-F.

Abstract

The development of software applications with data model representations is a complex task. The big issue in software development projects is to have to start from scratch each time the necessary coding for these applications begins. This work presents the design and development of a software framework for Java graphical applications development, which objective is to solve this problem. The work started with the development of Java applications with high graphical content. The framework first versions were born from these applications. For this thesis the framework design was formalized, its development was completed, and test were performed on it. Tests over the framework helped to prove its utility and ease of use, when compared to the traditional way of creating graphical applications, using only Java Swing.

Keywords

Software framework, Java AWT, Java Swing, UML-F.

Tabla de Contenidos

Listado de Figuras

Listado de Tablas

Listado de Código Fuente

1. Introducción

Durante la última década, la industria computacional ha sido dominada por el consumidor. Esto ha llevado a un aumento en el énfasis sobre la estética de las interfaces [Guy07]. Cada vez más, los clientes exigen que las aplicaciones que adquieren tengan elementos gráficos atractivos y que faciliten las tareas más sofisticadas mediante representaciones visuales.

Java es uno de los lenguajes más populares hoy en día, y es el más utilizado para el desarrollo de aplicaciones [*Pop11*] [*Tio11*]. Una de sus grandes ventajas frente a los competidores, es que incorpora de manera correcta y completa el paradigma orientado a objetos. Además ofrece una amplia gama de bibliotecas que permiten el desarrollo de virtualmente cualquier tipo de aplicación. Esto, sumado a su gran potencial de extensibilidad para proyectos más complejos que requieren más de lo básico.

En cuanto al soporte para interfaces de usuario, Java proporciona un conjunto de bibliotecas (una API, *Application Programming Interface*), que, siendo propiamente extendidas, tienen el potencial de crear virtualmente cualquier tipo de gráficos e interfaces de usuario. Esta API, llamada *Swing*, es poderosa, pero es compleja de aprender y aplicar [Top10].

Al ir aumentando las exigencias del mercado en cuanto a la riqueza gráfica de las aplicaciones, aumenta también la complejidad del código fuente necesario para lograr estos resultados. Surge de aquí la necesidad de contar con algún *framework de software* (conjunto de bibliotecas y código reusable), que facilite el desarrollo de cada aplicación y se evite lo más posible el comenzar desde cero cada nuevo proyecto de software.

En general, los proyectos medianos y grandes de software, crean o implementan un framework para el desarrollo. Los frameworks creados, son siempre propios de la empresa o equipo, y pocas veces se comparten o se hacen públicos [Tul08].

La necesidad de contar con un framework de software que facilite la tarea de crear aplicaciones gráficas en Java puede estar siendo satisfecha en un nivel privado y por proyecto. No se ha encontrado hasta el día de hoy un framework de software de acceso público y que se base en la plataforma Java, independiente de implementaciones (proyectos específicos).

Es el objetivo de esta tesis el desarrollar un framework de software que satisfaga esta necesidad, aplicarlo, y validarlo.

1.1 Descripción del Problema

Java Swing, es una biblioteca de funcionalidades en Java, a partir de la cual se pueden crear interfaces de usuario visualmente ricas. Su gran desventaja es ser de "bajo nivel". En el contexto de este trabajo, bajo nivel significa que se requiere desarrollar bastante código para poder lograr estas aplicaciones ricas. El código necesario para crear aplicaciones Swing no es tarea fácil para principiantes en Java. Sólo desarrolladores en Java de nivel intermedio-avanzado logran el desarrollo de aplicaciones simples en Swing. La abstracción del código debe ser comprendida y manipulada por el desarrollador. Este debe ser capaz de ver en su código el resultado visual esperado. La razón de ello es que una aplicación visualmente rica inevitablemente estará compuesta de muchos elementos. Estos componentes deben ser creados uno a uno, y dispuestos de tal forma que se logre el resultado final esperado. El código fuente puede entonces llegar fácilmente a estar compuesto de varios miles de líneas.

Si bien este problema es trivial para los desarrolladores expertos, no lo es para los recién iniciados en Java, quienes terminan por utilizar herramientas visuales para la creación automática del código necesario para obtener esa GUI (*Graphical User Interface*). Estas herramientas en la práctica terminan siendo más un agravamiento del problema que una solución. Al estar generando automáticamente cientos de líneas de código, el desarrollador pierde el control sobre su código.

Pese a todo, el problema de "comenzar desde cero" cada vez que se quiera desarrollar una aplicación gráfica, se aplica a todos los niveles de desarrolladores. Este problema es conocido en la comunidad de Java, y está formalizado en la ***Java Specification Request 296***.

Formalización de la necesidad

La necesidad de contar con un framework de software que proporcione una capa superior a Swing está formalizada en Sun a través del *Java Specification Request 296*. El **JSR 296** tiene

como objetivo el buscar una solución, por parte de la comunidad o directamente de Sun. Planteado en el 2006, su descripción dice:

"La ausencia de un framework de aplicación básico ha hecho que el desarrollo de aplicaciones en Swing sea difícil, particularmente para nuevos desarrolladores. Una base estándar para aplicaciones reduciría el trabajo requerido para despegar aplicaciones y así mejorar su calidad." [JSR296]

Soluciones parciales al problema planteado

Para este trabajo se analizaron tres soluciones parciales para el problema planteado. La primera, el *Swing Application Framework*, es la solución directa de Sun al JSR 296. La segunda, es un Framework que apunta al desarrollo de aplicaciones con más contenido gráfico pero no en un contexto Swing, sino que en un contexto de la plataforma de Eclipse.

Swing Application Framework

El Swing Application Framework es un framework que estuvo en desarrollo hasta mediados del 2009 y pretendía ser parte de la futura versión 7 de Java [*App10*]. Este framework buscaba solucionar en parte el JSR 296. Sólo ataca el problema de la creación desde cero de aplicaciones en Swing, y de la persistencia de estados de la aplicación. Su clase principal se llama *Application*, la cual debe ser extendida por el usuario de forma de lograr su objetivo. Esta clase a su vez utiliza otra clase llamada *ApplicationContext*, la cual se encarga de los estados de la aplicación. La Figura 1.1 ilustra estas dos clases y sus principales componentes.

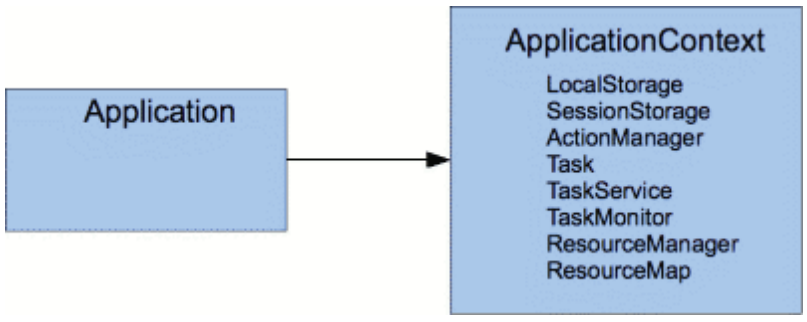

Figura 1.1: ***Las clases Application y ApplicationContext.***

Si bien este Framework no se compara directamente con el propuesto en este trabajo, sirve para mostrar el enfoque de solución al problema que sí es común, el de no tener una capa de abstracción más alta de lo que da Swing, para no siempre tener que comenzar de cero en el desarrollo de una aplicación.

En agosto del 2009 no se logró llegar a ciertos consensos entre sus desarrolladores miembros, por lo cual el proyecto no logró ser parte, al menos, de la versión de Java 7 del año 2010 [*Pot09*].

Graphical Editing Framework

El *Graphical Editing Framework* (**GEF**), o Framework de Edición Gráfica, permite a los usuarios desarrollar representaciones gráficas para modelos de datos. Es posible desarrollar editores gráficos ricos en funcionalidad usando GEF. Está creado sobre la plataforma Eclipse, lo que significa que en lugar de estar basado en *Swing* y *Java2D*, está basado en *SWT* y *Draw2D*, alternativas a estos dos respectivamente, lo que implica que utiliza clases y paquetes que no pertenecen a la API de Java. [IBM04]

GEF permite que el usuario construya editores gráficos para prácticamente cualquier modelo. Con estos editores, es posible realizar modificaciones simples al modelo, como cambiar propiedades de un elemento u operaciones complejas, como cambiar la estructura del modelo en diferentes maneras al mismo tiempo. Todas estas modificaciones al modelo pueden ser manejadas en un editor gráfico usando funciones comunes, como arrastrar, copiar/pegar, y acciones invocadas desde menús o barras de herramientas.

En resumen, el enfoque de GEF es proveer las herramientas para construir editores de los modelos de datos ya existentes del usuario que desea crear una manera gráfica de editarlos. Una de las desventajas de GEF, aparte de no ser Java estándar, es que su curva de aprendizaje es empinada. Esto debido a que GEF no fue creado para ser utilizado por usuarios medios. GEF es un sub-producto de la creación de la plataforma Eclipse, lo que prácticamente lo deja como un framework de “uso interno”, para quienes desarrollen sus aplicaciones sobre la plataforma Eclipse. En resumen, el requisito básico para entender o aprender GEF, es entender la plataforma Eclipse, lo que es lejano de la API de Java.

JavaFX

JavaFX es la nueva plataforma para la creación de aplicaciones de internet visualmente ricas. Es un nuevo lenguaje y entorno de ejecución introducido por Sun en el 2009-2010 [Top10].

Desde el punto de vista del desarrollador, JavaFX está integrado en el lenguaje de Java, por lo cual el código no representa mayores desafíos. Las bibliotecas de interfaces que JavaFX ofrece, son limpias y muy fáciles de utilizar.

JavaFX sigue en desarrollo, y si bien soluciona en parte el problema establecido, su enfoque es hacia las aplicaciones web, es decir, sus competencias directas son Adobe Flash y Microsoft Silverlight. El Framework propuesto en esta tesis es para aplicaciones de escritorio, JavaFX es para aplicaciones Web livianas y simples.

1.2 Objetivos

Objetivo General

- Diseñar y Desarrollar un Framework de Software que aumente la eficacia del desarrollo de aplicaciones gráficas en Java.

Objetivos Específicos

- Investigar y aprender las técnicas para la creación de gráficos interactivos en Java con Swing y Java2D
- Desarrollar bibliotecas y clases que en conjunto proporcionen las funcionalidades necesarias, determinadas por las aplicaciones del Framework.
- Evaluar, mediante pruebas con usuarios, la facilidad de uso del Framework, comparado con el uso de Swing.

1.3 Motivación

La investigación relativa al Framework nace el 2007, cuando el autor buscaba formas de optimizar el proceso de creación de representaciones gráficas en Java. El problema principal estaba radicado en que la manera convencional de crear interacción en Java era poco eficiente,

tanto en código como en tiempo computacional. Básicamente, esta manera consistía en realizar barridos a través de las figuras presentes en la pantalla para detectar la interacción (click de mouse) y "repintados" (repaint) de pantalla para cada movimiento, de forma de actualizar los contenidos de ésta y mostrar así el movimiento. Esta manera de realizar la interacción gráfica es la utilizada en el contexto de Java2D, y funciona bien sólo para situaciones con pocos elementos (menos de 100) en pantalla al mismo tiempo.

Se realizaron pruebas de rendimiento sobre estos casos, mostrando el alto consumo de CPU (más de 80%) y una alta utilización del recolector de basura de Java cuando se movían los elementos gráficos en pantalla. Esto llevó a tener que replantear la forma convencional de desarrollar la interacción de elementos gráficos.

Al principio el desarrollo se basó más que nada en una mejora local sólo para la aplicación que se estaba desarrollando en ese entonces. Sin embargo, la complejidad creciente de la solución que estaba naciendo, derivó en clases que debían estar estructuradas de una manera extensible y escalable, lo cual fue dando paso a un conjunto de elementos genéricos que podían ser utilizados en otros contextos distintos a los de la aplicación existente.

Con el objeto de evitar "reinventar la rueda" cada vez que se quisiera crear alguna representación gráfica interactiva, y de aprovechar los elementos base ya creados, nace la idea de crear el Framework.

El desarrollo fue iterativo con prototipos, los cuales se definen en la sección 4.3. Cada iteración consiste en diseñar, verificar en código, probar esa unidad y validar. Estas iteraciones son muchas, y su duración dependerá de la situación. Hay mucho desecho entremedio, es decir, código que se hizo en un momento y cumplía con la funcionalidad, pero que fue reemplazado, ya sea por una mejora, o por eliminar es funcionalidad, por un tema de compromiso.

Las iteraciones son internas, y también las pruebas. Sin embargo, cada prototipo se construye para poder ser usado por terceros, y se cuenta con un servidor de bugs, para realizar reportes de fallas, las que serán corregidas para la siguiente versión. Por tanto, los prototipos ya son usables y proveen de ciertas funcionalidades, por lo cual la lista de prototipos propuesta en el Capítulo 5 está enfocada a marcar funcionalidades completadas, así se entrega algo cerrado y no se presentan funcionalidades hechas a medias.

El desarrollo del Framework se basa en aplicaciones que se desarrollan en paralelo, las cuales utilizan el Framework y al mismo tiempo generan nuevos requerimientos para éste. A

continuación se presentan las dos primeras aplicaciones que cumplieron con este rol en el desarrollo del Framework.

1.4 Módulo *Scheduling* software *CAPPS*

La primera aplicación relacionada con el Framework fue CAPPS (*Capacitated Production Planning and Scheduling*), dentro del marco del proyecto *Fondef D04I1428*, "Arquitectura configurable para la optimización de la logística de producción en la industria de procesos por lotes" [*Fon04*]. Más detalles de esta aplicación y sus funcionalidades se presentan en el Anexo B.

Primera iteración

Durante el desarrollo del proyecto, se requirió que Reinaldo Espejo, al inicio de su participación en éste en el año 2007, realizara una vista gráfica estática de un plan de proceso de lotes de líneas de producción, similar a una carta Gantt. La Figura 1.2 muestra lo que se esperaba como resultado del trabajo.

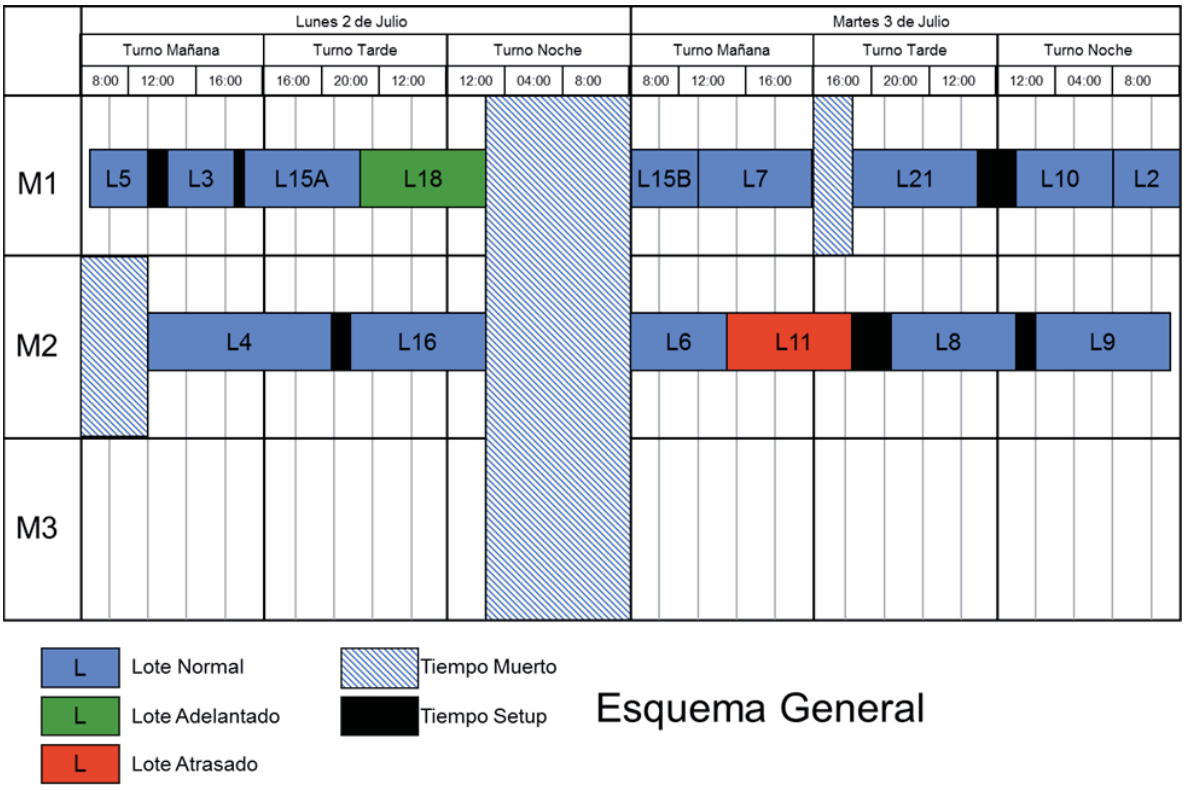

Figura 1.2 : ***Imagen de referencia para representación gráfica de planificación de lotes*** [*Fon04*]

Los elementos principales a ser creados para lograr esta representación son:

- **Lote**: se entiende por lote al conjunto de productos a ser procesados (manzanas, bebidas, semillas, etc). El lote tiene un "tamaño" en kilos, es decir, mientras más peso tenga el lote, más "grande" es. El lote tiene dos estados principales: programado y no programado. Programado significa que el lote será procesado en algún momento determinado en alguna línea o máquina, y el tendrá un tiempo de proceso, el cual variará según varios factores, siendo su peso el factor principal. Si el lote comienza antes o después de la fecha en que debería ser procesado idealmente, se dice que esta adelantado o atrasado respectivamente.
- **Líneas de Producción**: las líneas de producción son quienes se encargan de procesar los lotes, y su producción se representa horizontalmente en unidades de tiempo: minutos, horas, días, etc., y turnos cuando se trata de empresas que trabajen con esa separación de tiempo.
- **Tiempo de *Setup***: es el tiempo requerido para adecuar una línea de producción para que pueda procesar otro tipo de lote: por ejemplo, cambiar de embotellar jugo de naranja a jugo de frambuesa, debe ser lavado todo antes. Se representa como un espacio de tiempo entre dos lotes donde el segundo es quien gatilla la aparición de este tiempo.
- **Tiempo Muerto**: es el tiempo en que por alguna razón, no se trabaja. Horario de colación y días feriados son algunos de los casos más comunes.

Como se mencionó anteriormente, el requerimiento inicial consistía en realizar solamente una vista de representación gráfica de una programación de lotes sobre una o más líneas, similar al concepto presentado en la Figura 1.2. El trabajo a realizar consistió entonces en confeccionar una capa gráfica sobre una capa de datos que ya estaba desarrollada. La Figura 1.3 ilustra esta relación. Cabe mencionar que esta capa queda muy poco acoplada con la capa de datos, ya que en el fondo su única relación con ella es que la capa gráfica recibe todos los datos como un solo conjunto, una sola vez, al momento de la creación de la vista gráfica. Es decir, a la vista gráfica se le entregan colecciones de lotes, líneas de producción, tiempos de *setup* y tiempos muertos.

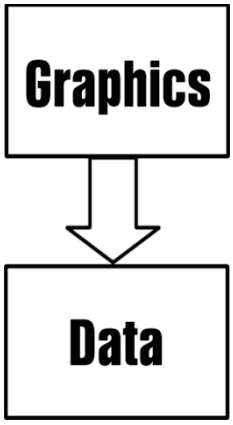

Figura 1.3: ***Relación inicial entre la capa gráfica y la capa de datos***

Para lograr la representación gráfica de un lote, y en general de todos los elementos presentes, es necesario convertir las unidades de tiempo en pixeles. Esto debido a que la representación gráfica es la representación de una línea de tiempo, y el ancho en este caso está determinado por una duración temporal. Por ejemplo si hacemos 1 pixel = 1 minuto, un rectángulo de 60 pixeles de ancho representa "algo" que dura 60 minutos. Es necesario crear una relación tiempo pixel que sea discreta (ya que los pixeles sólo son números enteros), ya que de lo contrario se producirían problemas de precisión, y esta relación debe mantenerse para todos los elementos gráficos por igual.

Además de representar el dato tiempo o duración de los elementos en juego, se extraen otros datos útiles para la visualización, como por ejemplo, código de lote, tipo de producto, etc. La Figura 1.4 muestra el caso de un lote gráfico.

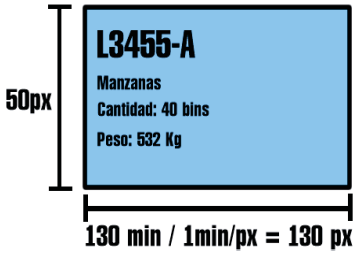

Figura 1.4: ***Representación gráfica de un lote, con proporción 1 minuto x 1 pixel***

La técnica para dibujar estos elementos en la pantalla consistía básicamente en la creación de un elemento gráfico para cada elemento de dato entregado, los cuales eran dibujados uno por uno

mediante el método *paintComponents (Graphics g)* del *JComponent* de *Swing* que contenía al panel de la representación completa. Este es un método estructurado de tipo *top-down*, lo que implica que lo que se pinta segundo está visualmente encima de lo que se pinta primero. Por ello, se seguía más o menos la siguiente secuencia de pintado:

1. Pintar fondo blanco
2. Pintar líneas, fechas, horas
3. Pintar lotes
4. Pintar tiempos de setup
5. Pintar tiempos muertos

Este pintado se realiza cada vez que la pantalla se actualiza, es decir, se podría tener hasta 60 llamadas a este método por segundo. Esto en un escenario normal de 60 fotogramas por segundo en una pantalla de 60 Hz de refresco vertical. Por lo tanto, se debe tener cuidado con las tareas que se realicen en el código de este método. Por ejemplo, cualquier bifurcación condicional de tipo *if-else* puede duplicar lo que el método se tarda en terminar, lo cual produce menos fotogramas por segundo, y ello a su vez se ve reflejado en lentitud de la vista gráfica. Sin embargo, para esta etapa de visualización estática de la programación de lotes, no afecta, y la metodología utilizada es la más adecuada para este caso. El Código 1.1 muestra cómo se realiza el pintado del lote. El código no está completo. Notar que para pintar cada lote gráfico, se está haciendo un recorrido por una colección de lotes gráficos previamente creada. Esto sigue las mismas reglas de que lo que se pinta segundo queda sobre lo que se pinta primero. El lote gráfico o *GraphicLot* es quien almacena datos como color del lote, posición X, Y, ancho y alto. Los datos X y ancho son determinados a través de un cálculo de transformación de fechas y tiempos en pixeles, según las proporciones dadas.

```
1     @Override
2     public void paintComponents (Graphics g)
3     {
4           super.paintComponents(g);
5           Graphics2D g2 = (Graphics2D) g;
6           //[...]
7           for(GraphicLot gl: graphicLots)
8           {
9                 //[...]
10                g2.setColor(gl.getColor());
11              g2.fillRect(
12                 gl.getX(),gl.getY(),gl.getWidth(),gl.getHeight());
13                      //[...]
14          }
15          //[...]
16    }
```

Código 1.1 : ***Segmento de código que logra el pintado de un lote gráfico***

Segunda iteración

Al notar los resultados rápidos y completos presentados en la primera versión, se generó la idea de aumentar las funcionalidades de la representación gráfica, añadiéndole interacción. La interacción requerida consistía en que el usuario pudiera mover con el mouse los lotes gráficos, de tal manera de "reprogramarlos".

Para crear esta interacción, se capturaron los eventos de mouse "apretado", "arrastrado" y "soltado". Al "apretar" el mouse (distinto a un *click*, que es apretar y soltar en un intervalo corto de tiempo), se identifica sobre cuál de todos los lotes fue realizada la acción, de tal manera de marcarlo como el lote gráfico que será manipulado. El código responsable de esto básicamente era un recorrido por la colección de lotes gráficos, y para cada uno veía si el punto del mouse estaba contenido dentro del rectángulo del lote.

El mouse al ser "arrastrado", realizaba cambios sobre el lote afectado y su posición *(x,y)* en la pantalla, según el cambio calculado con respecto a la posición inicial. Al "soltar" el mouse, se confirma la última posición *(x,y)*, y se altera el dato fecha del lote, según el cálculo pixel-tiempo. La relación entre la capa de datos y la capa de gráficos ahora es un poco más fuerte, dado que cambios en la parte gráfica alterarán al dato relacionado.

Al terminar esta primera etapa de interacción, fueron naciendo más requerimientos, el más importante fue el agregar una *pool* de lotes que no estuvieran programados, o que pudieran desprogramarse/programarse manualmente por el usuario. Esta *pool* se ubica debajo de las líneas y cuando se encuentran ahí, los lotes gráficos adoptan un tamaño estándar de 60x30 pixeles. La Figura 1.5 ilustra una sección de la aplicación, donde se muestra la acción de arrastrar un lote y la presencia del *pool* de lotes.

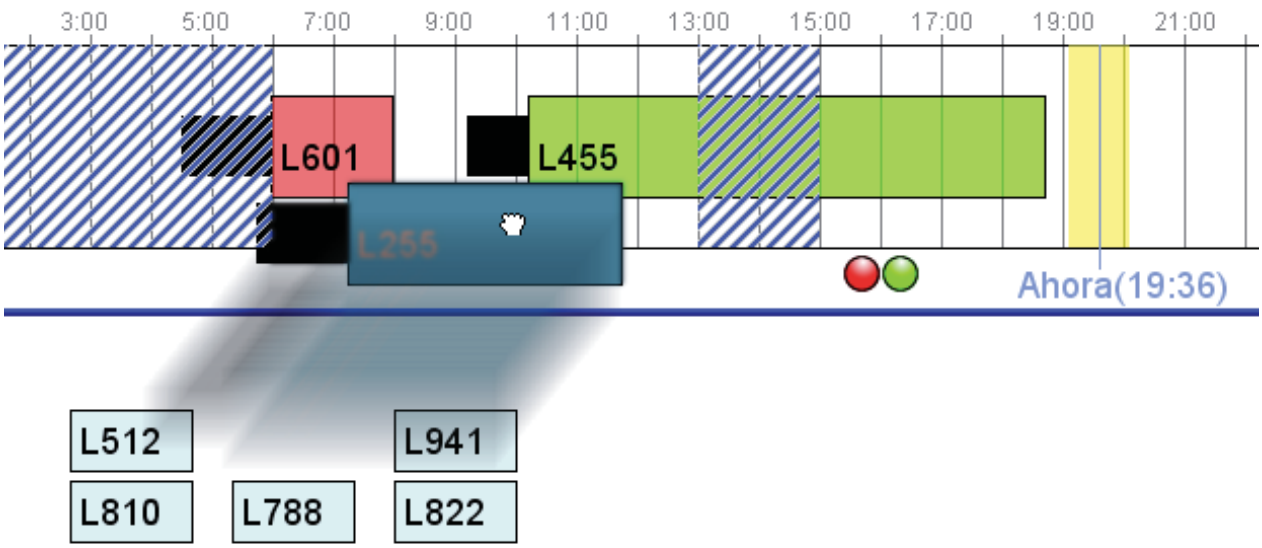

Figura 1.5 : ***Un lote gráfico siendo arrastrado con el mouse desde el pool de lotes no programados hacia la línea de producción***

La segunda iteración continuó sofisticándose y poniéndose más compleja. Algunas de sus nuevas características fueron:

- **Órdenes o pedidos**: se añade a la vista gráfica una representación gráfica de las órdenes o pedidos por parte de clientes, a los cuales se les asignan lotes para satisfacer sus cantidades demandadas, y a la vez se programa pensando en la fecha de entrega para el cliente. Los pedidos son asignados al lote mediante una línea dirigida, como muestra la Figura 1.6.
- ***Tooltips***: los *tooltips* son los textos de ayuda que aparecen cada vez que se posiciona el mouse sobre algún botón o elemento de la interfaz durante un par de segundos. Su propósito es proporcionar datos adicionales, información o ayuda para el usuario. Se agregaron *tooltips* en este caso para proveer al usuario con más datos relacionados a un lote u orden. La Figura 1.7 ilustra esta funcionalidad.

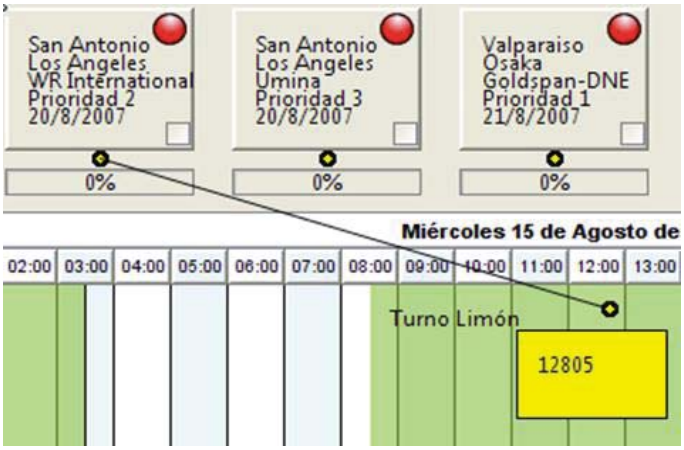

Figura 1.6 : ***Asignación de un pedido a un lote***

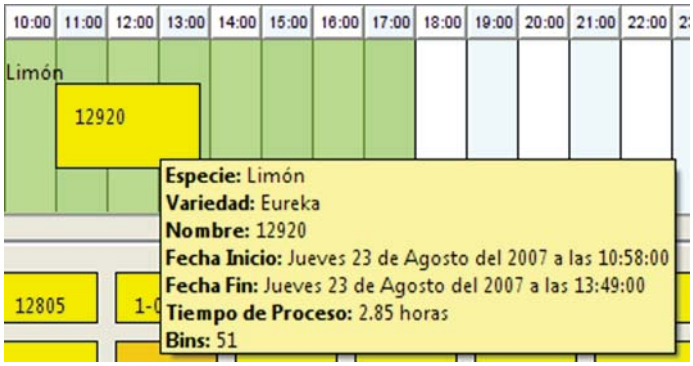

Figura 1.7 : ***Tooltip de un lote***

Otro cambio importante en esta versión fue sobre la arquitectura. Si bien antes se tenía *Graphics* y *Data* como capas de gráficos y datos respectivamente, ahora se deja la capa *Graphics* para elementos gráficos genéricos, y se crea una capa superior con elementos que extienden su funcionalidad. Como ejemplo, un componente gráfico genérico que tiene posición (x,y), un ancho y un alto, y métodos para determinarlos según la transformación minutos/pixel: los elementos en la capa superior que extienden a este elemento genérico serían el lote gráfico, el tiempo muerto gráfico, el turno gráfico, etc. La Figura 1.8 muestra esta nueva capa.

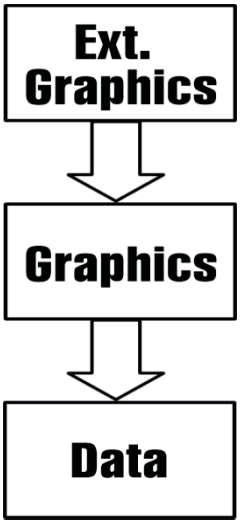

Figura 1.8 : ***Nueva capa que de extensión de gráficos***

Esta versión mantuvo la manera de dibujar los elementos gráficos de la primera versión. Dada la complejidad aumentada, el método *paintComponent* pasó de tener 100 líneas de código en la versión 1, a tener más de 1500 líneas de código en la versión 2. Esto claramente impuso un castigo sobre el rendimiento y la carga del procesador. Pruebas realizadas sobre la aplicación en situaciones extremas de líneas de un año de largo (525600 pixeles de ancho) con 300 lotes, terminaban en una carga de 99% de la CPU, y congelamientos de la pantalla. Si bien estas situaciones no se acercaban a la realidad, fue claro que la metodología de dibujado e interacción estaba perdiendo su eficiencia.

Tercera iteración

Para mejorar la segunda versión, fue necesario crear una nueva manera de realizar los gráficos interactivos, que fuera lo más eficiente posible en cuanto al dibujo y la interacción. En el fondo lo que se buscaba era evitar tener que realizar todos esos ciclos para dibujar e interactuar (dibujar lote a lote, preguntar lote a lote donde se presionó el mouse).

En el capítulo 6 se explica con detalles esta nueva solución para lograr gráficos interactivos eficientes, utilizando ejemplos menos complejos. Por ahora basta con entender que se dejó de lado el dibujar todo en un solo método, dejando la responsabilidad del dibujado a cada elemento por separado. Esto se logra básicamente haciendo que cada elemento gráfico se pinte a sí mismo. Para llegar a esto, fue necesario crear cada elemento como un componente (*Component* de Java AWT), de tal forma de que cada uno tuviera su propio método *paintComponent*.

El trabajo fue extenso, más de 5 meses, y frustrante cuando no se lograban los resultados, y todo se realizó sin apoyo de bibliografía ya que fue algo de creación propia. Finalmente se volvió al estado en que había quedado la versión 2. Esto, ya que se partió desde cero, con el dibujado y la interacción, por lo que el resto del equipo por mucho tiempo no confió en que se lograría llegar a lo mismo, y se mantuvo la versión 2 junto con la 3 durante varios meses.

La Figura 1.9 muestra la estructura final de las capas. Notar que *Component* toma los elementos genéricos de la capa *Graphics*, un esfuerzo adicional para así permitir independencia de las implementaciones (esto puede considerarse un pequeño Framework, o la primera versión informal del Framework).

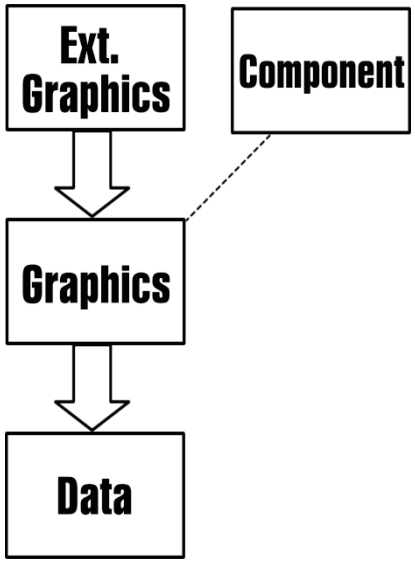

Figura 1.9: ***Nueva capa Component, para la tercera versión***

1.5 Software de diseño diagramas multiagente

Al finalizar el proyecto anterior, a fines del 2008, el autor comenzó a trabajar sobre un prototipo de una aplicación tipo CASE para sistemas multi-agente. Las características gráficas de esta aplicación hicieron que fuera natural el tratar de rescatar los elementos genéricos realizados en el primer proyecto, y junto con ello se comenzó a pensar en crear más elementos genéricos, para esta y otras aplicaciones con representaciones gráficas interactivas. Con esto se comenzó a concretar la idea de diseñar y construir un Framework, compuesto con estas clases genéricas.

Se construyeron 3 prototipos de esta aplicación hasta que el proyecto terminó tempranamente. El desarrollo de los prototipos consistía en desarrollar alguna característica nueva, como por ejemplo métodos para cambiar de tamaño a algún elemento, y se rescataba este código de una manera genérica para ser incorporado al Framework. Finalmente se eliminaba este método en la implementación y se cambiaba por un uso del método genérico del Framework. Esto significaba doble trabajo, pero valía la pena ya que no se alteraba el Framework hasta que los nuevos métodos estuvieran funcionando bien. La Figura 1.10 muestra una pantalla del primer prototipo de esta aplicación. Más detalles de este prototipo se presentan en el Anexo B.

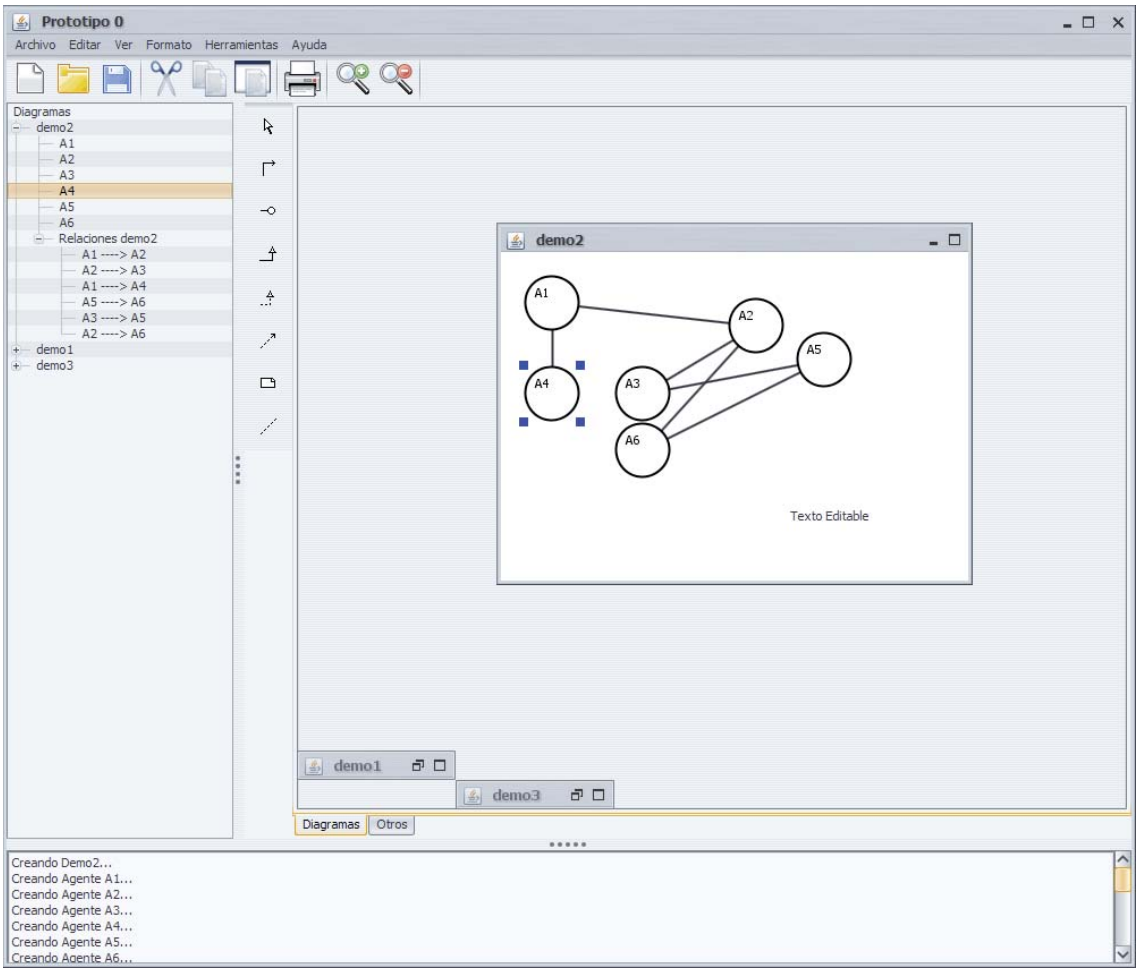

Figura 1.10: ***Primer prototipo de la herramienta tipo CASE para sistemas multi-agente***

1.6 Estructura de la tesis

Los capítulos 2 y 3, introducen los conceptos y temáticas necesarios para la comprensión del problema y el contexto de la solución. El Capítulo 2 introduce y comenta brevemente sobre las herramientas que proporciona el lenguaje Java para el desarrollo de gráficos e interfaces de usuario.

En el Capítulo 3 se definen los conceptos de framework de software. Además, se presenta la notación escogida, el perfil UML-F, y sus componentes, de manera que el lector pueda comprender los diagramas que se presenten posteriormente.

En el Capítulo 4, se plantea la solución, el Framework de software, y se define la metodología que se siguió para su desarrollo. Abarca el desarrollo del Framework, y explica con ejemplos y código, cómo funciona y cómo se aplica en un proyecto de desarrollo. Además de ello, se presenta la estructura del Framework, sus clases y paquetes más importantes.

El Capítulo 5 muestra las pruebas realizadas sobre el Framework, y analiza los resultados obtenidos.

2. Creación de Gráficos en Java

Dentro de la API de Java, el desarrollador encuentra un conjunto de clases que puede utilizar para la creación de elementos visuales, como botones, ventanas, campos de texto, etc. Estas clases componen los dos grandes paquetes de elementos de interfaz de usuario de Java: **AWT** y **Swing**. A continuación se analizará cada una de ellas, comenzando por AWT. Seguido de esto se revisarán Swing, y Java2D. Este último es utilizado para la realización de gráficos en dos dimensiones en Java.

2.1 Java Abstract Window Toolkit

El **Abstract Window Toolkit** (**AWT**) fue creado en una de las primeras versiones de Java. Está compuesto de clases que permiten la creación de ventanas y elementos visuales para aplicaciones. El objetivo que tenía era solucionar el problema de tener que programar todo el código de interfaces cada vez que se cambiaba de plataforma. Sin embargo, AWT depende de la plataforma, en el sentido de cómo se verán los componentes, es decir, tendrá un look "nativo" a la plataforma, lo cual no siempre es bueno.

Las interfaces de usuario modernas están construidas alrededor de la idea de **componentes**: elementos gráficos reusables que implementan una cierta parte específica de la interfaz gráfica. Botones, menús, ventanas, barras de desplazamiento, son algunos ejemplos de componentes. AWT viene con un grupo básico de componentes de interfaz de usuario, dando además la posibilidad de poder crear componentes propios (combinaciones de componentes básicos), junto con los elementos para la comunicación entre componentes y el resto del programa.

AWT está compuesto de los siguientes paquetes:

- **java.awt** - Contiene todas las clases para crear interfaces de usuario y para pintar gráficos e imágenes.

- **java.awt.event** - Provee de interfaces y clases para manejar distintos tipo de eventos disparados por los componentes de AWT.
- **java.awt.datatransfer** - Provee las interfaces y clases necesarias para transferir datos entre aplicaciones (en un contexto de "drag and drop").
- **java.awt.dnd** - Drag and drop en es un gesto de manipulación directa, que se puede encontrar en muchos sistemas de GUI, que provee un mecanismo para transferir información entre dos entidades asociadas lógicamente mediante elementos de presentación en la GUI.
- **java.awt.color** - Conjunto de clases para el uso de colores, en un contexto de Java2D.
- **java.awt.font** - Conjunto de clases que proveen creación y manipulación de Fuentes, en un contexto Java2D y de componentes que contengan texto.
- **java.awt.geom** - Contiene las clases que representan figuras geométricas básicas o "shapes", en un contexto Java2D.
- **java.awt.image** - Las clases utilizadas para manipulación y visualización de imágenes, en un contexto Java2D.
- **java.awt.image.renderable** - Provee las clases para la creación de imágenes independientemente renderizables, en Java2D.
- **java.awt.print** - Contiene las clases que conforman la API para impresión.
- **java.awt.im** - Contiene las clases e interfaces que componen al Input Method Framework.

No se profundizará mucho más en AWT, ya que hoy en día es sólo la base de APIs gráficas más sofisticadas, como Swing y Java2D. La Figura 2.11 muestra la jerarquía de clases que forman parte de AWT. Se puede ver que todas derivan de un tipo "fundamental", *Component*. Además, se puede apreciar que proporciona sólo elementos básicos, como botones (*java.awt.Button*), cajas de chequeo (*java.awt.Checkbox*) y campos de texto (*java.awt.TextField*).

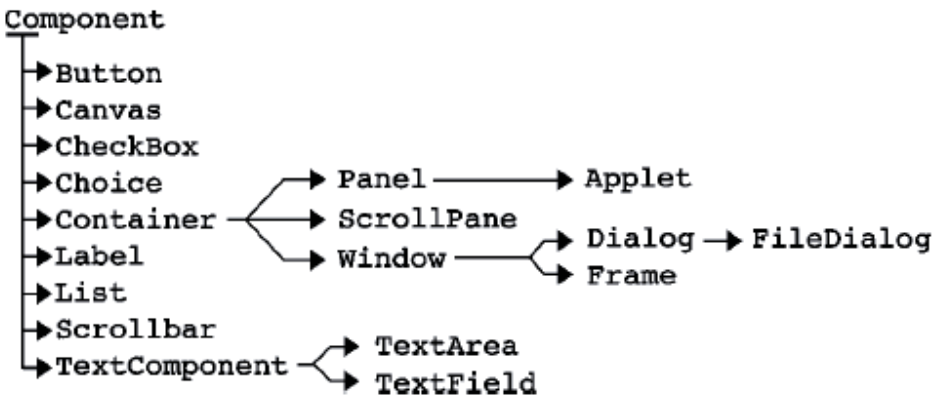

Figura 2.11 : ***Jerarquía de componentes de AWT*** **[Ada05]**

2.2 Java Swing

En Abril de 1997, JavaSoft anuncia el *Java Foundation Classes* (**JFC**), el cual aumenta las prestaciones de AWT. La mayor parte del JFC la conforma un conjunto de componentes de interfaz de usuario que son mucho más completos, flexibles y portables. A estos nuevos componentes se les denomina **Swing**. Más que un nuevo conjunto de herramientas, Swing es una extensión de AWT. Como se mencionó anteriormente, AWT es dependiente de la plataforma en cuanto a cómo se verán sus componentes en la aplicación final. Estos componentes se llaman "pesados", en inglés "*heavyweight*". Swing está basado sólo en componente "livianos" ("*lightweight"*). Esto significa que Swing maneja cómo se mostrarán todos sus componentes (utiliza un *look and feel* propio). La ventaja de esto es que ya no sólo las aplicaciones son multi-plataforma, sino que también se mostrarán al usuario de la forma que el desarrollador pretendía que fuera. [Eck98] [Ada05]

Además de los nuevos componentes, Swing realiza 3 mejoras importantes sobre AWT. Primero, no depende del sistema operativo para la obtención de componentes nativos. Está escrita enteramente en Java, y genera sus propios componentes. Este nuevo enfoque soluciona el problema de la portabilidad, ya que los componentes no heredan los comportamientos cambiantes del entorno de ejecución. Segundo, como Swing está en control total de los componentes, también lo está en cuanto a cómo los componentes se ven en la pantalla. Esto le otorga un mayor control al desarrollador sobre la manera en que se presentará la aplicación. Tercero, Swing distingue claramente entre lo que son los datos que un componente muestra, el "modelo", y lo que se está viendo, la "vista". Como se verá en el Capítulo 3, esto significa que Swing utiliza el patrón **Modelo-Vista-Controlador**. Esta separación significa que los componentes son extremadamente flexibles. Es fácil adaptar los componentes para mostrar nuevos tipos de datos

que su diseño original no anticipaba, o cambiar la manera en que un componente se ve, sin tener que preocuparse sobre los datos que éste representa.

La Figura 2.12 muestra la jerarquía del *JComponent*, el equivalente de Swing del *Component* de AWT. Se puede apreciar que Swing provee de muchos más elementos gráficos que AWT (ver Figura 2.11), lo que permite el desarrollo de aplicaciones visualmente más ricas.

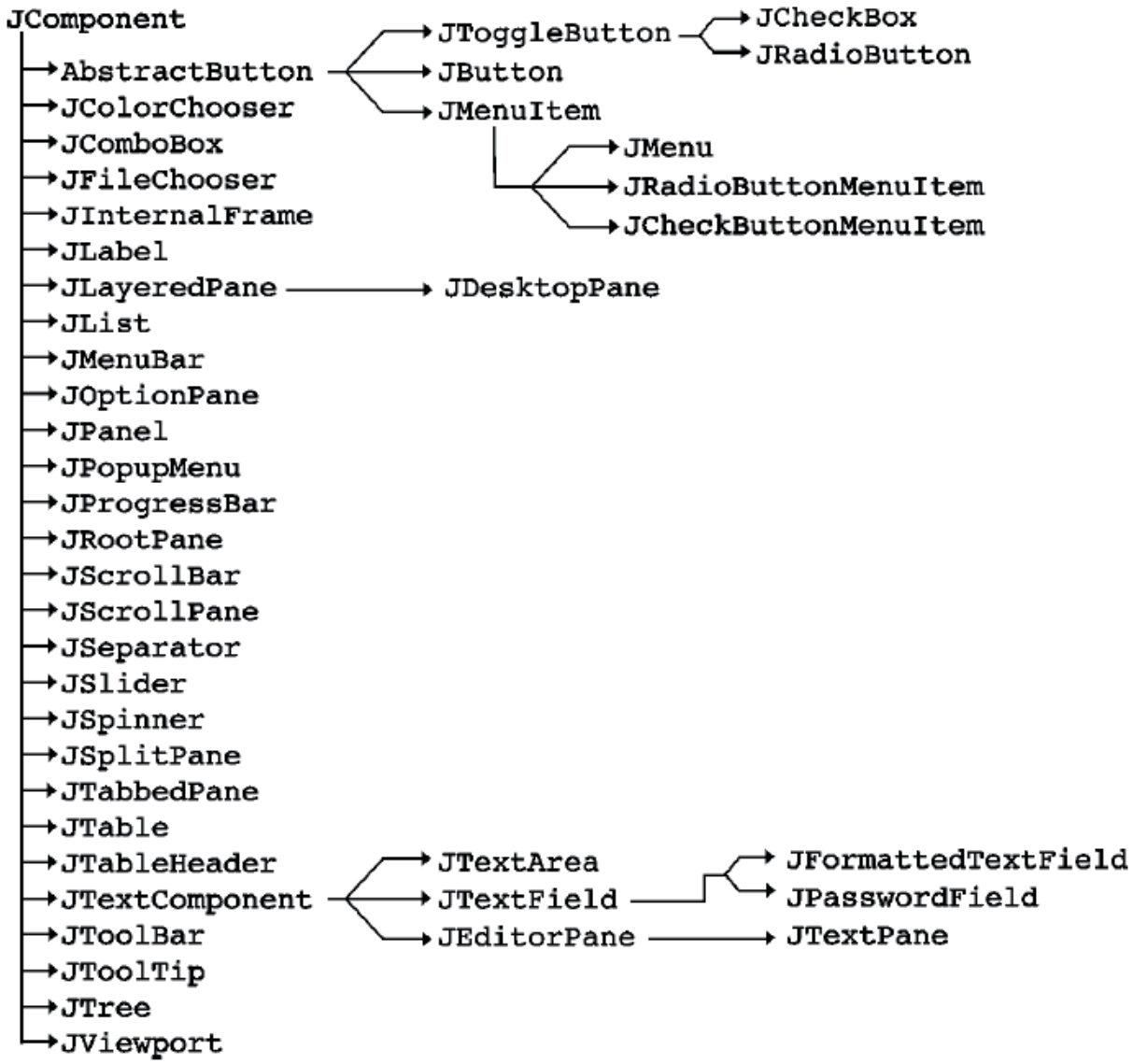

Figura 2.12 : ***Jerarquía de componentes de Swing*** **[Ada05]**

2.3 Java2D

Java2D es un conjunto de clases necesarias para crear gráficos de alta calidad. Incluye la capacidad de realizar transformaciones geométricas, procesamiento de imágenes, despliegue de texto bidireccional, etc. Así como Java Swing y AWT construyen las interfaces de usuario, con Java2D se construyen gráficos, figuras y dibujos. Java2D forma parte de AWT, y está formado por los siguientes paquetes:

- **java.awt** - Contiene todas las clases para crear interfaces de usuario y para pintar gráficos e imágenes.
- **java.awt.image** - Contiene todas las clases para crear interfaces de usuario y para pintar gráficos e imágenes.
- **java.awt.color** - Conjunto de clases para el uso de colores, en un contexto de Java2D.
- **java.awt.font** - Conjunto de clases que proveen creación y manipulación de Fuentes, en un contexto Java2D y de componentes que contengan texto.
- **java.awt.geom** - Contiene las clases que representan figuras geométricas básicas o "shapes", en un contexto Java2D.
- **java.awt.print** - Contiene las clases que conforman la API para impresión.
- **java.awt.image.renderable** - Provee las clases para la creación de imágenes independientemente renderizables, en Java2D.

Previo al lanzamiento de Java2D, los elementos gráficos en Java eran bastante limitados:

- Las líneas podían ser dibujadas sólo con un píxel de grosor.
- Existían muy pocas fuentes disponibles para texto.
- No se podían manipular las figuras geométricas.
- Las operaciones de rotación y escala tenían que ser implementadas manualmente.
- Si se querían rellenos especiales, como gradientes o patrones, había que implementarlos manualmente.
- Existía una manipulación de imágenes rudimentaria.
- El control sobre las transparencias era ineficiente.

Java2D supera todos estos problemas, y además ofrece más funcionalidades. No es objeto de este trabajo analizarlas, existe una demostración de Java2D donde se pueden apreciar todas las posibilidades de elementos gráficos y efectos que se pueden crear. La Figura 2.13 muestra una captura de pantalla de la demostración de las capacidades de Java2D en ejecución. Se pueden apreciar ciertas funcionalidades avanzadas, como el dibujo de curvas paramétricas (*java.awt.geom.CubicCurve2D, java.awt.geom.QuadCurve2D*).

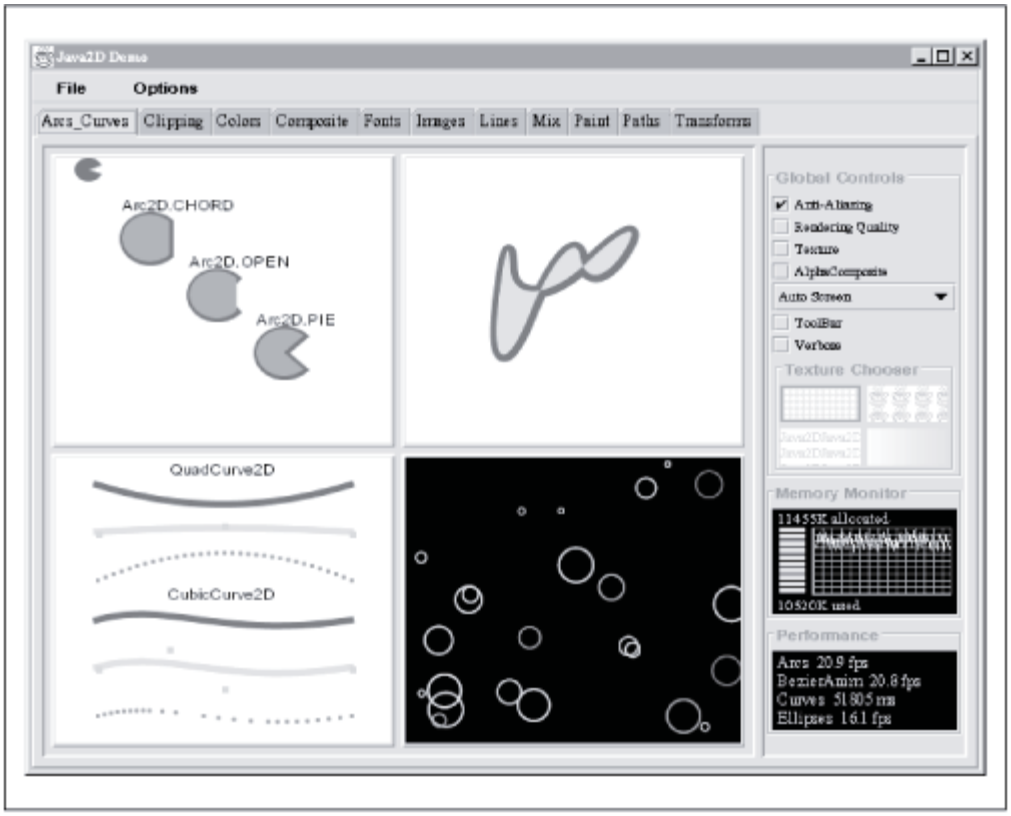

Figura 2.13 : ***Captura de Pantalla demostración de Java2D en ejecución***

Para utilizar Java2D, se necesitan dos elementos: un "papel" (o "*canvas*" en inglés) donde se mostrarán los gráficos y un "lápiz" o "pincel" al cual se le darán las instrucciones sobre el qué dibujar en pantalla. En Swing, este papel es un componente (*JComponent*) y su lápiz es la clase *Graphics2D*. La manera de realizar el "dibujo" es realizando una redefinición del método paintComponent(Graphics g) del JComponent, lo que en el fondo se entiende como que el JComponent ya no realizará su "pintado" por defecto, sino que se le dirá qué dibujar en pantalla y cómo hacerlo.

A modo de ejemplo se muestra un segmento de código donde se dibuja un rectángulo de 20x20 pixeles en pantalla:

```
1     public void paintComponent(Graphics g)
2     {
3                   g.drawRect(0, 0, 20, 20);
4     }
```

Código 2.2 : ***Segmento de código necesario para dibugar un rectángulo de 20x20 pixeles***

Java AWT, Swing y Java2D, conforman lo que se denomina *Java Foundation Classes*. La Figura 2.14 muestra de manera general, las JFC, de las cuales interesan para este trabajo sólo las 3 estudiadas en este capítulo.

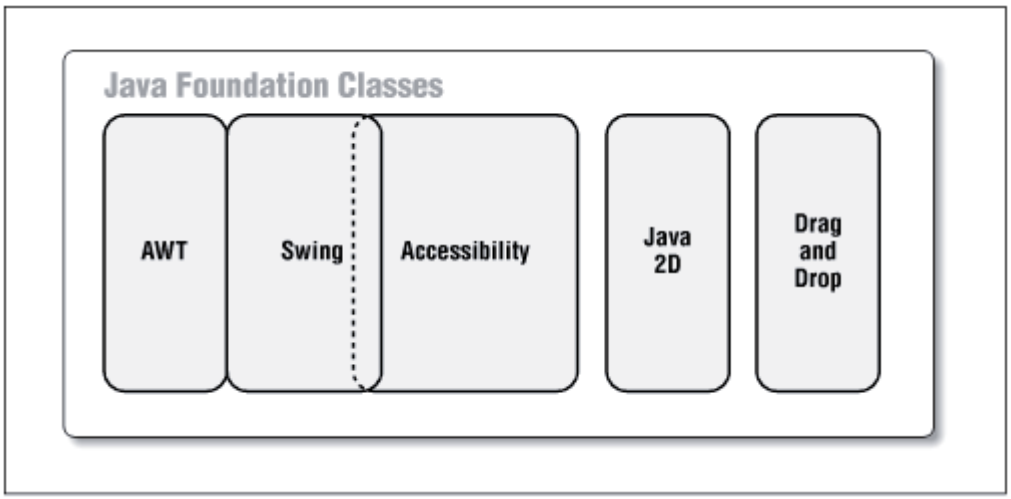

Figura 2.14 : ***Java Foundation Classes*** **[Ada05]**

3. Frameworks de Software

Todo el trabajo realizado está en torno al desarrollo de un *framework de software*, por lo cual es importante que el lector comprenda lo que esto significa, y el enfoque, propósitos y usos que tiene este tipo de Framework.

3.1 Conceptos

Un framework de software es un conjunto de códigos fuente o bibliotecas, que provee una funcionalidad común a un cierto tipo de aplicaciones. Mientras que una biblioteca usualmente proveerá sólo una cierta pieza específica de funcionalidad, los frameworks ofrecen un más amplio rango donde todos son usualmente utilizados por cierto tipo de aplicación. [TUL2008]

Las ventajas de utilizar un framework de software son [CWA06]:

- Al usar código que ya ha sido construido, probado, y usado por otros programadores, aumenta la confiabilidad y reduce el tiempo de programación, lo que ahorra dinero en un contexto de desarrollo empresarial.
- Los frameworks pueden proveer seguridades que son comúnmente requeridas por un tipo en común de aplicaciones. Esto le da a cada aplicación desarrollada con el framework el beneficio de tener estas seguridades sin el costo extra de tener que desarrollarlas.
- Los frameworks ayudan a la modularidad del código al estar manejando tareas de "bajo nivel". Un ejemplo típico es que el desarrollador se preocupa de tareas específicas del negocio, mientras que un framework realiza las tareas de conexión a las bases de datos, autentificación de usuarios, etc.

- Los frameworks llevan a la programación basada en patrones de diseño y mejores prácticas en general, al imponer estas arquitecturas a los desarrolladores implementadores.
- Las mejoras realizadas a un framework pueden aumentar la funcionalidad de la aplicación, sin requerir programación extra por parte del desarrollador de la aplicación final.

Las desventajas de utilizar un framework de software son [CWA06]:

- A veces el desempeño puede verse afectado; puede ocurrir cuando el código del framework está demasiado generalizado y no está optimizado para situaciones particulares. Sin embargo es un costo pequeño comparado con la facilidad de desarrollo que está dando.
- Los frameworks a menudo requieren un cierto aprendizaje de ellos para poder usarlos correcta y eficientemente. Esto no es un problema si el framework va a ser utilizado de manera reiterada por el desarrollador.
- Si el framework es demasiado rígido y estructurado, será un problema a la hora de querer cambiar una funcionalidad, incluso más problema que hacer la funcionalidad desde cero. Un buen framework debería dar utilidad y estructura al mismo tiempo que da suficiente flexibilidad como para no imponerse en el camino del desarrollador.
- Problemas de seguridad y fallas en el framework se traspasan a todas las aplicaciones que usan el framework, por lo cual deben realizarse pruebas y parches a la aplicación por separado.

3.2 Cualidades de un framework bien diseñado

Según KRZYSZTOF CWALINA [Cwa09], un buen Framework de Software, debe tener los siguientes atributos:

- ***Simplicidad***: es muy fácil añadir funcionalidades a un framework a medida que van apareciendo nuevos requerimientos. Sin embargo, se debe cuidar el diseño del framework, realizando constantes revisiones de éste, para comprobar que la arquitectura no se ha visto comprometida y que no haya quedado un diseño complejo a causa de las nuevas añadiduras. No se debe entregar si no se está conforme con el diseño resultante.

- ***Costo***: un buen diseño de framework no ocurre mágicamente. Es un trabajo duro y consume mucho tiempo y recurso. Si no se está dispuesto a invertir en el diseño, no se puede esperar crear un framework bien diseñado. El diseño debe ser una parte explícita del proceso de desarrollo del framework.

- ***Compromiso***: el diseño contiene muchos compromisos (sacrificar una cosa por otra) y el tomar las decisiones correctas, considerando las alternativas, los beneficios y problemas. Si uno se encuentra con un diseño sin compromisos, seguramente hay algo grande que no se está viendo. Cada framework presentará sus propios compromisos, y es responsabilidad del desarrollador del framework el tomar la decisión correcta.

- ***Elementos del pasado***: muchos de los frameworks exitosos toman prestado y se construyen a partir de diseños existentes. Es posible, y deseable, introducir soluciones completamente nuevas en el diseño del framework, pero debe ser realizado con cautela. Mientras más conceptos nuevos tenga, disminuye la probabilidad de que el diseño global se mantenga correcto.

- ***Evolutivo***: el pensar en cómo evolucionará el framework en el futuro cercano tiene sus compromisos. Por un lado, el diseñador del framework ciertamente puede tomarse más tiempo y esfuerzo en el proceso de diseño, y ocasionalmente se puede introducir complejidad adicional "por si acaso". Pero, por otro lado, la consideración cuidadosa puede también salvar a los desarrolladores del framework de entregar algo que se degradará con el tiempo, o peor aún, algo que no será capaz de preservar su compatibilidad hacia atrás. Como regla general, es mejor dejar una funcionalidad completa para la siguiente entrega, que realizarla a medias en la entrega actual.

- ***Consistente***: consistencia es la característica clave de un framework bien diseñado. Es uno de los factores de productividad más importantes. Un framework consistente permite el traspaso de conocimiento entre las partes del framework que el desarrollador conoce y las partes que el desarrollador está tratando de aprender. La consistencia también ayuda a los desarrolladores a reconocer rápidamente qué partes del diseño son verdaderamente únicas para un área funcional en particular que por lo cual requieren atención especial, y qué partes son sólo los mismos patrones de diseño de siempre.

3.3 Patrones de diseño orientados a objetos

Los patrones de diseño orientados a objetos propuestos por ERICH GAMMA et al. [Gam94], tienen como propósito el formalizar soluciones a problemas de diseño, que se presentan a menudo en el

desarrollo de software orientado a objetos. La tabla 3.1 muestra la lista de patrones propuesta por ellos [Gam94]. Los patrones en la tabla están clasificados bajo dos criterios.

El primer criterio, llamado **propósito**, refleja lo que el patrón hace. Los patrones pueden tener un propósito **creacional**, **estructural** o de **comportamiento**. Los patrones creacionales se preocupan del proceso de la creación de objetos. Los patrones estructurales se preocupan con la composición de las clases u objetos. Los patrones de comportamiento caracterizan las maneras en que las clases u objetos interaccionan y distribuyen la responsabilidad.

El segundo criterio, llamado **alcance**, especifica si el patrón se aplica primariamente a **clases** o a **objetos**. Estas relaciones se establecen a través de herencia, por lo cual son estáticos, es decir, fijos en tiempo de compilación. Los patrones de objetos se encargan de las relaciones entre objetos, las cuales pueden ser cambiadas en tiempo de ejecución y son más dinámicas. Casi todos los patrones utilizan herencia en algún nivel. Por lo tanto, sólo los patrones etiquetados como patrones "clase", son los que se enfocan en relaciones entre clases. Notar en la tabla que la mayoría de los patrones se encuentran en el alcance objeto.

	Propósito			
		Creacional	**Estructural**	**Comportamiento**
Alcance	**Clase**	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Tabla 3.1 : ***Patrones de Diseño Orientados a Objetos***

El lenguaje Java se encuentra construido con estos patrones en consideración. Se pueden apreciar casos como el patrón *Observer*, el cual se utiliza en toda la arquitectura de *listeners* de Java. También se encuentra el patrón *Singleton* en varias secciones de Java, con clases que comparten una sola instancia de ellas en todo momento.

El patrón clave para todo este trabajo, y en el cual se basan las APIs de GUI de Java, el patrón Model-View-Controller, no se encuentra en esta lista, y se presenta a continuación.

El Patrón Modelo-Vista-Controlador

El patrón **Modelo-Vista-Controlador** (**MVC**) es un conocido patrón de diseño orientado a objetos de descomposición de diseño de interfaz de usuario. Los componentes (elementos fundamentales de las GUIs, como botones, ventanas, paneles, etc.) se separan en 3 partes: el modelo, la vista, y el controlador. Si bien el patrón es simple, es la base de diseño para Java AWT y Swing, las APIs de Java para la creación de interfaces de usuario, por lo que es importante revisar cómo se genera esta separación. La Figura 3.15 ilustra la arquitectura Modelo-Vista-Controlador.

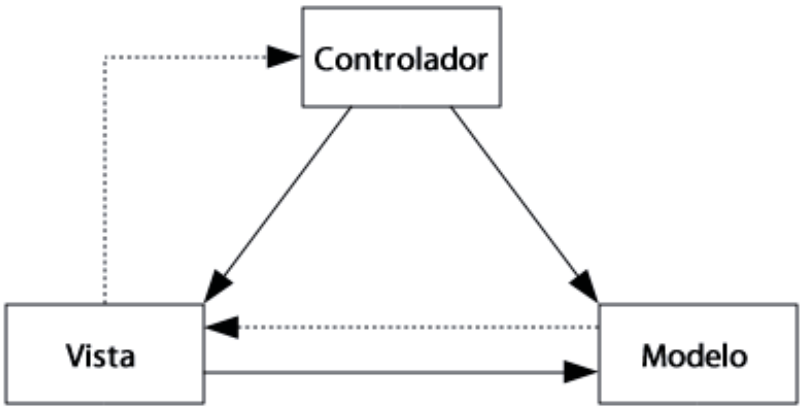

Figura 3.15 : ***La Arquitectura Modelo-Vista-Controlador.***

Modelo: el modelo es el responsable de mantener todos los aspectos del estado de un componente. Esto incluye, por ejemplo, valores como el estado de presionado/no presionado en un botón, o la información de la fuente utilizada por el texto de un componente, o la información de cómo está estructurado el componente. Un modelo puede ser responsable por la comunicación indirecta con la vista y el controlador. Es decir, el modelo no "conoce" su vista y su controlador, no mantiene o extrae referencias a ellos. En lugar de ello, el modelo enviará notificaciones (en Java lo que se conoce como eventos (*events*)). En la Figura 3.15 esta comunicación indirecta está representada por las líneas punteadas.

Vista: la vista determina la representación visual del modelo del componente, el "*look*". Por ejemplo, la vista muestra el color correcto del componente, en el caso de que el componente se encuentre en un estado de levantado o aplastado, en el caso de un botón, o muestra los textos con la fuente deseada. La vista es responsable de mantener su representación en pantalla actualizada constantemente, lo que puede hacer luego de haber recibido mensajes indirectos por parte del modelo o mensajes desde el controlador.

Controlador: el controlador es el responsable de determinar si el componente debe reaccionar a algún evento desde dispositivos de entrada como el teclado o mouse. El controlador es el "*feel*" del componente, y determina qué acciones se realizan cuando se está usando el componente. El controlador puede recibir mensajes desde la vista, y mensajes indirectos desde el modelo.

Si bien se ha dicho que el MVC no se utiliza de esta manera básica, directamente, sí se mantiene siempre la separación clara entre los tres. Esto se respeta tanto en Java AWT, Swing y el framework propuesto. En el caso de este último, se propondrá una separación del modelo, lo que se explica en la sección 4.3.

3.4 El Perfil UML-F

Mecanismos de extensibilidad en UML

El estándar de UML ofrece mecanismos de extensión del lenguaje. El motivo de esto es que UML no pretende presentar restricciones en cuanto al nivel de expresividad deseado en cada modelo, así como tampoco restringir todos los dominios posibles de utilización de UML como lenguaje de modelado. De esta manera, y según el estándar de UML 2.0, existen tres mecanismos centrales para extender el lenguaje de manera controlada: Estereotipos, Valores Etiquetados, y Restricciones. [Fon01]

Un **estereotipo** extiende el vocabulario de UML, permitiendo obtener nuevos tipos de bloques de construcción. Estos bloques, derivados de los ya existentes, tienen la ventaja que son específicos a un problema en particular. Algunas de las aplicaciones de los estereotipos son como sigue:

- Un estereotipo se puede ocupar de las características de una pieza del software.
- El estereotipo puede ocuparse de la representación visual de un modelo en una vista determinada.

- Un estereotipo también puede describir ciertas restricciones. Por ejemplo, <<persistencia>> puede expresar que las instancias de una clase son almacenables, sin indicar cómo se implementa en la práctica esa característica.
- Un estereotipo puede expresar una intención desde el modelador hacia cualquier usuario, sobre un elemento del modelo que estos últimos deben entender. Por ejemplo, una clase se puede estereotipar como <<adapt-static>>, indicando con esto que es una clase candidata a ser extensible y adaptada por sus subclases.
- Finalmente, un estereotipo puede describir el rol de una clase en un patrón de colaboración. Esto permite una descripción compacta de los participantes de patrones particulares.

Por su parte, un Valor Etiquetado extiende las propiedades de un estereotipo de UML, lo que permite agregar información adicional en la especificación de dicho estereotipo. Un valor etiquetado está compuesto de dos elementos: un nombre, y un valor, escribiéndose de la forma {nombre-etiqueta = valor}. Ejemplos de su aplicación son:

- Un valor inicial de un atributo.
- El nombre del creador de una clase.
- El número de versión o la fecha del último cambio realizado sobre el elemento en cuestión.
- Una cadena de caracteres que contiene un comentario o una restricción en el elemento modelado.
- Una política de almacenamiento o transmisión sobre una red.
- Un ícono para representación gráfica en una herramienta case.

Por último, una *Restricción* extiende la semántica de un bloque de construcción de UML, lo que permite agregar reglas adicionales, o eventualmente modificar las ya existentes.

La figura 3.16 muestra un ejemplo de los mecanismos de extensibilidad de UML. Por un lado, el estereotipo <<centralDepot>> no es un estereotipo provisto por el estándar de UML, sino que emerge directamente del dominio que se está modelando. Así mismo, una de las operaciones asociadas a la clase Pedidos puede desencadenar un problema de “sobre utilización” de alguna capacidad (o problema de desbordamiento), por lo que se ha incluido otro estereotipo

denominado <<exception>>, también suscitado por el dominio de aplicación. En este caso, las excepciones observadas pasan a conformar un bloque básico de construcción.

Por otro lado, se desea agregar información adicional sobre el estereotipo <<centralDepot>>, como lo es su tipo de estereotipo, y el autor. El tipo y su autor no son conceptos primitivos de UML; sin embargo, pueden ser añadidos a cualquier bloque de construcción (como en este caso, a una clase).

Finalmente, también se amplía la semántica de la clase Pedidos. En este caso, se desea restringir a la clase, de tal modo que todas las inserciones de pedidos se efectúen respetando una jerarquía de prioridades.

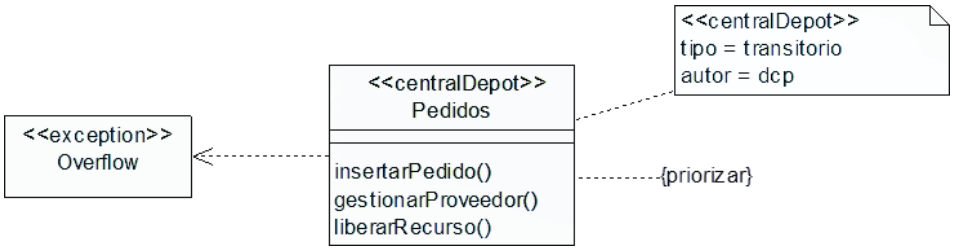

Figura 3.16: ***Mecanismos de Extensibilidad en UML 2.0.***

En conjunto, estos tres mecanismos de extensibilidad permiten configurar y ampliar UML de acuerdo a las necesidades requeridas. Es posible añadir nuevos bloques de construcción, modificar la especificación de los bloques ya existentes, entre otros.

UML-F como perfil de UML

Ciertamente, el aprender y comprender la arquitectura y funcionamiento de un framework de software requiere de esfuerzos adicionales respecto de la comprensión del software convencional. La notación UML estándar no ofrece elementos preparados para este propósito. Las extensiones resumidas del perfil UML-F apoyan el desarrollo y adaptación de frameworks, centrándose en la localización de los puntos de variabilidad de un framework.

El perfil UML-F persigue alcanzar ciertas metas dentro del área de desarrollo de arquitecturas frameworks, las que se detallan a continuación:

- UML-F entrega elementos notacionales para documentar de manera adecuada patrones de diseño bien conocidos.

- UML-F es, en sí mismo, el medio que de forma directa permite documentar cualquier patrón de framework, incluyendo los que se produzcan a futuro.
- UML-F emplea elementos nemotécnicos en su notación.
- UML-F está construido sobre el estándar UML, es decir, las extensiones generadas pueden ser definidas sobre la base de mecanismos de extensión de UML ya existentes.
- Los elementos notacionales son adecuados para la integración en entornos de desarrollo de UML (tools y herramientas case).

Según se indica en la sección anterior, el estándar UML define tres mecanismos de extensibilidad. Sobre el mecanismo de extensibilidad basado en restricciones, es posible decir que si se observa una alta utilización de él, se genera una complejidad que no es necesariamente útil en la aproximación de UML-F. Es por ello que, en primera instancia, UML-F no contempla la utilización de restricciones. Sin embargo, restricciones más sencillas del estándar UML son introducidas como etiquetas en UML-F.

UML-F identifica un subconjunto de UML, que basta para el desarrollo de frameworks y su adaptación. Esto no significa que el resto de UML no sea utilizado. En cambio, ello ayuda al desarrollador de frameworks a identificar un subconjunto de UML que permite el modelado de los rasgos importantes de un framework.

En segundo lugar, UML-F entrega los mecanismos adicionales requeridos para describir frameworks. Para este propósito, el perfil UML-F adapta el estándar UML, apuntando al diseño y la puesta en práctica de los sistemas de software. No se centra en la ingeniería de requisitos o elementos relacionados. Esto último es la principal razón por la cual ciertos diagramas de UML se dejan fuera de UML-F.

En UML-F, un número de conceptos del UML estándar toman un significado más específico. De forma particular:

- En UML-F, todas las clases son clases de implementación, a diferencia de UML estándar, en donde las clases pueden también denotar los conceptos del mundo real, y que no aparecen necesariamente en una implementación de sistemas software.
- La invocación de métodos son la forma primaria de considerar la interacción entre objetos.

- La composición de objetos se basa en asociaciones con una dependencia adicional sobre los ciclos de vida de los objetos asociados.

Por otro lado, el "principio de la subespecificación" (*underspecification*) se aplica de manera consistente. Este principio es un concepto importante para un lenguaje de modelado, que permite el modelado de características deseadas, y la omisión de los detalles que aún no son entendidos por el desarrollador. Como un framework se diseña para un conjunto de sistemas con una amplia variedad de funcionalidad y comportamientos, el principio de subespecificación es particularmente útil para la descripción de frameworks. Debido a su naturaleza ejecutable, un lenguaje de programación puro no puede ser utilizado para este propósito. Por ejemplo, un lenguaje de programación no se puede emplear para describir la gama de comportamientos previstos de un método que se pueda redefinir en subclases. Una implementación por defecto muestra solamente un comportamiento, dentro de un conjunto de posibles comportamientos. El principio de subespecificación aplicado en el contexto de UML es una herramienta apropiada (ejemplo: diagramas de secuencias con alternativas).

En lo referente al etiquetado, es posible decir que todas las adaptaciones de clases y objetos se deben marcar explícitamente por las etiquetas que correspondan. Estas etiquetas constituyen la pieza de extensión del perfil UML-F, y responden a los siguientes propósitos:

- Algunas etiquetas indican qué es visible en un modelo. Por ejemplo, estas etiquetas expresan si la información presentada en dicho modelo está completa ó incompleta.
- Algunas etiquetas revelan información adicional sobre la clase o el objeto en cuestión. Esto puede ser información sobre la signatura, el comportamiento, las trayectorias de la interacción, o la funcionalidad.
- Una tercera categoría de las etiquetas de UML-F permite al desarrollador de frameworks expresar su intención de establecer la base de las futuras implementaciones. Esto es particularmente útil cuando una clase o interfaz no entregan una implementación, pero proporcionan una signatura que debe ser implementada.

Especificación de etiquetas de UML-F

Etiquetas de Completitud y Jerarquía

Etiqueta	Resumen
©	**Marcador de Completitud** La representación gráfica es completa. Los métodos y/o atributos mostrados de una clase, por ejemplo, son efectivamente todos los métodos y/o atributos que dicha clase posee. Sirve también para indicar que todas las generalizaciones de la clase son incluidas (mostradas), toda la información sobre una asociación es mostrada, etc.
...	**Marcador de Completitud** Indica lo contrario a ©, es decir, posiblemente no todos los métodos o atributos que existen realmente son incluidos en el diagrama, así como no todas las generalizaciones, etc.
δ	**Representación Jerárquica** La representación gráfica muestra sólo los miembros que se han introducido o redefinido al nivel actual. Los miembros heredados no son mostrados.
Λ	**Representación Plana** La representación gráfica muestra los miembros heredados como también los nuevos miembros introducidos o redefinidos. Se muestra la jerarquía completa.

Tabla 3.2 : ***Etiquetas y descripciones para Etiquetas de Completitud y Jerarquía en UML-F***

Indicadores Gráficos Realzados de Herencia

Los siguientes elementos gráficos transportan la información sobre métodos o atributos en un diagrama de clases o diagrama de objetos (ver tabla 3.2). Ellos se unen a un costado de la caja de la clase o del objeto, justo al lado del método o atributo a los cuales se aplican. Si están contenidos dentro de la caja, también indican que el método o el atributo es inaccesible por otros elementos en el diagrama. Si cruzan el límite de la caja de la clase o del objeto, quiere decir que el método o el atributo al que se hace referencia es accesible por otros elementos en el diagrama.

Etiqueta	Resumen
	**Rectángulo sin relleno** Se hereda y no se redefine el método o atributo.
	**Rectángulo gris** El método o atributo es definido nuevamente, o este se hereda pero es completamente redefinido.
	**Rectángulo con mitad gris y mitad blanco** El método es redefinido pero utilizando el método heredado. Esto se efectúa mediante la sentencia super().
	**Rectángulo blanco con línea diagonal** El método es abstracto, y debe ser implementado en las subclases.

Tabla 3.3 : ***Etiquetas y descripciones Indicadores Gráficos Realzados de Herencia en UML-F***

Etiquetas para Diagramas de Secuencia

A través de elementos gráficos, los diagramas de secuencia llegan a ser más expresivos (ver tabla 3.4).

Etiqueta	**Resumen**
*****	Repetición No Controlada Para un mensaje: el mensaje se puede repetir tan a menudo como se desee. Se permiten cero ocurrencias. Para un objeto: pueden haber múltiples instancias del objeto. Se permiten cero ocurrencias.
+	Opcional El mensaje es opcional, ocurriendo a lo más una vez.
|	Alternativas El diagrama de secuencia se subdivide en diagramas alternativos.
(A-B)	Repetición Controlada Una manera consciente de describir la ocurrencia múltiple de un mensaje o de un objeto (entre los tiempos A y B).
<<trigger>>	Trigger Marca el mensaje en un diagrama de secuencia que desencadene una interacción subsiguiente.

Tabla 3.4 : ***Etiquetas y descripciones de Etiquetas para Diagramas de Secuencia en UML-F***

Etiquetas Básicas para Modelado de Frameworks

Las siguientes etiquetas de UML-F muestran características de anotación en el modelado de elementos y son particularmente útiles en el contexto de diseño y documentación de frameworks.

Etiqueta	Resumen
<<**application**>>	El elemento pertenece a la aplicación, no al framework.
<<**framework**>>	El elemento pertenece al framework.
<<**utility**>>	El elemento pertenece a una biblioteca utilitaria o al entorno de ejecución.
<<**fixed**>>	El elemento es fijo. No permite ningún tipo de cambios (alteración de un método, adición de nuevos métodos a una clase, adición de nuevas subclases, etc.).
<<**adapt-static**>>	El elemento se puede adaptar durante el tiempo de diseño. En tiempo de ejecución, el elemento es fijo (no cambia).
<<**adapt-dyn**>>	El elemento se puede cambiar en tiempo de ejecución.

Tabla 3.5 : ***Descripciones de Etiquetas Básicas para Modelado de Frameworks en UML-F***

Etiquetas *template* y *hook*

Etiqueta	Resumen
<<**template**>>	El método es un "método plantilla" o la clase contiene un método plantilla.
<<**hook**>>	El método es un método *hook* (hot spot) o la clase/interface contiene un método hook.

Tabla 3.6 : ***Descripciones de etiquetas esenciales para los principios de construcción de frameworks en UML-F***

Herramientas para el diseño de diagramas UML-F

THOMAS ASCHAUER en su tesis [Asc05], realiza un completo estudio de las herramientas que pudieran ser extendidas para lograr crear diagramas del perfil UML-F. Sus criterios de selección fueron:

- **Criterio I: Visualización de las etiquetas de UML-F** – determina si la herramienta es capaz de mostrar los estereotipos, y las etiquetas únicas de la notación
- **Criterio II: Soporte de grupos de etiquetas** – que la herramienta permita crear, renombrar y eliminar grupos de etiquetas, además de poder visualizarlos
- **Criterio III: Movimiento entre capas** – evalúa si la herramienta permite el ocultar ciertas etiquetas y/o vistas del diagrama UML del framework representado mediante el uso de capas o *layers*
- **Criterio IV: Procedimiento de anotación de paquetes** – determina si la herramienta soporta un procedimiento de anotación de paquetes, considerando su nivel de usabilidad
- **Criterio V: Soporte para patrones específicos del dominio** – evalúa si es posible el crear patrones específicos del dominio, administrarlo, documentarlos, y visualizarlos correctamente
- **Criterio VI: Cumplimiento con UML** – asigna la versión de UML con la cual la herramienta es compatible

- **Criterio VII: Usabilidad** – asigna la usabilidad de la herramienta tanto para usuarios principiantes como para usuarios avanzados
- **Criterio VIII: Estándares Abiertos** – determina si la herramienta es compatible con estándares abiertos, como el *XML Metadata Interchange (XMI)*
- **Criterio IX: Portabilidad** – establece las plataformas de sistema operativo que la herramienta soporta

El autor comprobó cada punto del criterio en las herramientas más comunes en ese momento. Las herramientas estudiadas fueron:

- Rational Rose 2003: una de las herramientas de modelado de UML más completas del mercado, creada por IBM. Ofrece una plataforma de extensibilidad para incorporar perfiles UML creados por el usuario.
- Rational Rose XDE: otra herramienta de IBM, independiente de la línea de productos Rational Rose. Se integra en la plataforma Eclipse de desarrollo. XDE soporta UML 1.4 y XMI 1.3 como formato de intercambio.
- Fujaba Tool Suite: una herramienta open source, desarrollada por la Universidad de Paderborn. Fujaba acepta UML 1.3, pero no todos los diagramas están soportados.
- ArgoUML: herramienta open source creada en la Universidad de California, y ahora está disponible en la comunidad open source. ArgoUML cumple con UML 1.3 de forma completa.

Criterio	Prioridad	Rational Rose	Rose XDE	Fujaba	ArgoUML
I	(1)	X	X	√	√
II	(2)	X	√	√	√
III	(2)	X	√	√	√
IV	(2)	X	√	√	√
V	(2)	X	√	√	√
VI	(3)	1.4	1.4	1.3	1.3
VII	(3)	+	+	-	o
VIII	(4)	o	o	o	+
IX	(4)	Linux, Unix, Windows	Windows	Java	Java
√: realizable X: no realizable +: evaluación positiva -: evaluación negativa o: evaluación neutral					

Tabla 3.7 : Tabla de evaluación de las herramientas CASE según los criterios de evaluación

THOMAS ASCHAUER finalmente escogió ArgoUML como la herramienta a extender para poder modelar diagramas con el perfil UML-F. Sin embargo, la antigüedad de la versión utilizada y su casi nula representación gráfica de etiquetas como los de completitud, hizo que se descartara esa alternativa.

Actualmente, Rational Rose es Rational Software Architect. Se volvió a aplicar los criterios originales, y no hubo cambios en las prestaciones de la nueva versión. Como se mencionó anteriormente, ArgoUML no siguió desarrollándose y hoy en día es una herramienta obsoleta. El autor de esta tesis contactó a THOMAS ASCHAUER para verificar la no existencia en la actualidad de herramientas para el modelado de UML-F.

Dado que lo que se buscaba para este proyecto era el diseñar diagramas UML-F de baja complejidad que implementaran las etiquetas más visuales del perfil como los de completitud, surgió la necesidad de crear una pequeña herramienta que permitiera crear este tipo de diagramas. La herramienta fue desarrollada con la ayuda del Framework, y básicamente funciona añadiendo las etiquetas sobre una imagen de diagramas de clase previamente creados.

4.
Desarrollo del Framework

Considerando el problema establecido anteriormente, se propone el diseño y construcción de un Framework de Software para Java. Básicamente, este Framework proporcionará el código necesario para lograr la creación de aplicaciones con representaciones gráficas de datos, evitando comenzar desde cero su desarrollo.

El Framework ofrecerá lo siguiente:

- Un conjunto de paquetes, lógicamente conectados, con sus clases correspondientes. Todo esto utilizando sólo Java, de manera de evitar tener que recurrir a herramientas externas.
- Un diseño basado en patrones y en las guías de diseño de frameworks.
- Facilidad de uso y aprendizaje para desarrolladores novatos e intermedios.

4.1 Metodología de desarrollo

Por lo general, un framework nace a partir del desarrollo de una aplicación. Se identifican las piezas de código que se repiten, y se extraen de la forma de un componente reutilizable [Cwa06]. DIRK RIEHL propone un desarrollo de frameworks basado en el desarrollo paralelo de aplicaciones de éste.

Como se puede apreciar en la Figura 4.17, el dominio de la aplicación es quien determina el diseño y rediseño del framework. Se puede decir que las aplicaciones son un "cliente" del framework, y generan demanda de funcionalidades y prestaciones, sobretodo en etapas tempranas del desarrollo.

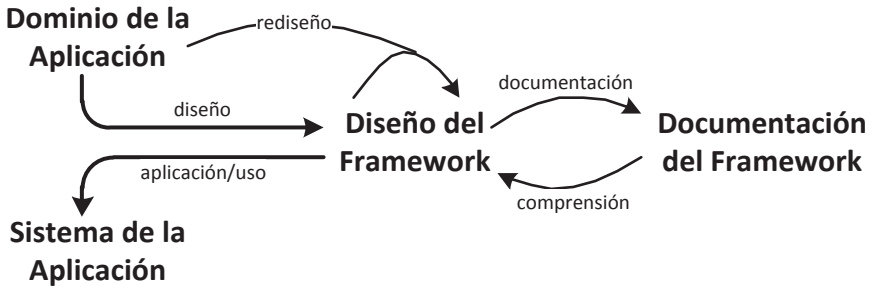

Figura 4.17 : ***Representación de la metodología de desarrollo del framework propuesto***

Este diagrama muestra 4 artefactos: dominio de la aplicación, sistema de la aplicación, diseño del framework y documentación del framework. Muestra claramente la distinción entre artefactos y actividades. Por ejemplo, el diseño del framework es un artefacto, mientras que el diseño es una actividad. De los artefactos, sólo la documentación es completamente tangible. El dominio de la aplicación, el sistema de la aplicación, y el diseño del framework son artefactos parciales o completamente abstractos.

Es importante distinguir entre el diseño y la documentación del framework. El diseño del framework contiene todas las ideas y conceptos involucrados en un framework. La documentación del framework los hace lo más explícitas posible (por ejemplo, utilizando diagramas UML-F), pero nunca es capaz de revelarlos por completo. Más que nada, la documentación describe aquellos aspectos del diseño que son más importantes que el desarrollador conozca para entender el framework.

Los artefactos se relacionan por actividades. Diseñar un sistema en un dominio de aplicación lleva a un diseño. Documentar el diseño lleva a la documentación. Leer la documentación y trabajar en el framework lleva a comprender el diseño del framework. Aplicar el framework lleva a una aplicación. Etc.

El Framework propuesto consiste en proveer una capa sobre las JFC, es decir, se basa en Swing y Java2D para su desarrollo, heredando su estructura de Modelo-Vista-Controlador, y todas sus clases útiles. La idea central de esta nueva capa provista por el Framework, es lograr el proveer de una alternativa simple y rápida para lograr aplicaciones gráficas. Como ilustra la Figura 4.18 ilustra la idea de esta capa, mostrando que principalmente se está extendiendo Swing y su componente elemental, el *JComponent*.

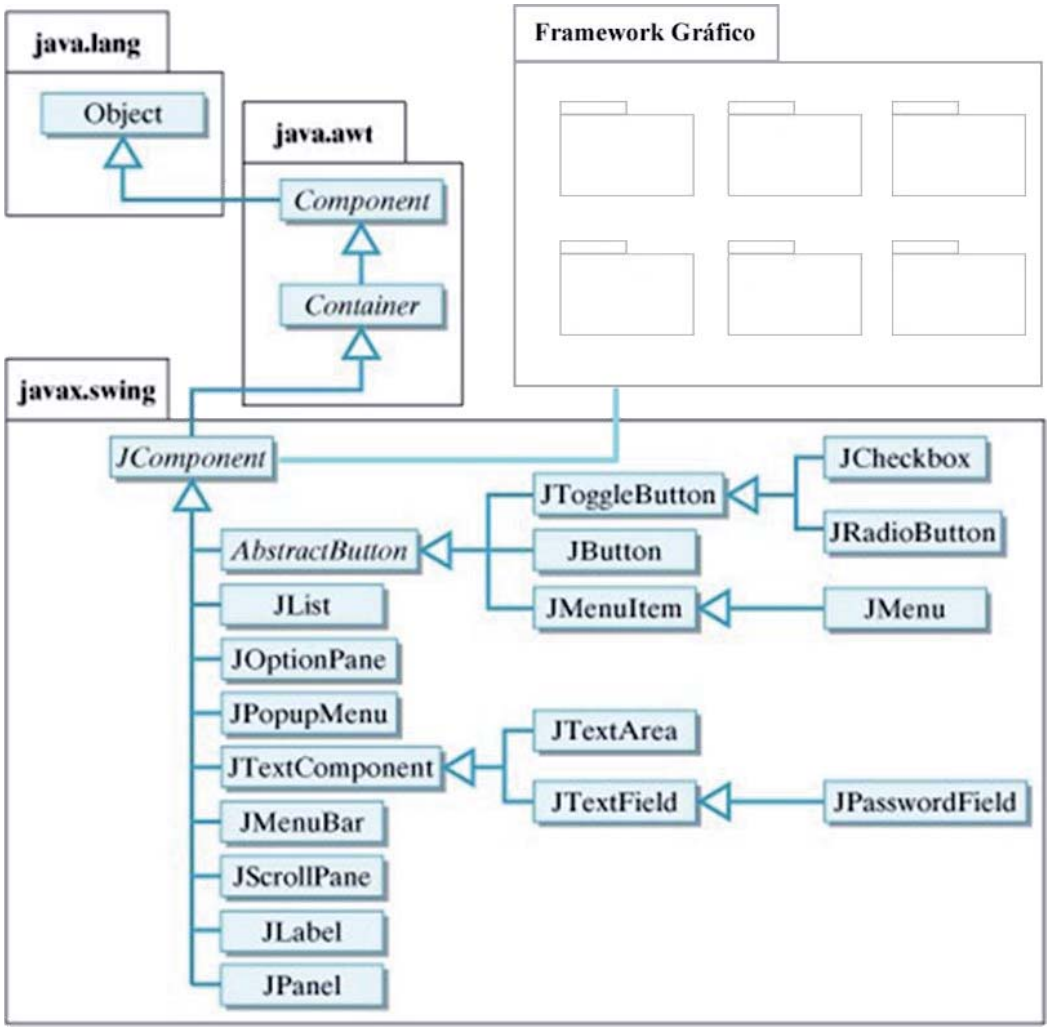

Figura 4.18 : ***El Framework Gráfico (FG) como capa sobre Swing***

4.2 Alcances del Framework

El Framework proporcionará al desarrollador las clases necesarias para lograr el desarrollo simple de aplicaciones con representaciones gráficas interactivas. Se entiende por representación gráfica interactiva a una representación gráfica que no es estática. Esto implica que el usuario final podrá interactuar con ella a través de los dispositivos de entrada. Estas interacciones pueden generar o no cambios en el modelo de datos. El Framework no proporciona modelos de datos (pensando en el patrón Modelo-Vista-Controlador) ya que, como todo framework de software, debe ser siempre independiente de las implementaciones.

El alcance y el tipo de ayuda que dará el Framework, se explican con un ejemplo práctico.

Desarrollo de la solución basada en capas

Planteamiento del problema

Suponer que se está desarrollando una aplicación, la cual contiene dentro de sus clases una clase llamada Tarea, que tiene un sólo atributo de tipo String llamado nombre. Se entiende además que la clase Tarea no es de carácter gráfico, ya que contiene sólo un dato, su nombre. Suponer además que la aplicación maneja un conjunto de Tareas y las relaciones entre ellas, con una clase llamada Relación, que se construye a través de dos tareas, es decir: Relación (Tarea t1, Tarea t2).

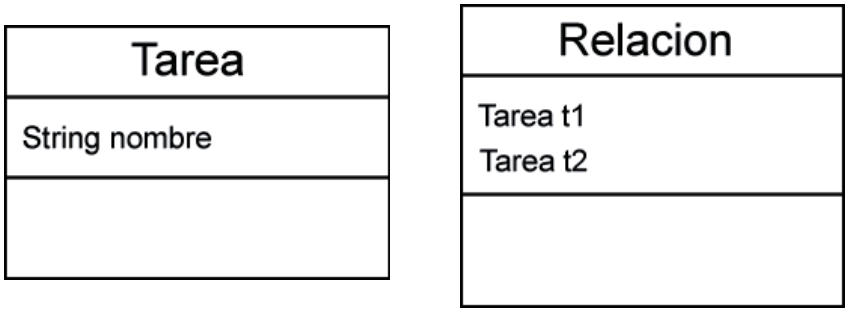

Figura 4.19: ***Clase Tarea y clase Relación.***

En un cierto punto del desarrollo se decide que sería muy útil incorporar una "vista gráfica" en donde las Tareas se representan con un círculo que lleva su nombre escrito, y las relaciones son líneas entre ellas.

Como se vio en el Capítulo 2, para "dibujar" elementos gráficos en Java, se utiliza la API *Java2D*, que consta de dos elementos esenciales: un lápiz para dibujar, y un papel sobre el cual dibujar. Usualmente se utiliza un *JPanel* o *JComponent* como "papel" y el objeto *Graphics* de éste como "lápiz" [Guy07]. Para realizar un dibujo, usualmente se sobrescribe el método *paintComponent(Graphics g)* del *JPanel* o *JComponent*, en donde se escribe una lista de instrucciones secuenciales para producir los resultados gráficos deseados. Por ejemplo, para dibujar una línea que vaya desde el punto (0,0) al punto (30,30) relativos al "papel", se escribe:

```
1     public void paintComponent(Graphics g)
2     {
3         g.drawLine(0, 0, 30, 30);
4     }
5
```

Código 4.3 : ***Dibujo de una línea desde (0,0) a (30,30)***

Volviendo al ejemplo inicial, para dibujar *Tareas*, será necesario crear una *TareaGrafica*, de tal forma de no contaminar la clase *Tarea* con atributos que no correspondan. *TareaGrafica* tendrá 3 elementos: una referencia a un objeto *Tarea*, y dos enteros para representar sus coordenadas *x* e *y*.

Figura 4.20: ***Clase TareaGrafica***

Se plantea la siguiente interrogante: ¿Cómo pintar las líneas que representan las relaciones entre tareas? Como se vio anteriormente, para pintar una línea, basta con tener los dos pares de coordenadas, entonces bastaría con pedir las coordenadas de cada una de las dos tareas que componen la relación y pintar esa línea. No precisamente, o al menos, no directamente, ya que las *Relaciones* contienen dos *Tareas* no dos *TareasGraficas*. Llegar a las coordenadas necesarias requerirá un proceso de ir comparando cada *Tarea* de las relaciones con cada *Tarea* de las *TareasGraficas*. El problema nace al momento de llevar a la *TareaGrafica* una referencia al objeto *Tarea*. Al hacer esto, la representación gráfica queda aislada de cualquier lógica en la que participe este objeto *Tarea*.

La única solución aparente sería hacer que todo lo gráfico fuera un sub-tipo lo que sería dato, o que imitara todos sus comportamientos gráficamente. Claramente las dos formas estarían mal. La primera porque se está obligado en algún momento a tener instanciadas las clases gráficas sin estar necesariamente ocupándolas, lo cual le da más carga a la aplicación, y la segunda porque cualquier cambio en el comportamiento debe realizarse dos veces: una vez en la capa datos y otra vez en la capa gráfica, lo cual lleva inevitablemente a inconsistencias y errores.

Sin duda el ejemplo es simple, sin embargo aparece un problema de complejidad no menor. El Framework apunta entonces a evitar potenciales malas elecciones de arquitectura, generadas a partir de este tipo de problemas.

Volviendo al tema netamente gráfico, suponer que se tiene un conjunto de objetos *Tarea*, y se desea dibujar las representaciones gráficas. Para lograr esto simplemente es necesario realizar una operación de pintado de cada uno de estos objetos. El método básicamente sería como sigue:

```
1     public void paintComponent(Graphics g)
2     {
3         for(TareaGrafica tareaGrafica : tareasGraficas)
4         {
5                 g.drawRect(tareaGrafica.x, tareaGrafica.y, 20, 20);
                  g.drawString(tareaGrafica.getTarea().getNombre(),
                              tareaGrafica.x, tareaGrafica.y);
6         }
7     }
```

Código 4.4 : ***Dibujado de un conjunto de objetos TareaGrafica***

Es claro que este código no representa mucha complejidad, sin embargo, hay que analizar los resultados. Lo que se está haciendo aquí no es más que un "dibujo" de las tareas. Es decir, es algo plano, incapaz de manejar interacciones del usuario, como gestos de mouse (*click*, *drag and drop*, etc.), que es algo deseable en este tipo de representaciones gráficas.

Para lograr interacción, lo más básico que puede hacerse es incorporar un *Listener* de gestos de mouse a nivel del papel, no de las tareas gráficas, debido a que no existen como un objeto que sea capaz de reconocer gestos de mouse. Cada vez que se haga un *click* sobre el papel, habría que realizar lo siguiente:

```
8     public void mouseClicked(MouseEvent e)
9     {
10        for (TareaGrafica tareaGrafica : tareasGraficas)
11              {
12                Rectangle r = new Rectangle(tareaGrafica.x,
13                                            tareaGrafica.y, 20, 20);
14              if(r.contains(e.getPoint()))
15                {
16                    System.out.println("Click sobre la Tarea "+
17                         tareaGrafica.getTarea().getNombre());
18                }
19              }
20    }
```

Código 4.5 : ***Implementación método para capturar clicks de mouse***

Lo que hace este método es preguntar si el punto (x,y) está contenido dentro de alguno de los rectángulos que representan a las tareas. Recorre toda la lista de tareas gráficas preguntando, lo cual claramente es poco óptimo, y todo tipo de interacción deberá ser manejado de forma similar, debido a esta naturaleza de papel-lápiz-dibujo.

Solución al problema

El Framework propuesto soluciona éste y muchos otros problemas del mismo tipo, no utilizando esta naturaleza de papel-lápiz-dibujo previamente establecida, sino que una estructura del tipo papel-papel-dibujo. Si la primera era similar a un papel con dibujos, la segunda es similar a un papel en donde se ponen recortes de figuras hechas en papel, en donde se ubican los dibujos, como lo ilustra la Figura 4.21.

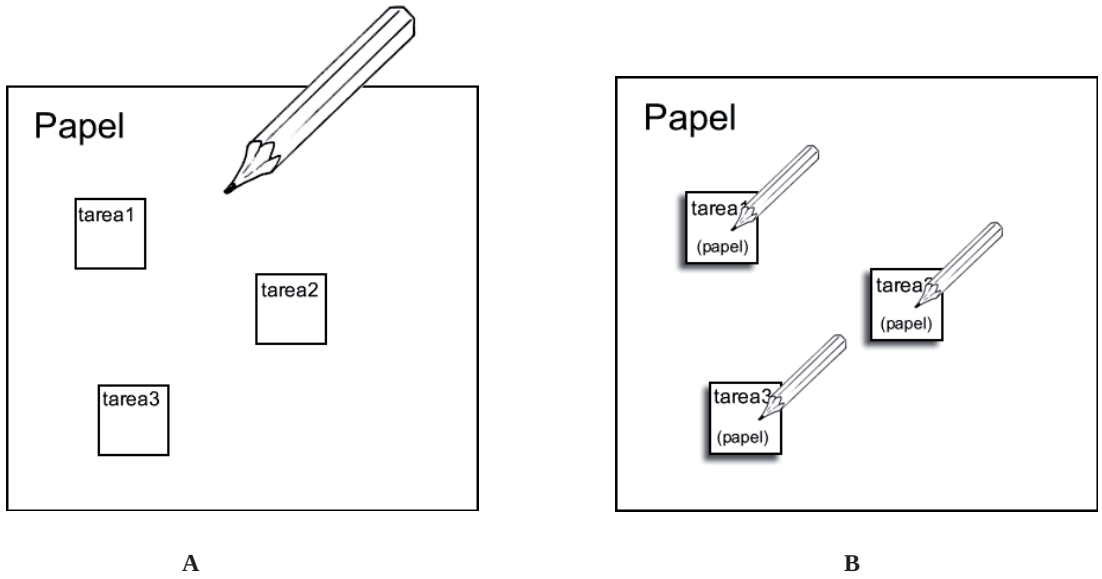

Figura 4.21: ***A) La naturaleza papel-lápiz-dibujo: un papel y un lápiz que dibuja sobre éste B) La naturaleza papel-papel-dibujo: un papel "principal" y un papel por cada representación gráfica con su respectivo lápiz.***

Es fácil notar que al adoptar este enfoque, se solucionan varios problemas que presentaba la primera forma. Uno de éstos, y el más importante, es el de la interacción. Cada papel responde por su cuenta a las interacciones del usuario, ya sean de mouse, teclado, etc.

Es posible crear estos "papeles" con el uso de una de las clases base que conforman el Framework, *AbstractShape*.

Suponer ahora que la clase *TareaGrafica* extiende *AbstractShape*. Esta nueva clase se muestra en la Figura 4.22. Al extender esta clase del Framework, *TareaGrafica* ya no necesita preocuparse

de pintados ni posiciones ni respuestas básicas de interacción, como *clicks* y *drag and drop*. Si se quisiera responder de manera especial a estos gestos, bastaría con sobrescribir los métodos necesarios.

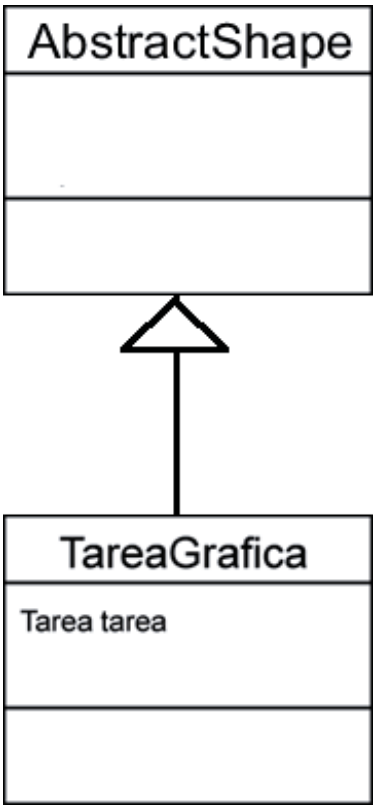

Figura 4.22 : ***La clase TareaGrafica extendiendo a AbstractShape.***

El rol del *papel* "principal" lo realiza una clase que acepte estos objetos *AbstractShape*, y que disponga de controles y operaciones básicas que se harían sobre toda la pantalla, como por ejemplo, el zoom. En el contexto de diagramas, existe la clase *DiagramCanvas* (*canvas* en este caso en el sentido de lienzo de pintor). Esta clase no debería extenderse.

Finalmente, el modelo que cumple con esta arquitectura se muestra en la Figura 4.23. Este modelo también sirve para ilustrar el patrón de diseño Modelo-Vista-Controlador en el contexto del Framework y su delimitación: mantener separados el modelo gráfico con el modelo de datos.

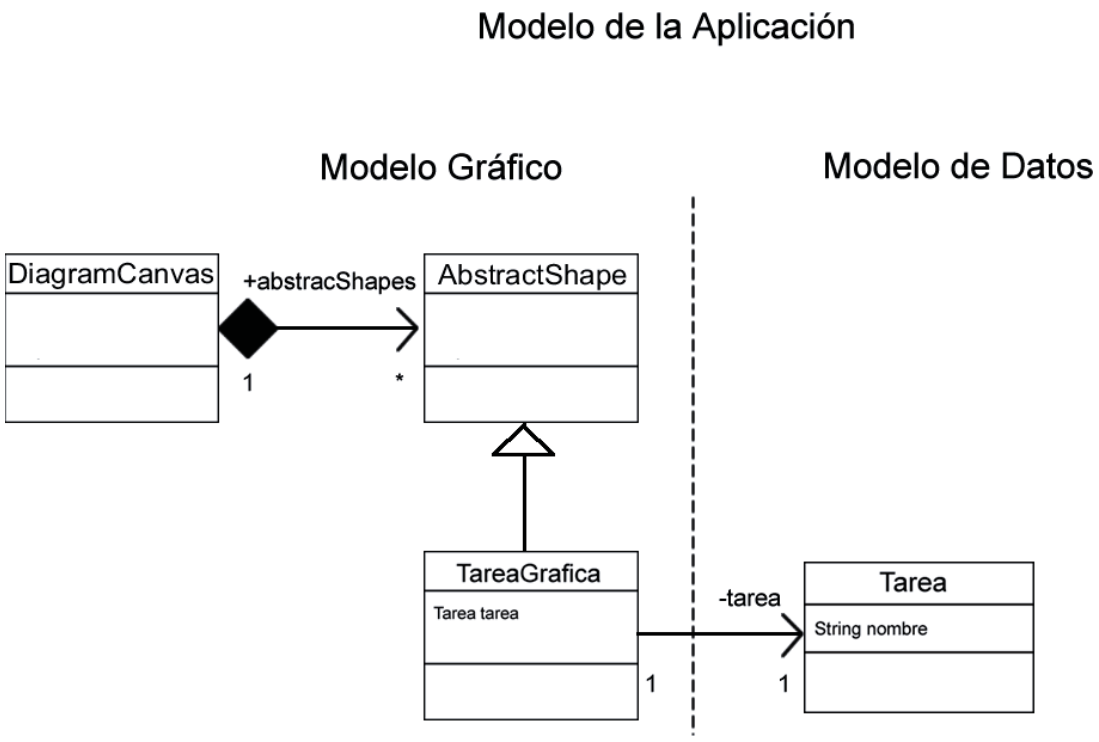

Figura 4.23 : ***Modelo final implementando el Framework para representar gráficamente los objetos Tarea***

Para mostrar la simplicidad de código que aporta el Framework a la aplicación, basta con ver el código necesario para mostrar las tareas gráficas:

```
1     new DiagramCanvas(tareasGraficas);
```

Código 4.6 : ***Construcción de representaciones gráficas de objetos Tarea utilizando el Framework***

Esta línea construye un *DiagramCanvas* junto con un set de tareas gráficas para mostrar inicialmente, utilizando el siguiente constructor:

```
1     public DiagramCanvas(Collection <? extends AbstractShape>)
```

Código 4.7 : ***Constructor utilizado para mostrar las tareas gráficas***

4.3 Diseño general del Framework

Como se vio anteriormente en el ejemplo práctico, el Framework sigue el esquema del patrón Modelo-Vista-Controlador, pero sin preocuparse de los modelos de datos. Estos serían determinados en cada contexto de aplicación que se esté desarrollando con apoyo del Framework.

En su arquitectura básica, el Framework sigue siendo MVC. Pero hay paquetes dentro del Framework que siguen otro tipo de patrones, como es el caso del paquete framework.state, cual se muestra continuación.

Ejemplo: diseño del paquete framework.state

En el contexto de una aplicación de diagramas, siempre existen diversos tipos de opciones dentro del diagrama: se puede dibujar figuras, conectarlas con líneas, insertar texto, etc. El paquete **framework.state** es el encargado de manejar todas las opciones que se presenten en la aplicación, contando con un conjunto de opciones por defecto.

Cada opción estará definida por un objeto *State* (estado): hay estados de dibujado, de selección, etc. El primer paso es crear la clase abstracta que represente a un estado, para así dar lugar a que existan estados creados por el usuario. Esta clase se llama *DiagramState*, como la ilustra la Figura 4.24.

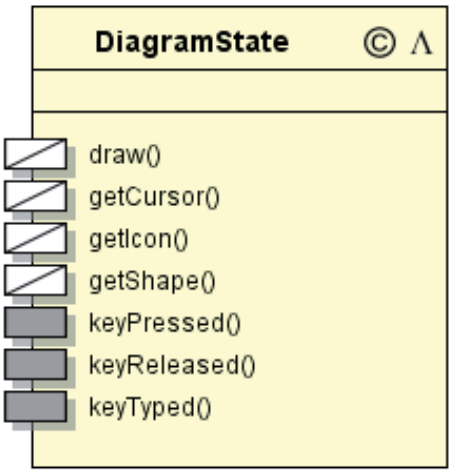

Figura 4.24 : ***Clase DiagramState***

Cada subclase debe darle comportamiento a cada método, incluidos los métodos de la clase *MouseAdapter*, que son métodos de control de eventos generados por el mouse. Todos estos

estados deben ser manejados, administrados, por alguien, por lo cual se crea una clase *Singleton* para tal efecto. Esta clase siempre sabe cuál es el estado actual, y maneja el cambio de un estado a otro. Esta clase se llama *DiagramController,* y que se muestra en la Figura 4.25.

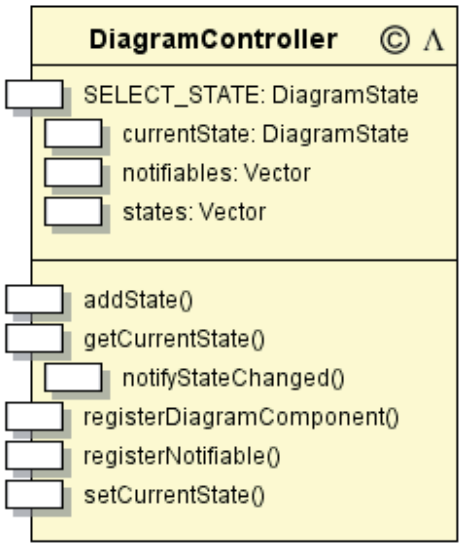

Figura 4.25 : *Clase DiagramController*

Con estos elementos, es posible crear estados básicos y cambiarlos a través del controlador. De los estados básicos que se crearon para formar parte del Framework, para así proveer una base y asegurar el buen comportamiento de ellos, se tiene: *SelectionState*, para la selección de figuras en el diagrama, *DrawLineState*, para el dibujo de líneas en el diagrama, *DrawShapeState*, para el dibujo de figuras en el diagrama, y *DrawTextState*, para el dibujo de texto en el diagrama. En términos de patrones de diseño, el diseño del paquete **framework.state** es una combinación de los patrones *State* y *Strategy*. La Figura 4.26 muestra el diagrama de clases del paquete.

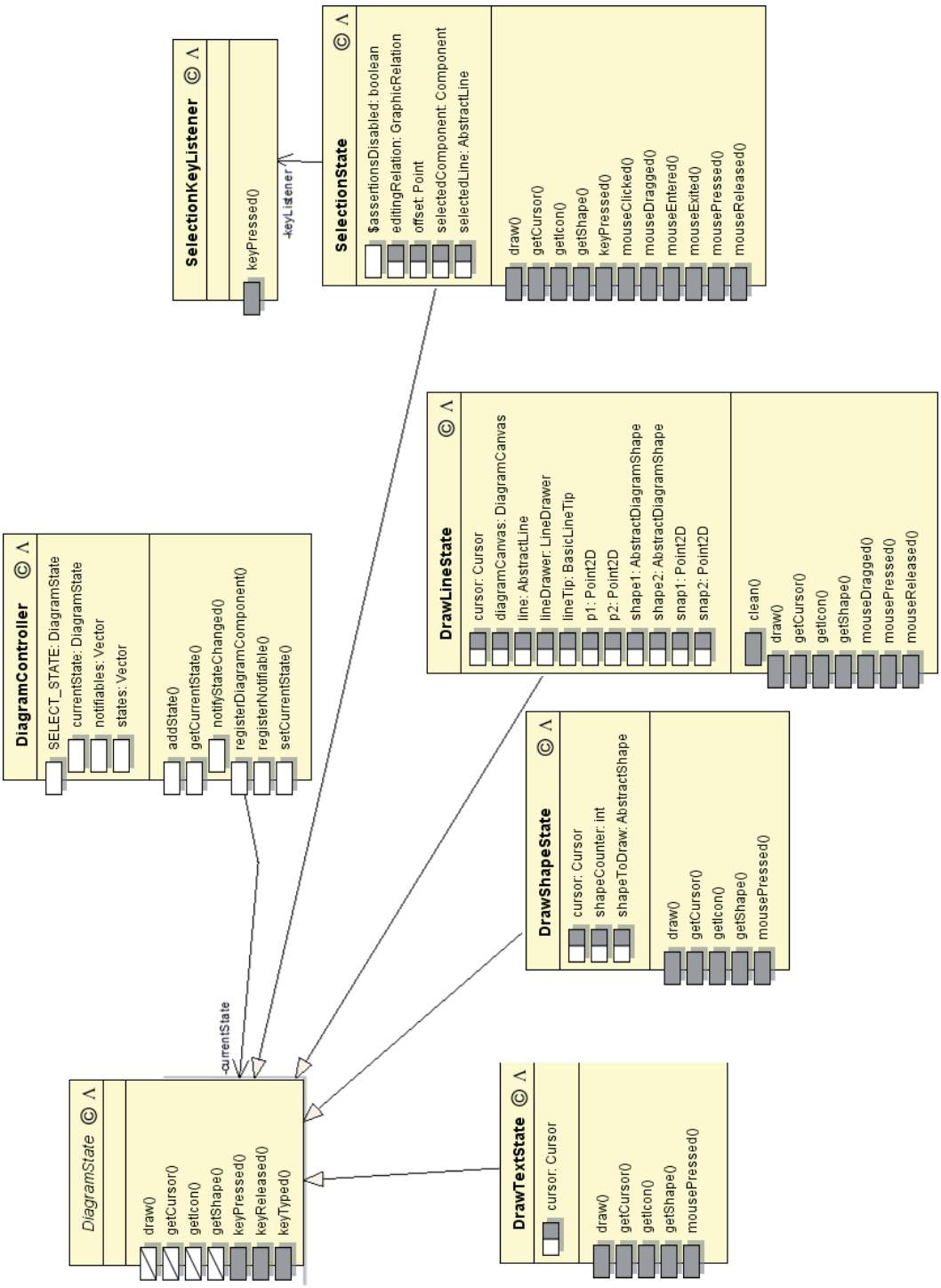

Figura 4.26 : ***Diagrama de Clases paquete framework.state***

Desarrollo de prototipos

Como se mencionó anteriormente, el Framework será desarrollado por prototipos. Se definieron 10 prototipos, desde la versión 0.1 a la versión final entregable 1.0. Cada uno está definido por un hito alcanzado, es decir, una funcionalidad completada. La idea de definir 10 prototipos es para así poder tener objetivos claros en el desarrollo y también para poder ir desarrollando pruebas con desarrolladores, sin tener que esperar la versión final para ello.

Hitos por prototipo:

V0.1: Dibujo de figuras y relaciones, con interacción

V0.2: Exportación de diagramas a imágenes JPG, PNG

V0.3: Persistencia de los diagramas (guardar/cargar)

V0.4: Líneas editables (puntos de control)

V0.5: Líneas con auto-ruta

V0.6: Diagramas con auto-layout

V0.7: Deshacer/Rehacer

V0.8: Bibliotecas de implementaciones pre-hechas

V0.9: Bibliotecas de efectos especiales, y animación

V1.0: Versión entregable

4.4 Estructura del Framework versión 1.0

A continuación se presenta la estructura general el Framework en su versión 1.0. La manera de presentarlo será paquete por paquete, y en cada uno presentando sus clases principales, una breve descripción de su propósito y usos más comunes, además de mostrar un ejemplo de utilización cuando corresponda. Junto a cada presentación del paquete, se muestra además un diagrama de clases de las clases participantes en el paquete.

Los paquetes que componen el Framework son:

- **framework.component**
- **framework.data**
- **framework.diagram**
- **framework.drawer**
- **framework.etc**
- **framework.interfaces**
- **framework.line**
- **framework.listener**
- **framework.persistence**
- **framework.shape**
- **framework.state**
- **framework.toolbar**

Paquete framework.component

Este paquete contiene las clases base del Framework. El término "*component*" se recoge del utilizado en AWT y Swing para describir a sus componentes gráficos. Los componentes del Framework extenderán o tendrán relación con alguna de las clases de este paquete, sobretodo de su clase principal *AbstractComponent*.

Clases más importantes del paquete:

- ***AbstractComponent***: esta clase extiende la clase de Swing *JLayeredPane* para manejar capas, e implementa el *layout SpringLayout*. Esto último permite que se puedan

agregar elementos mediante coordenadas, algo que se logra solo en este contexto, al haber creado los métodos que permiten estos parámetros y logran el objetivo. La Figura 4.27 muestra la estructura de la clase AbstractComponent.

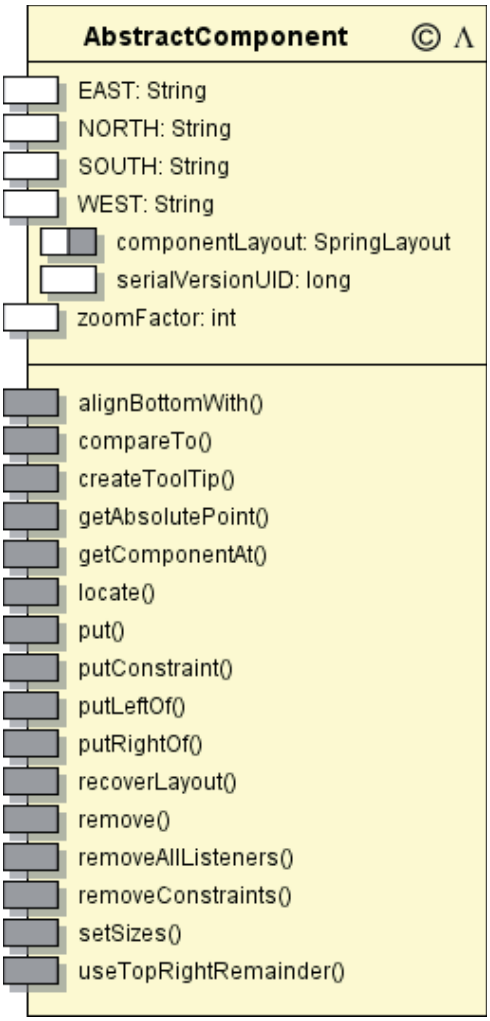

Figura 4.27 : ***La clase AbstractComponent del paquete framework.component***

Paquete framework.diagram

Este paquete contiene las clases necesarias para lograr la creación de diagramas y sus elementos. Considera una jerarquía de figuras y un diagrama que las contiene, administra y controla. Estas clases son *AbstractDiagramShape* y *DiagramCanvas* respectivamente.

Clases más importantes del paquete:

- ***AbstractDiagramShape***: esta clase hereda de la clase *AbstractShape* del paquete **framework.shape**, y añade al dibujo de la figura elementos compatibles con un diagrama, como puntos especiales para ubicar líneas y flechas de conexión. Además implementa notificaciones de cambio de posición, etc. La Figura 4.28 muestra la clase *AbstractDiagramShape*.

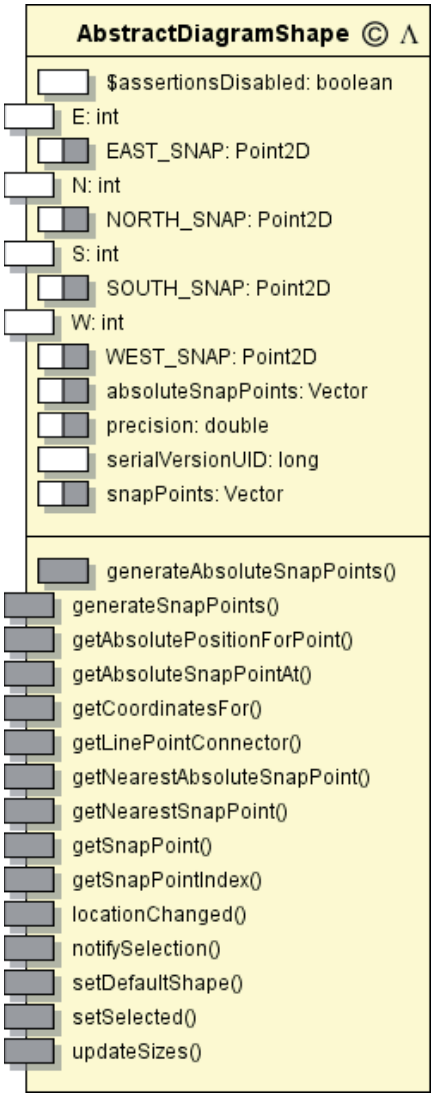

Figura 4.28 : ***La clase AbstractDiagramShape del paquete framework.shape***

- ***DiagramCanvas***: es una de las clases más importantes del Framework. Unifica y maneja la mayoría de los elementos en juego en un diagrama. Contiene las figuras, administra la interacción, etc. Esta clase es un *AbstractComponent*. La Figura 4.29 muestra la estructura de esta clase (se muestran sólo los miembros públicos).

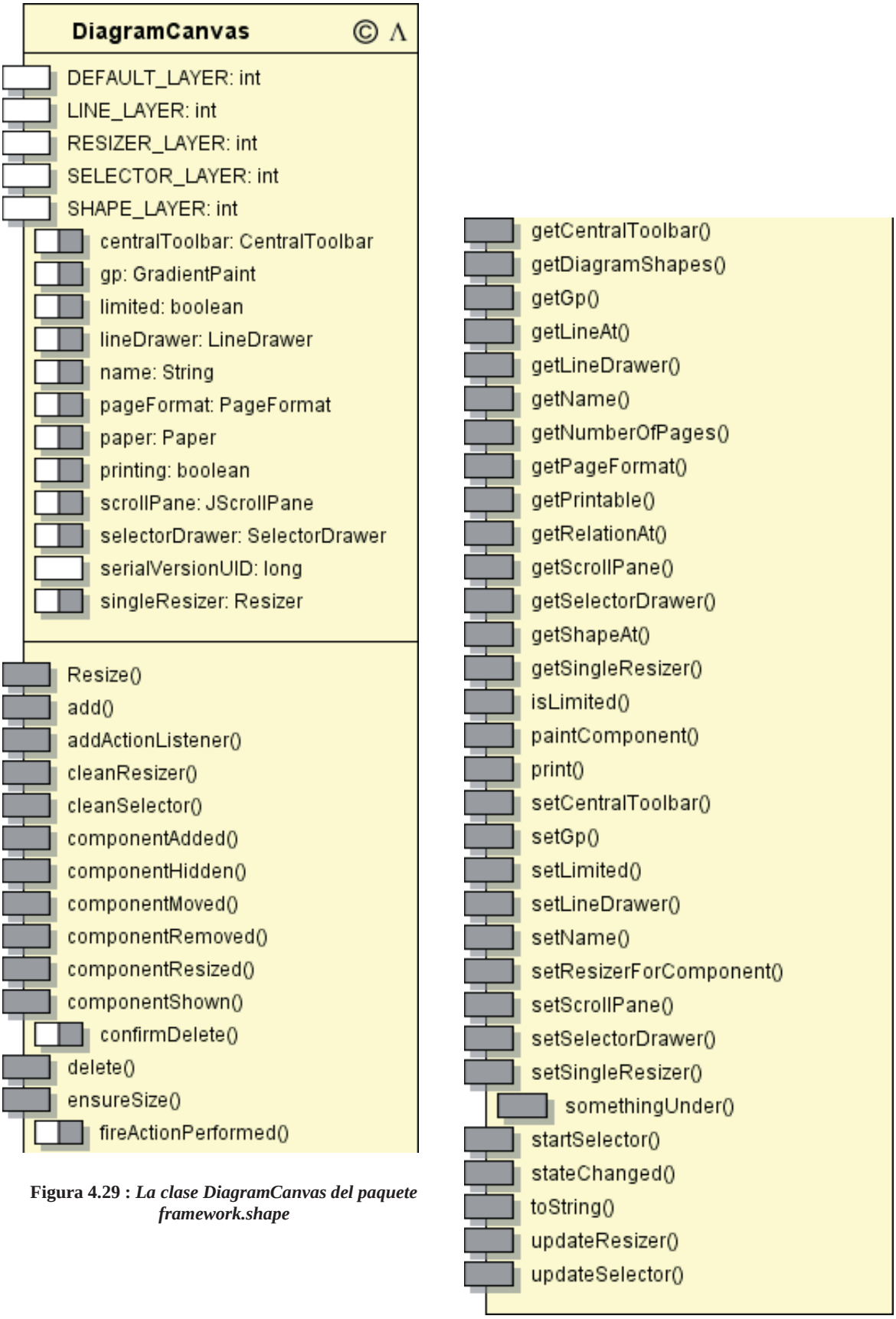

Figura 4.29 : ***La clase DiagramCanvas del paquete framework.shape***

Paquete framework.interface

Este paquete está compuesto por las interfaces que determinarán ciertas propiedades para los elementos gráficos del Framework. Al implementarlas, el componente adquiere las propiedades que la interfaz proporciona. A continuación se presenta una lista de ellas, y en la Figura 4.30 se muestran sus estructuras:

- ***Deletable***: quien implemente esta interfaz declara a quien corresponda (un *DiagramCanvas* por ejemplo) que es borrable, es decir, se puede eliminar del diagrama o contenedor.
- ***DoubleClickable***: implementada cuando se quiere que el elemento gráfico tenga un comportamiento específico cuando se realiza un doble *click* sobre él.
- ***Editable***: especifica que el elemento puede ser editado (sus propiedades).
- ***Manipulable***: implica que la figura puede ser manipulada de alguna manera.
- ***Moveable***: la figura puede ser movida con el mouse.
- ***Resizable***: determina que quien la implemente se le podrá alterar el tamaño.
- ***Selectable***: si se implementa, se puede seleccionar el elemento, si no, la figura queda como un dibujo plano.
- ***Snapper***: la clase posee un efecto de "magnetismo".
- ***StateNotifiable***: en un contexto de estados (se verá en el paquete **framework.state**), implica que la clase que lo implemente necesita ser notificada de un cambio de estado.

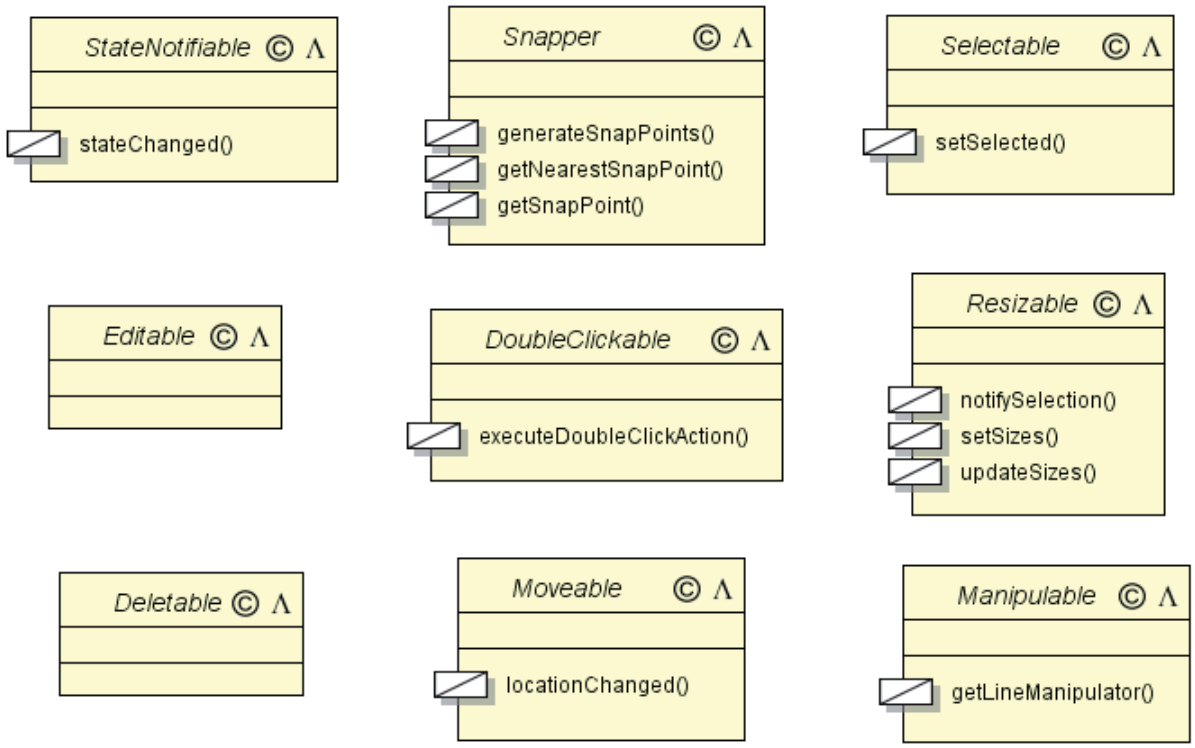

Figura 4.30 : ***Interfaces del paquete framework.interfaces***

Paquete framework.shape

Este paquete está compuesto por una sola clase, AbstractShape. Esta clase es la base de cualquier figura que quiera ser dibujada en el Framework. Es base de la metodología presentada previamente de "papel con su propio lápiz". La estructura de la clase se puede apreciar en la Figura 4.31. En el Anexo A, se muestra el código de la clase AbstractDiagramShape, la cual extiende AbstractShape para contextos de diagramas (por ejemplo, diagramas UML).

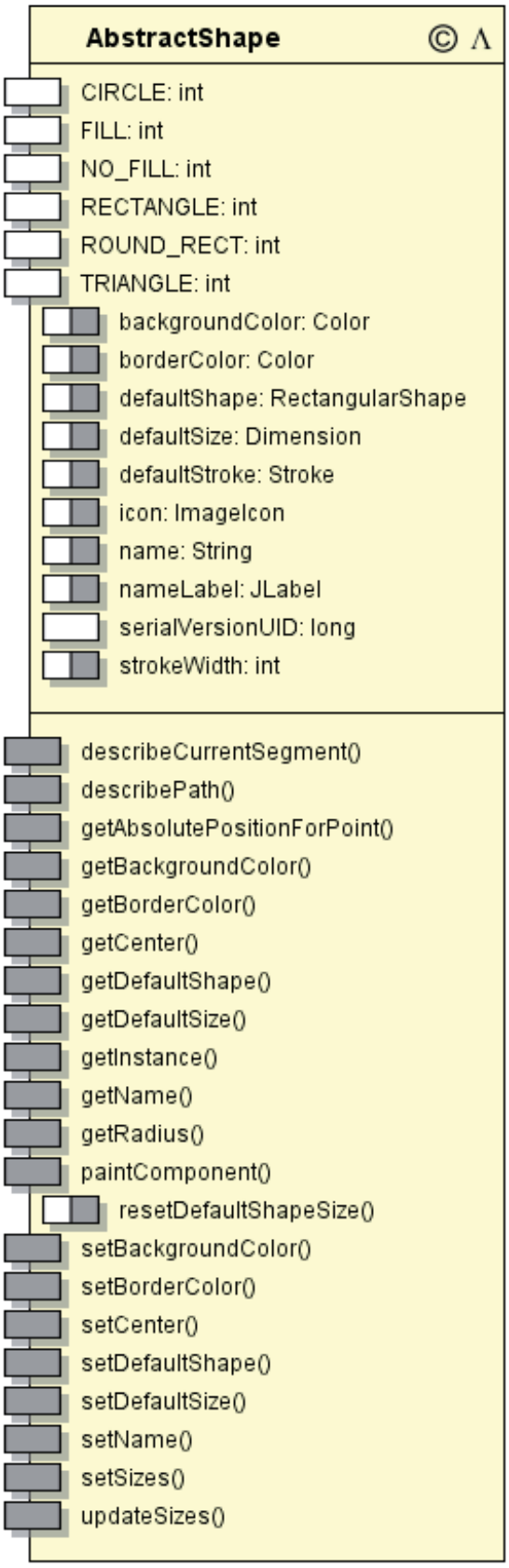

Figura 4.31 : ***La clase AbstractShape del paquete framework.shape***

Otros paquetes

- **framework.state**: presentado en la sección 4.3
- **framework.data**: paquete que contiene clases que permiten relacionar modelos de datos con modelos gráficos de manera holgada.
- **framework.drawer**: paquete que contiene clases de "dibujadores", como el dibujador de las líneas conectoras entre figuras, LineDrawer, o el dibujador del rectángulo punteado de selección de múltiples figuras.

framework.line: paquete que contiene las clases que representan líneas, además de contener algunas implementaciones "típicas" como la línea punteada, línea con punta de flecha, etc.

5.

Pruebas Sobre el Framework

Se realizó una prueba del Framework con desarrolladores Java, para comprobar su eficacia para lograr la creación de gráficos interactivos en Java de manera simple, rápida y limpia, con pocas líneas de código. Para esto se diseñó un ejercicio de desarrollo, junto con un cuestionario pre y post-test. Se realizó una misma prueba de control sin utilizar el Framework, sólo utilizando Swing, de manera de obtener resultados de control, y contrastarlos con los del Framework.

5.1 Objetivos de las pruebas

Las pruebas con usuario tienen como objetivo:

- Comprobar el nivel de dificultad que tiene el lograr desarrollar gráficos interactivos utilizando sólo Swing.
- Verificar que el Framework es entendible por usuarios que lo utilizan por primera vez.
- Comprobar que las clases y métodos del Framework son bien asimilados por los usuarios y se utilizan para lo que están diseñadas.
- Demostrar que, de lograrse el resultado esperado del test (el programa), el código obtenido **sin** el uso del Framework es proporcionalmente mayor en líneas de código y no es extensible, comparado con el obtenido **con** el uso del Framework

5.2 Diseño de las pruebas

En esta sección se presenta un resumen del diseño de las pruebas realizadas. En el Anexo C se encuentra el diseño completo y detallado.

Ambas pruebas estaban formadas por 3 secciones: cuestionario pre-test, test, y cuestionario post-test. Los cuestionarios pre-test de las dos pruebas son idénticos, y tienen como fin el determinar el nivel de experiencia del usuario en Java, Swing y AWT en general.

El test consiste en pedirle al usuario que genere el código necesario para obtener una ventana cuyo contenido fuera un rectángulo rojo con borde negro, conectado a un círculo azul sin borde a través de una línea punteada. Además, se especifica que debe implementarse de forma tal que ambas figuras puedan ser movidas con el mouse, y que la conexión entre ambas se mantenga. La Figura 5.32 muestra lo que se esperaba como resultado.

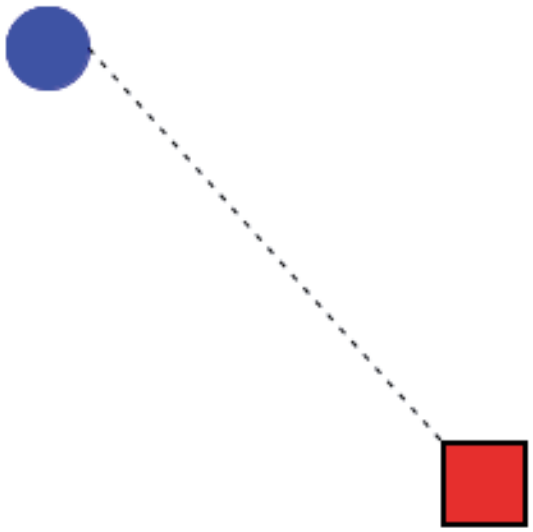

Figura 5.32 : ***Resultado esperado de la prueba***

El resultado es posible de lograr utilizando Swing y el Framework. Para ello se construyó cada versión, de manera de verificar que se lograra, y a su vez medir el mínimo de clases y líneas de código necesarias para llegar al resultado. En resumen, se espera que los resultados sean:

- Con el uso Swing:
  - Entre 5 y 10 clases
  - Entre 300 y 1000 líneas de código
- Con el uso del Framework:
  - Entre 2 y 6 clases
  - Entre 50 y 200 líneas de código

La Figura 5.33 muestra capturas de pantallas ambos casos.

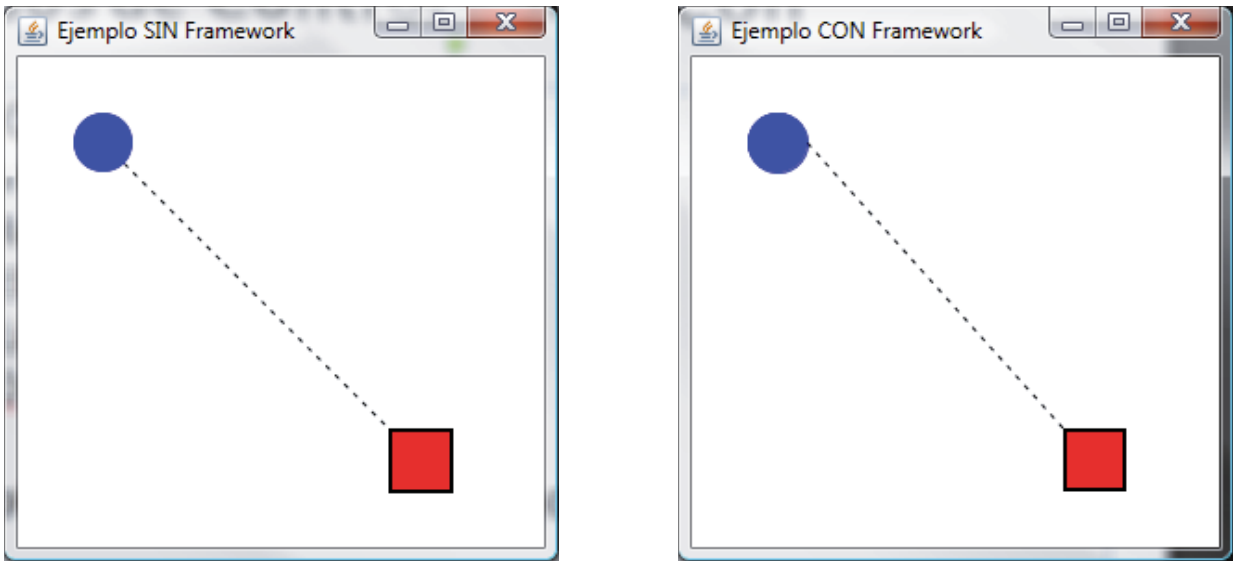

Figura 5.33 : ***Captura de pantalla aplicaciones de control, con y sin Framework***

Los cuestionarios post-test de ambas pruebas contiene preguntas comunes al inicio, y luego preguntas específicas de la solución implementada. En el caso de Swing se le pregunta sobre los métodos y técnicas utilizadas, junto con el nivel de completitud que el usuario estima que logró con respecto a lo requerido. En el caso de la prueba con el Framework, se le preguntó sobre las clases que utilizó para obtener el resultado, de tal manera de verificar que no utilizó clases que no correspondían para lograr ciertos resultados, si no que utilizó las que estaban destinadas para ello. Además, se le dio un espacio a los programadores para comentar sobre el Framework, sus ventajas o falencias.

5.3 Análisis de resultados

En esta sección se analizarán los resultados de las pruebas. Para ver los resultados en detalle, ver el Anexo D.

Las pruebas fueron realizadas por dos grupos de 4 personas cada uno, sumando un total de 8 participantes. Todos cumplían con tener una cierta experiencia en desarrollar en Java.

Para las dos pruebas, se analizarán los resultados en cuanto a la información que se extrajo de los cuestionarios, como perfiles de los usuarios y nivel de conocimientos. El test se evalúa en cuanto al código entregado y su nivel de completitud y nivel de complejidad.

Resultados prueba Swing (control)

Nivel de experiencia de los usuarios con Java: El nivel de experiencia en años desarrollando con Java fue variado, de tal forma que se pudo comprobar la complejidad con distintos niveles de experiencia en el desarrollo de aplicaciones. Los porcentajes fueron:

- Menos de 1 año: 50% (2)
- 2 años (25%) (1)
- Más de 2 años (25%) (1)

Conocimientos de Swing: La primera pregunta fue contestada correctamente por el 100% de los usuarios: JFrame es la clase más adecuada para construir una ventana en Java.

Conocimientos de dibujo en Java:

- El 75% de los usuarios contestó erradamente la pregunta sobre qué clase se utiliza para contener dibujos dentro de una ventana, marcando la alternativa JCanvas, la cual no existe. El resto contestó correctamente JComponent, sin embargo Canvas era más esperada como alternativa correcta.
- En la pregunta sobre qué método se utiliza para dibujar, el 75% contestó correctamente.
- Sobre la clase que realiza los dibujos, un 50% marcó Graphics como alternativa, la cual es correcta pero no es Java2D. El otro 50% marcó Graphics2D, la cual era la alternativa más correcta y permite funcionalidades de Java2D.
- Cuando se les preguntó sobre qué clase es más adecuada para dibujar un rectángulo, el 50% marcó la alternativa más adecuada, Rectangle2D.
- El 50% de los usuarios escogió erróneamente al método refreshContents(), método que no existe, como el método encargado de actualizar los contenidos de la pantalla. El otro 50% contestó correctamente la alternativa repaint().

Análisis del código entregado: Ningún usuario logró el objetivo requerido. Los distintos niveles de complejidad establecidos en orden creciente eran:

- Dibujar las figuras en pantalla: 100% logró esto
- Mover las figuras en pantalla: 50% logró esto
- Unir las figuras mediante la línea: 25% logró esto, pero de manera incompleta
- Mantener la unión entre las figuras después del movimiento: un 0% logró esto

Ninguno de los usuarios logró el objetivo utilizando sólo Swing. Se demostró que lo complejo no es dibujar gráficos en la pantalla, si no que añadirles la interacción con el mouse. En Código 5.8 se muestra el código creado por el desarrollador que más se acercó al resultado requerido. La Figura 5.34 muestra el resultado de la pantalla lograda, la cual se acercó bastante a lo esperado.

```
1     public class ACircle extends JPanel
2     {
3           private static final long serialVersionUID = 1L;
4           public static final int WIDTH = 110;
5           public static final int HEIGHT = 110;
6
7           private Ellipse2D.Double circle = new Ellipse2D.Double(0, 0,
      WIDTH, HEIGHT);
8
9           public void paintComponent(Graphics g)
10          {
11                super.paintComponent(g);
12                Graphics2D g2d = (Graphics2D)g;
13                g2d.setStroke(new BasicStroke(2f));
14                g2d.setColor(Color.BLUE);
15                g2d.fill(circle);
16          }
17    }

18    public class ALine extends JComponent
19    {
20          private static final long serialVersionUID = 1L;
21          int ix, iy, fx, fy;
22
23          public void paintComponent(Graphics g)
24          {
25                super.paintComponent(g);
26                Graphics2D g2d = (Graphics2D)g;
27                g2d.setStroke(new BasicStroke(2.0f, BasicStroke.CAP_BUTT,
      BasicStroke.JOIN_BEVEL, 1.0f, new float[]{8.0f, 3.0f,
28                            8.0f, 3.0f}, 0.0f));
29                g2d.setColor(Color.ORANGE);
30                g2d.drawLine(ix, iy, fx, fy);
31          }
32
33          public void setPoints(int ix, int iy, int fx, int fy)
34          {
35                this.ix = ix;
36                this.iy = iy;
37                this.fx = fx;
38                this.fy = fy;
39          }
40    }

41    public class ARectangle extends JPanel
42    {
43          private static final long serialVersionUID = 1L;
44          public static final int WIDTH = 100;
45          public static final int HEIGHT = 100;
46
47          private Rectangle2D.Double square = new Rectangle2D.Double(0, 0,
      WIDTH, HEIGHT);
48          private     Rectangle2D.Double     squareOutline     =     new
      Rectangle2D.Double(0, 0, WIDTH, HEIGHT);
49
50          public void paintComponent(Graphics g)
51          {
```

El programador hace un buen uso de las primitivas de Java2D, en este caso la clase Ellipse2D

Utiliza correctamente el método paint() para mostrar los elementos aráficos.

```
52                super.paintComponent(g);
53                Graphics2D g2d = (Graphics2D)g;
54                g2d.setStroke(new BasicStroke(4f));
55
56                g2d.setColor(Color.RED);
57                g2d.fill(square);
58
59                g2d.setColor(Color.BLACK);
60                g2d.draw(squareOutline);
61
62          }
63    }

64    public class MouseHandler extends MouseAdapter
65    {
66          private JComponent comp1;
67          private JComponent comp2;
68          private JComponent moving;
69          private ALine line;
70          private JFrame frame;
71
72          public   MouseHandler(JFrame   frame,   JComponent   component1,
      JComponent component2, ALine line)
73          {
74                this.comp1 = component1;
75                this.comp2 = component2;
76                this.line = line;
77                this.frame = frame;
78          }
79
80          @Override
81          public void mousePressed(MouseEvent e)
82          {
83                Point mouse = (Point)e.getPoint().clone();
84                Point figure = comp1.getLocation();
85                mouse.translate(-figure.x, -figure.y);
86                if(comp1.contains(mouse))
87                {
88                      moving = comp1;
89                      System.out.println("El 1");
90                      return;
91                }
92
93                mouse = e.getPoint();
94                figure = comp2.getLocation();
95                mouse.translate(-figure.x, -figure.y);
96                if(comp2.contains(mouse))
97                {
98                      moving = comp2;
99                      System.out.println("El 2");
100                     return;
101               }
102               else
103               {
104                     System.out.println("na");
105                     moving = null;
106               }
```

Se puede apreciar en este clase MouseHandler, lo complejo que es el lograr un simple selector de figuras con arrastre. Es un código abstracto, y este caso, no funciona correctamente.

```
107         }
108
109         @Override
110         public void mouseDragged(MouseEvent e)
111         {
112               if(moving != null)
113               {
114                     Point figure = moving.getLocation();
115                     moving.setBounds(e.getX() - figure.x / 2, e.getY() -
      figure.y / 2, moving.getWidth(), moving.getHeight());
116
117                     Point center1 = comp1.getLocation();
118                     center1.translate(comp1.getWidth()        /        2,
      comp1.getHeight() / 2);
119                     Point center2 = comp2.getLocation();
120                     center2.translate(comp2.getWidth()        /        2,
      comp2.getHeight() / 2);
121
122                     int ix, iy, fx, fy;
123                     if(center1.getX() <= center2.getX())
124                     {
125                           if(center1.getY() <= center2.getY())
126                           {
127                                 ix = (int)Math.round(center1.getX());
128                                 fx = (int)Math.round(center2.getX());
129                                 iy = (int)Math.round(center1.getY());
130                                 fy = (int)Math.round(center2.getY());
131                           }
132                           else
133                           {
134                                 ix = (int)Math.round(center2.getX());
135                                 fx = (int)Math.round(center1.getX());
136                                 iy = (int)Math.round(center2.getY());
137                                 fy = (int)Math.round(center1.getY());
138                           }
139                     }
140                     else
141                     {
142                           if(center1.getY() <= center2.getY())
143                           {
144                                 ix = (int)Math.round(center2.getX());
145                                 fx = (int)Math.round(center1.getX());
146                                 iy = (int)Math.round(center2.getY());
147                                 fy = (int)Math.round(center1.getY());
148                           }
149                           else
150                           {
151                                 ix = (int)Math.round(center2.getX());
152                                 fx = (int)Math.round(center1.getX());
153                                 iy = (int)Math.round(center2.getY());
154                                 fy = (int)Math.round(center1.getY());
155                           }
156                     }
157                     line.setPoints(ix, iy, fx, fy);
158                     frame.repaint();
159               }
160         }
```

```
161   }

162   public class Start
163   {
164         private static final int FS = 800;
165
166         public static void main(String[] args)
167         {
168               JFrame frame = new JFrame();
169               frame.setLayout(null);
170               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
171               frame.setSize(FS, FS);
172
173               ALine line = new ALine();
174               line.setPoints(100, 100, 210, 210);
175               line.setBounds(0, 0, FS, FS);
176               frame.add(line);
177
178               ARectangle rect = new ARectangle();
179               frame.add(rect);
180               rect.setBounds(50,          50,          ARectangle.WIDTH,
      ARectangle.WIDTH);
181
182               ACircle circle = new ACircle();
183               frame.add(circle);
184               circle.setBounds(150,    150,    ACircle.WIDTH    +    1,
      ACircle.WIDTH + 1);
185
186               MouseHandler  mh  =  new  MouseHandler(frame,  circle,  rect,
      line);
187               frame.addMouseListener(mh);
188               frame.addMouseMotionListener(mh);
189
190               frame.setVisible(true);
191         }
192   }
```

Código 5.8 : ***Código prueba Swing más cercana a lo esperado***

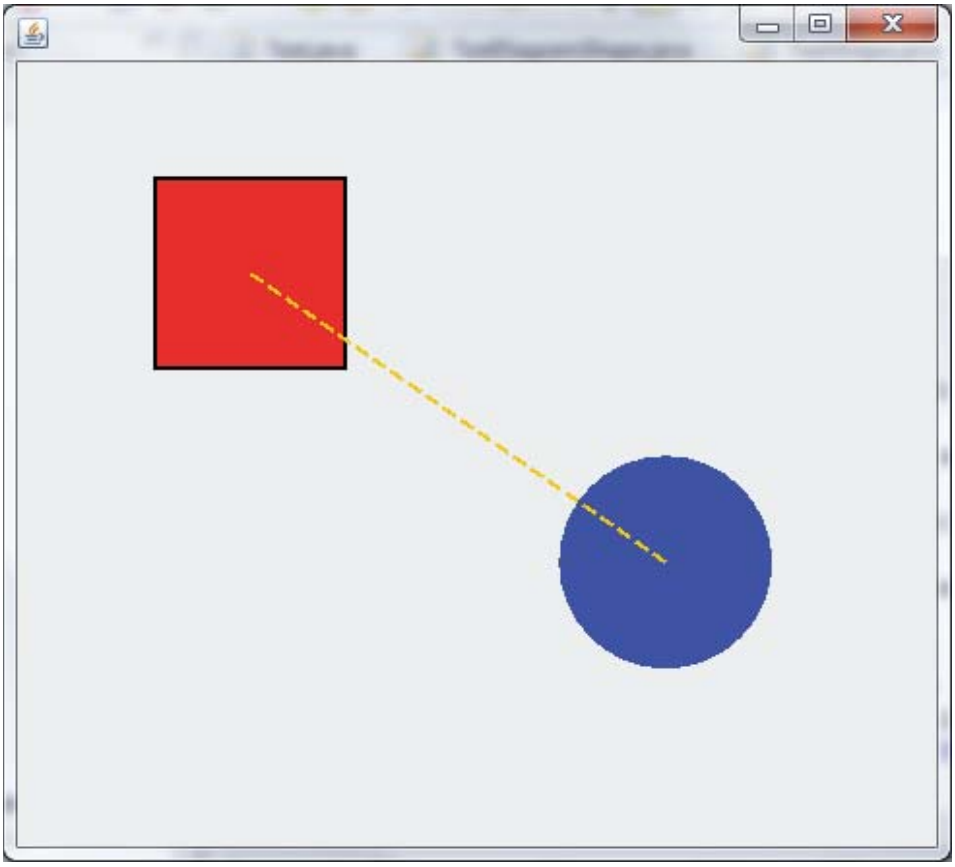

Figura 5.34 : ***Captura de pantalla mejor resultado en prueba Swing***

Análisis de respuestas en post-test: Las primeras preguntas del cuestionario post-test eran las mismas del cuestionario pre-test, y el objetivo de esto era ver si quienes se equivocaron en el pre-test, luego de su experiencia de desarrollo del test, habían aprendido. Sin embargo, se repitieron exactamente los mismos resultados, por lo que se puede concluir que no hubo mayor aprendizaje de Swing con la experiencia de desarrollar la prueba.

Interacción con el mouse: Sólo el 25% pudo contestar esta pregunta, el mismo 25% que logró obtener interacción. El 75% restante declaró no haber logrado la interacción, como era de esperarse.

Pintar colores: El 100% marcó correctamente la alternativa Color.RED para pintar de color rojo.

Complejidad: El 50% declaró no haber logrado los resultados, otro 25% encontró complejo lograr los resultados, y el restante encontró poco complejo, siendo éste el usuario que más logró o se acercó a lo requerido.

Extensibilidad: El 100% consideró que su código resultante no era fácilmente extensible (por ejemplo, agregar otro rectángulo o línea).

Resultados prueba Framework

Nivel de experiencia de los usuarios con Java: El nivel de experiencia en años desarrollando con Java fue variado, de tal forma que se pudo comprobar la complejidad con distintos niveles de experiencia en el desarrollo de aplicaciones. Para esta prueba se buscaba un nivel un poco más experimentado de los usuarios. Los porcentajes fueron:

- 2 años (25%) (1)
- Más de 2 años (75%) (3)

Conocimientos de Swing: La primera pregunta fue contestada correctamente por el 50% de los usuarios: JFrame es la clase más adecuada para construir una ventana en Java. El otro 50% marcó Frame, que es medianamente correcta.

Conocimientos de dibujo en Java:

- El 50% de los usuarios contestó erradamente la pregunta sobre qué clase se utiliza para contener dibujos dentro de una ventana, marcando la alternativa JCanvas, la cual no existe. El resto contestó correctamente Canvas.
- En la pregunta sobre qué método se utiliza para dibujar, el 75% contestó correctamente.
- Sobre la clase que realiza los dibujos, un 25% marcó Graphics como alternativa, la cual es correcta pero no es Java2D. El otro 50% marcó Graphics2D, la cual era la alternativa más correcta y permite funcionalidades de Java2D. El otro 25% erró marcando Drawer, que no existe.
- Cuando se les preguntó sobre qué clase es más adecuada para dibujar un rectángulo, el 25% marcó la alternativa más adecuada, Rectangle2D.
- El 25% de los usuarios escogió erróneamente al método repaintAllContents(), método que no existe, como el método encargado de actualizar los contenidos de la pantalla. El otro 75% contestó correctamente la alternativa repaint().

Análisis del código entregado: el 75% de los usuarios logró el objetivo. Los distintos niveles de complejidad establecidos en orden creciente eran:

- Dibujar las figuras en pantalla: 100% logró esto
- Mover las figuras en pantalla: 100% logró esto
- Unir las figuras mediante la línea: 75% logró esto
- Mantener la unión entre las figuras después del movimiento: un 50% logró esto

Quedó mostrado por los resultados que el dibujo y la interacción son creados de manera simple y directa. Sin embargo la unión entre figuras fue un poco más obscura para algunos usuarios.

Análisis de respuestas en post-test: Las primeras preguntas del cuestionario post-test eran las mismas del cuestionario pre-test, y el objetivo de esto era ver si quienes se equivocaron en el pre-test, luego de su experiencia de desarrollo del test, habían aprendido. Sin embargo, se repitieron exactamente los mismos resultados, por lo que se puede concluir que no hubo mayor aprendizaje de Swing con la experiencia de desarrollar la prueba.

Interacción con el mouse: Sólo el 25% utilizó el MouseAdapter. El 75% restante declaró no haber hecho nada para lograrlo, lo que está bien ya que el Framework automáticamente añade interacción con el mouse.

Pintar colores: El 75% marcó correctamente la alternativa Color.RED para pintar de color rojo.

Complejidad: El 25% declaró no haber logrado los resultados, otro 25% encontró complejo lograr los resultados, y el restante encontró poco complejo.

Extensibilidad: El 25% consideró que su código resultante no era fácilmente extensible (por ejemplo, agregar otro rectángulo o línea).

En Código 5.9 se muestra el código creado por el desarrollador que más se acercó al resultado requerido. La Figura 5.35 muestra el resultado de la pantalla lograda, la cual se acercó bastante a lo esperado.

```
1     public class DummyTest {
2
3           /**
4            * @param args
5            */
6           public static void main(String[] args) {
7                 // TODO Auto-generated method stub
8                 JFrame myFrame = new JFrame("TesT Framework");
9                 myFrame.setBounds(0, 0, 500, 600);
10                DiagramCanvas myCanvas = new DiagramCanvas("test");
11                AbstractDiagramShape myShape = new AbstractDiagramShape();
12                myShape.setBorderColor(Color.BLACK);
13                myShape.setBackground(Color.RED);
14                myShape.setBackgroundColor(Color.RED);
15                AbstractDiagramShape        myCircle        =        new
      AbstractDiagramShape();
16                myCircle.setBackground(Color.BLUE);
17                myCircle.setBackgroundColor(Color.BLUE);
18                myCircle.setBorder(null);
19
20                PointyLine myLine = new PointyLine(myShape.getCenter(),
21
22                            myCircle.getCenter());
23
24
25                myCircle.setDefaultShape(new
      Ellipse2D.Double(0d,0d,30d,30d));
26                myShape.setDefaultShape(new
      Rectangle2D.Double(0d,0d,30d,30d));
27
28                myCanvas.add(myShape, 40 , 40 );
29                myCanvas.add(myCircle, 50 , 50 );
30                myCanvas.getLineDrawer().addRelation(myCircle,    myShape,
      myLine,   new   EmptyLineTip()   ,   myCircle.getLinePointConnector(),
      myShape.getLinePointConnector());
31                myFrame.add(myCanvas);
32                myFrame.setVisible(true);
33          }
34
35    }
```

Confunde PointyLine con DashedLine, por lo tanto no obtiene la línea requerida

El desarrollador hizo un buen uso general de las clases provistas por el Framework, sin embargo cometió un pequeño error al unir la línea con un borde del círculo y no con

Código 5.9 : ***Código prueba Framework más cercana a lo esperado***

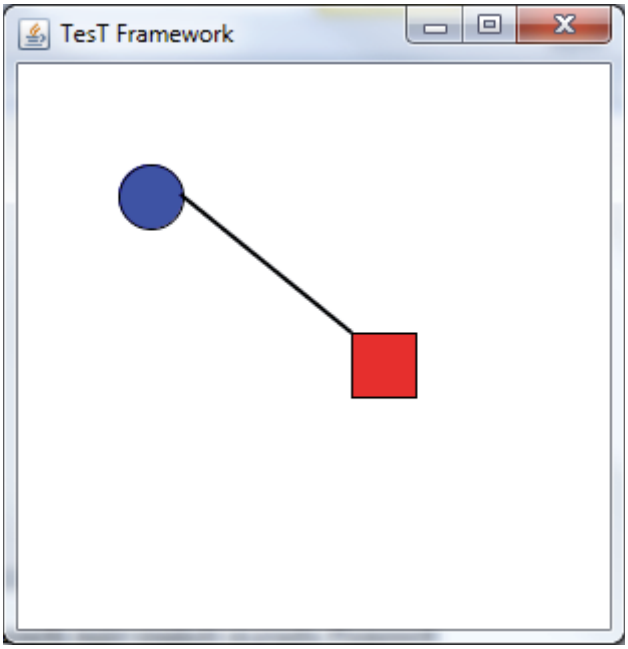

Figura 5.35 : ***Captura de pantalla mejor resultado en prueba Framework***

En Código 5.10 está el código de referencia. Se puede ver que el código generado por el desarrollador en la prueba no dista mucho de lo esperado. Es claro que la cantidad de líneas de código (menos de 40) para lograr un resultado cercano al requerido es mucho menor que en el caso de Swing (más de 200).

```
1     public class VentanaLite extends JFrame
2     {
3           public VentanaLite()
4           {
5                 super("Ejemplo CON Framework");
6                 this.setSize(300, 300);
7                 DiagramCanvas dc = new DiagramCanvas();
8                 this.getContentPane().add(dc);
9                 AbstractDiagramShape r = new AbstractDiagramShape();
10                r.setDefaultShape(new
                                Rectangle2D.Double(0.0,0.0,30.0,30.0));
11                r.setBackgroundColor(Color.RED);
12                r.setBorder(BorderFactory.createLineBorder(
                                                    Color.BLACK, 2));
13                dc.add(r,200,200);
14                AbstractDiagramShape e = new AbstractDiagramShape();
15                e.setDefaultShape(
                            new Ellipse2D.Double(0.0,0.0,30.0,30.0));
16                e.setBackgroundColor(Color.BLUE);
17                e.setBorderColor(Color.BLUE);
18                dc.add(e,30,30);
19                dc.getLineDrawer().addRelation(
                                    r, e, new DashedLine(), null,
20              r.getAbsoluteSnapPointAt(0),
21              e.getAbsoluteSnapPointAt(0));
22          }
23
24
25
26    public static void main(String[] args)
27          {
28                VentanaLite v = new VentanaLite();
29                v.setVisible(true);
30                v.setLocationRelativeTo(null);
31          }
32    }
```

getAbsoluteSnapPointAt(0) o getCenter(), retornan el centro de cada figura

Código 5.10 : ***Código necesario para completar el test con el Framework***

Mejoras del Framework posteriores a las pruebas

Una de las principales observaciones hechas sobre el Framework fue su alta interdependencia entre los componentes. Esto tiene como consecuencia que el programa falle si no se relacionan los objetos que correspondan entre ellos. Otro aspecto observado fue que mucho de los métodos heredados de Swing no funcionan o no hacen lo que se esperaría, lo que se presta para confusiones.

El primer problema se solucionó sólo parcialmente, donde realmente había dependencia entre objetos sin que fuera justificado o útil. La parcialidad se debe a que el Framework debe mantener un cierto nivel de interdependencia para poder lograr el ser rápido y cómo de implementar.

El segundo problema se solucionó creando *overrides* (“redefiniciones”) sobre los métodos de Swing que no estuvieran haciendo lo que corresponde. Con este *override* se puede atrapar las llamadas al método y realizar la tarea esperada por el usuario.

6. Conclusiones

El uso de frameworks de software como apoyo en el desarrollo de aplicaciones complejas es cada día más común. Su utilización agiliza el proceso de codificación del software, y proporciona funcionalidades robustas y probadas. El desarrollo de sistemas de gran tamaño (más de 100.000 líneas de código) se ve beneficiado por la construcción y uso interno de frameworks de software. Ejemplos claros de ello son la plataforma Eclipse y Netbeans.

Desarrollar aplicaciones con representaciones gráficas no es una tarea menor: el código necesario es extenso y complejo, y muy propenso a errores. La única forma de evitar estos problemas es contando con un desarrollador experto en el área dentro del equipo de programadores. Sin embargo, el tiempo necesario para lograr los resultados esperados sigue siendo el mayor factor disuasivo.

Java y sus bibliotecas gráficas permiten realizar virtualmente cualquier elemento gráfico. El costo de esto se traduce en clases complejas de bajo nivel. Para que un programador en Java logre obtener los resultados gráficos deseados, es necesario estudiar, aprender y practicar el uso de estas bibliotecas, cuya curva de aprendizaje varía entre 1 y 2 años.

Una solución propuesta para aminorar la curva de aprendizaje y atenuar la complejidad de Java Swing, es el crear una capa superior. Esta capa, desarrollada entre Swing y la aplicación final, permitiría obtener aplicaciones gráficas con menos código y sin requerir conocimientos sobre Swing y el uso de sus componentes. Además, esta capa debe estar desarrollada como un framework de software, de manera de ser realmente una base sobre la cual construir aplicaciones gráficas.

Hasta el día de hoy dicha solución no se ha desarrollado. Sin embargo, existen alternativas paralelas a Java, que proponen una arquitectura propia, independiente de Swing, como el Graphical Editing Framework de Eclipse. Este Framework, si bien proporciona más herramientas y apoyo para desarrolladores novatos, no utiliza Java Swing, y es por ello que no se considera como una solución al problema planteado.

Fue el objetivo de este trabajo el diseñar y desarrollar un framework de software que fuera capaz de proporcionar una capa sobre Swing, por el medio de la cual los desarrolladores pudieran lograr resultados gráficos mediante un código simple y fácil de mantener y modificar.

El Framework final, bajo la perspectiva de las cualidades de un framework bien diseñado establecidas en la sección 3.2, logra cumplir con:

- ***Simplicidad***: si bien parte del código del Framework es relativamente compleja en cuanto a las funcionalidades que incorpora, esta parte no es visible por el desarrollador (son elementos privados). El resto del Framework y sus clases extensibles por el programador siguen siendo simples y bien documentadas. Sin embargo, existen complejidades inevitables que requieren un poco más de comprensión del desarrollador sobre los mecanismos del Framework, como lo es el uso de las clases del paquete **framework.diagram**.

- ***Costo***: el desarrollo del Framework fue de manera iterativa, pausado, durante un periodo de 3 años, con diferentes versiones y cambios durante su ciclo de vida.

- ***Compromiso***: inicialmente hubo funcionalidades propuestas para el Framework que tuvieron que ser sacadas, ya que afectaban el carácter genérico de este. Un caso particular fue la funcionalidad de motor de videojuego y la capacidad de mostrar animaciones gráficas. Ambas requerían cambios en la arquitectura base del Framework, y "ensuciaban" el resto de las clases más genéricas.

- ***Elementos del pasado***: como se planteó en las primeras secciones, el Framework nace a partir del desarrollo de un módulo gráfico de un software.

- ***Evolutivo***: el Framework se dividió en versiones o prototipos, donde cada uno lograba incorporar una nueva funcionalidad. No se trató de completar todas las funcionalidades simultáneamente en una misma versión.

- ***Consistente***: la consistencia del Framework está asegurada principalmente por el hecho que fue desarrollado por un solo profesional, Reinaldo Espejo. Sin embargo, esto no es suficiente. Con la ayuda de las implementaciones, fue posible ir atacando los diversos problemas que surgieron de la falta de consistencia parcial entre ciertas clases del Framework.

El Framework fue diseñado y desarrollado basándose en una metodología que se apoya fuertemente en aplicaciones del Framework. Estas aplicaciones cumplen un rol de "cliente", y proporcionan una retroalimentación y validación para el Framework, y además generan nuevos requerimientos.

El Framework resultante de este trabajo, logró cumplir con los objetivos establecidos para él. Uno de los factores determinantes en su éxito fue el hecho de contar con un entorno de desarrollo donde siempre hubo proyectos que requirieran de representaciones y elementos gráficos.

En términos de eficacia del código obtenido al utilizar el Framework, se puede decir con seguridad que el "ahorro" para el desarrollador puede ser de varios miles de líneas de código, dado un software de complejidad relativamente mediana. Eficacia en este contexto, se entiende como el lograr resultados de manera más simple y rápida, desde el punto de vista del código fuente. Esta eficacia se hace manifiesta cuando se compara con el código necesario en Swing para lograr resultados similares, como lo mostraron las pruebas en una menor escala.

El uso del Framework a nivel general no está definido oficialmente. Hasta el 2011, el Framework está sólo disponible bajo una licencia académica, para uso dentro de la PUCV en proyectos de desarrollo con fines académicos. El Framework podría ser presentado a la comunidad internacional de desarrolladores de Java con un poco más de trabajo para estandarizar y optimizar ciertos segmentos de este. Sin embargo, el masificar el Framework implica un costo de tiempo por parte del autor para mantenerlo, corregir bugs, y mejorar ciertas funcionalidades, asumiendo que existiría un *feedback* constante por parte de los usuarios.

La demanda por aplicaciones visualmente ricas ha ido en aumento, pero la tendencia se inclina más hacia aplicaciones web (cliente-servidor) que hacia aplicaciones de escritorio (ejecutables), que es donde se desenvuelve el Framework. Es una consideración que a futuro, se realice un equivalente al Framework para aplicaciones web.

7. Referencias

[Ada05]

C. Adamson. *Swing Hacks*. O'Reilly, 2005.

[Asc05]

Thomas Aschauer. *Design Pattern Annotation in a UML-Tool,* Paris-Lodron Universitat Salzburg, 2005.

[Cwa06]

Krzysztof Cwalina, Brad Abrams. *Framework Design Guidelines*, Addison-Wesley Professional, 2006.

[Fon01]

Marcus Fontoura. *The UML Profile for Framework Architectures*, Addison-Wesley, 2001.

[Gam94]

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley Professional, 1994.

[Guy07]

Romain Guy, Chet Haase. *Filthy Rich Clients: Developing Animated and Graphical Effects for Desktop Java Applications,* Prentice Hall, 2007.

[IBM04]

Bill Moore. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*, IBM Corp., 2004.

[Rie00]

Dirk Riehle. *Framework Design: A Role Modeling Approach*. Ph.D. Thesis, No. 13509. Zürich, Switzerland, ETH Zürich, 2000.

[Top10]

Kim Topley. *JavaFX Developer's Guide*, Addison-Wesley Professional, 2010.

[Tul08]

Jaroslav Tulach. *Practical API Design: Confessions of a Java Framework Architect*, Apress, 2008.

[*App10*] (Visitada en Diciembre del 2010)

Appframework, Swing Application Framework.

http://java.net/projects/appframework/

[*Fon04*] (Visitada en Octubre del 2009)

Proyecto D04I1428. *ARQUITECTURA CONFIGURABLE PARA LA OPTIMIZACION DE LA LOGISTICA DE PRODUCCION EN LA INDUSTRIA DE PROCESOS POR LOTES*, 2004. http://www.fondef.cl/bases/fondef/PROYECTO/04/I/D04I1428.HTML

[*Pop11*] (Visitada en Marzo del 2011)

Programming Language Popularity, 2011.

http://www.langpop.com/

[*Pot06*] (Visitada en Septiembre del 2009)

Alexander Potochkin. *JSR 296: Swing Application Framework*, 2006.
http://jcp.org/en/jsr/detail?id=296

[*Pot09*] (Visitada en Marzo del 2011)

Alexander Potochkin. *SAF and JDK7*, 2009.
http://weblogs.java.net/blog/archive/2009/08/19/saf-and-jdk7

[*Tio11*] (Visitada en Abril del 2011)

TIOBE Software. *TIOBE Programming Community Index for March 2011*.
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

8. Anexos

Anexo A.

Muestra de Código Fuente

El propósito de este anexo es mostrar el código fuente de una clase representativa del framework. La clase elegida es *AbstractDiagramShape*. En el código a continuación, se puede ver lo siguiente:

- Aplicación de la licencia *Java Research License*. Esta licencia debe ir al comienzo de cada clase, obligatoriamente (líneas 1-173).
- El uso de otros componentes del framework, en este caso las interfaces *Deletable*, *Editable*, *Moveable*, *Selectable*, *Snapper*, *Resizable* (líneas 202-203).
- Código para ciertas funcionalidades complejas, como lo es la determinación del contorno de la figura para establecer puntos con "magnetismo" (donde una flecha se "pega" a un componente) (líneas 287-388).
- Comentarios compatibles para compilación de Javadoc (líneas 195-202).

Es obligatorio citar la licencia a la cual está sujeto el código fuente, como comentario inicial de cada archivo fuente.

```
/*

JAVA RESEARCH LICENSE
Version 1.6


I.    DEFINITIONS.

"Licensee" means You and any other party that has entered into and has
in effect a version of this License.

"Modifications" means any change or addition to the Technology.

"Sun" means Sun Microsystems, Inc. and its successors and assignees.

"Research Use" means research, evaluation, or development for the
purpose of advancing knowledge, teaching, learning, or customizing the
Technology or Modifications for personal use. Research Use expressly
excludes use or distribution for direct or indirect commercial
(including strategic) gain or advantage.

"Technology" means the source code and object code of the technology
made available by Sun pursuant to this License.

"Technology Site" means the website designated by Sun for accessing
the Technology.

"You" means the individual executing this License or the legal entity
or entities represented by the individual executing this License.

II.     PURPOSE.

Sun is licensing the Technology under this Java Research License (the
"License") to promote research, education, innovation, and development
using the Technology.  This License is not intended to permit or
enable access to the Technology for active consultation as part of
creating an independent implementation of the Technology.

COMMERCIAL USE AND DISTRIBUTION OF TECHNOLOGY AND MODIFICATIONS IS
PERMITTED ONLY UNDER A SUN COMMERCIAL LICENSE.

III.     RESEARCH USE RIGHTS.

A.    License Grant. Subject to the conditions contained herein, Sun
grants to You a non-exclusive, non-transferable, worldwide, and
royalty-free license to do the following for Your Research Use only:

1.    Reproduce, create Modifications of,  and use the Technology
alone, or with Modifications;

2.    Share source code of the Technology alone, or with
Modifications, with other Licensees; and

3.    Distribute object code of the Technology, alone, or with
Modifications, to any third parties for Research Use only, under a
license of Your choice that is consistent with this License; and
```

```
89    publish papers and books discussing the Technology which may include
90    relevant excerpts that do not in the aggregate constitute a
91    significant portion of the Technology.
92
93    B.     Residual Rights.  If You examine the Technology after accepting
94    this License and remember anything about it later, You are not
95    "tainted" in a way that would prevent You from creating or
96    contributing to an independent implementation, but this License grants
97    You no rights to Sun's copyrights or patents for use in such an
98    implementation.
99
100   C.    No Implied Licenses.  Other than the rights granted herein, Sun
101   retains all rights, title, and interest in Technology, and You retain
102   all rights, title, and interest in Your Modifications and associated
103   specifications, subject to the terms of this License.
104
105   D.    Third Party Software.  Portions of the Technology may be
106   provided with licenses or other notices from third parties that govern
107   the use of those portions. Any licenses granted hereunder do not alter
108   any rights and obligations You may have under such licenses, however,
109   the disclaimer of warranty and limitation of liability provisions in
110   this License will apply to all Technology in this distribution.
111
112   IV.    INTELLECTUAL PROPERTY REQUIREMENTS
113
114   As a condition to Your License, You agree to comply with the following
115   restrictions and responsibilities:
116
117   A.    License and Copyright Notices. You must include a copy of this
118   Java Research License in a Readme file for any Technology or
119   Modifications you distribute. You must also include the following
120   statement, "Use and distribution of this technology is subject to the
121   Java Research License included herein", (a) once prominently in the
122   source code tree and/or specifications for Your source code
123   distributions, and (b) once in the same file as Your copyright or
124   proprietary notices for Your binary code distributions. You must cause
125   any files containing Your Modification to carry prominent notice
126   stating that You changed the files. You must not remove or alter any
127   copyright or other proprietary notices in the Technology.
128
129   B.    Licensee Exchanges.    Any Technology and Modifications You
130   receive from any Licensee are governed by this License.
131
132   V.    GENERAL TERMS.
133
134   A.    Disclaimer Of Warranties.
135
136   THE TECHNOLOGY IS PROVIDED "AS IS", WITHOUT WARRANTIES OF ANY KIND,
137   EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, WARRANTIES
138   THAT THE TECHNOLOGY IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A
139   PARTICULAR PURPOSE, OR NON-INFRINGING OF THIRD PARTY RIGHTS.  YOU
140   AGREE THAT YOU BEAR THE ENTIRE RISK IN CONNECTION WITH YOUR USE AND
141   DISTRIBUTION OF ANY AND ALL TECHNOLOGY  UNDER THIS LICENSE.
142
143   B.    Infringement; Limitation Of Liability.
144
```

```
1.    If any portion of, or functionality implemented by, the
Technology becomes the subject of a claim or threatened claim of
infringement ("Affected Materials"), Sun may, in its unrestricted
discretion, suspend Your rights to use and distribute the Affected
Materials under this License. Such suspension of rights will be
effective immediately upon Sun's posting of notice of suspension on
the Technology Site.

2.    IN NO EVENT WILL SUN BE LIABLE FOR ANY DIRECT, INDIRECT,
PUNITIVE, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION
WITH OR ARISING OUT OF THIS LICENSE (INCLUDING, WITHOUT LIMITATION,
LOSS OF PROFITS, USE, DATA, OR ECONOMIC ADVANTAGE OF ANY SORT),
HOWEVER IT ARISES AND ON ANY THEORY OF LIABILITY (including
negligence), WHETHER OR NOT SUN HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.  LIABILITY UNDER THIS SECTION V.B.2 SHALL BE SO LIMITED
AND EXCLUDED, NOTWITHSTANDING FAILURE OF THE ESSENTIAL PURPOSE OF ANY
REMEDY.

C.     Termination.

1.    You may terminate this License at any time by notifying Sun in a
writing addressed to Sun Microsystems, Inc., 4150 Network Circle,
Santa Clara, California 95054, Attn.: Legal Department/Products and
Technology Law.

2.    All Your rights will terminate under this License if You fail to
comply with any of its material terms or conditions and do not cure
such failure within thirty (30) days after becoming aware of such
noncompliance.

3.    Upon termination, You must discontinue all uses and distribution
under this agreement, and all provisions of this Section V ("General
Terms") shall survive termination.

D.     Miscellaneous.


1.    Trademark.  You agree to comply with Sun's Trademark & Logo
Usage Requirements, as modified from time to time, available at
http://www.sun.com/policies/trademarks/.  Except as expressly provided
in this License, You are granted no rights in or to any Sun trademarks
now or hereafter used or licensed by Sun.

2.    Integration.  This License represents the complete agreement of
the parties concerning the subject matter hereof.

3.    Severability.  If any provision of this License is held
unenforceable, such provision shall be reformed to the extent
necessary to make it enforceable unless to do so would defeat the
intent of the parties, in which case, this License shall terminate.

4.    Governing Law.  This License is governed by the laws of the
United States and the State of California, as applied to contracts
entered into and performed in California between California residents.
In no event shall this License be construed against the drafter.

```

```
201   5.    Export Control.   As further described at
202   http://www.sun.com/its, you agree to comply with the U.S. export
203   controls and trade laws of other countries that apply to Technology
204   and Modifications.
205    */
206
207   package framework.diagram;
208
209
210   import java.awt.Color;
211   import java.awt.Graphics;
212   import java.awt.Graphics2D;
213   import java.awt.geom.PathIterator;
214   import java.awt.geom.Point2D;
215   import java.awt.geom.RectangularShape;
216   import java.util.Vector;
217
218   import framework.interfaces.Deletable;
219   import framework.interfaces.Editable;
220   import framework.interfaces.Moveable;
221   import framework.interfaces.Resizable;
222   import framework.interfaces.Selectable;
223   import framework.interfaces.Snapper;
224   import framework.shape.AbstractShape;
225   import framework.state.DiagramController;
226
227   /**
228    * This class extends <code>AbstractShape</code> adding to it diagram
229    * behavior: it can be selected, moved, edited, etc.
230    *
231    * @author Reinaldo Espejo Matte
232    *
233    */
234   public class AbstractDiagramShape extends AbstractShape implements Deletable,
235   Editable, Moveable, Selectable, Snapper, Resizable
236   {
237
238       private static final long serialVersionUID = 7019331481478538422L;
239       protected Vector<Point2D> snapPoints;
240       // TODO Replace with a nice runtime method :)
241       protected Vector<Point2D> absoluteSnapPoints;
242       protected double precision = 1d;
243
244       protected Point2D NORTH_SNAP = new Point2D.Double(); public static final int N
      = 10;
245       protected Point2D SOUTH_SNAP = new Point2D.Double(); public static final int S
      = 20;
246       protected Point2D WEST_SNAP = new Point2D.Double(); public static final int W
      = 30;
247       protected Point2D EAST_SNAP = new Point2D.Double(); public static final int E
      = 40;
248
249       /**
250        * Constructs a new AbstractDiagramShape
251        */
252       public AbstractDiagramShape() {
```

Por defecto, la figura es un rectángulo blanco

Fin Licencia

Clases "base" de Swing utilizadas

Clases e interfaces del Framework utilizadas

Snaps, son los puntos donde la punta de una línea se magnetiza a la figura. Aquí se asegura que al menos se cuente con 4 snaps, N-S-E-O, además del centro de la figura

```
253         this("", null, RECTANGLE, FILL);
254       }
255
256       /**
257        * Constructs a new AbstractDiagramShape with a given name and icon
258        * @param name the name
259        * @param icon the icon
260        */
261       public AbstractDiagramShape(String name, String iconFile) {
262         this(name, iconFile, RECTANGLE, FILL);
263       }
264
265       public AbstractDiagramShape(String name, String iconFile, int... params)
266       {
267         super(name, iconFile, params);
268         this.snapPoints = new Vector<Point2D>();
269         this.absoluteSnapPoints = new Vector<Point2D>();
270         //    generateSnapPoints();
271         DiagramController.registerDiagramComponent(this, true, true, true, false);
272       }
273
274       /**
275        * @param defaultShape the defaultShape to set
276        */
277       public void setDefaultShape(RectangularShape defaultShape)
278       {
279         super.setDefaultShape(defaultShape);
280         //    if(snapPoints.size()==0)
281         //    describePath(defaultShape);
282         generateSnapPoints();
283       }
284       /*
285        * (non-Javadoc)
286        *
287        * @see interfaces.Selectable#setSelected(boolean)
288        */
289       @Override
290       public void setSelected(boolean selected) {
291         // TODO Auto-generated method stub
292
293       }
294
295
296       public void paintComponent(Graphics g)
297       {
298         super.paintComponent(g);
299         Graphics2D g2 = (Graphics2D) g;
300         //                      Un-comment for DEBUG
301         //                      g2.setColor(Color.GREEN);
302         //                      for(Point2D p: snapPoints)
303         //                      {
304         //                          g2.fillRect((int)p.getX()-2,  (int)p.getY()-2,  5,
      5);
305         //                      }
306         //                      g2.setColor(Color.MAGENTA);
307         //                      for(Point2D p: absoluteSnapPoints)
```

El código debug, muestra gráficamente los puntos de snap de la figura. Son dos, uno relativo a la figura, y otro relativo a la pantalla

```
308         //                      {
309         //                          g2.fillRect((int)p.getX()-2,  (int)p.getY()-2,  5,
    5);
310         //                      }
311       }
312
313       /**
314        * This method generates the snap points to be used for snapping lines.<br>
315        * <strong>Warning:</strong> This method is very cpu demanding: handle with
    care.
316        * @see framework.interfaces.Snapper#generateSnapPoints(double)
317        */
318       @Override
319       public void generateSnapPoints()
320       {
321         if (snapPoints != null)
322         {
323             snapPoints.removeAllElements();
324             if (this.getDefaultShape() != null)
325             {
326               PathIterator path = this.getDefaultShape().getPathIterator(
327
        null,this.getWidth()>this.getHeight()?this.getWidth()/2.0:this.getHeight()/2
    .0);
328               float[] pointPos = new float[6];
329               Point2D.Double newSnapPoint;
330               //first one is useless, it repeats on last SEG_LINETO
331               Point2D.Double firstSnapPoint = new Point2D.Double();
332               if (path.currentSegment(pointPos) == PathIterator.SEG_MOVETO )
333               {
334                   firstSnapPoint = new Point2D.Double(pointPos[0],
335                         pointPos[1]);
336                   snapPoints.add(firstSnapPoint);
337               }
338               path.next();
339               while (!path.isDone()) {
340                   if (path.currentSegment(pointPos) == PathIterator.SEG_LINETO)
341                   {
342                     // it's a vertical line
343                     if (!snapPoints.isEmpty())
344                     {
345                         if (snapPoints.lastElement().getX() == pointPos[0])
346                         {
347                           double a, b, c, d;
348                           if (snapPoints.lastElement().getY() > pointPos[1])
349                           {
350                               a = snapPoints.lastElement().getY();
351                               b = pointPos[1];
352                           } else {
353                               a = pointPos[1];
354                               b = snapPoints.lastElement().getY();
355                           }
356                           c = (a - b) / 2;
357                           d = a - c;
358
359                           if (Math.abs(a - b) > 10) // First iteration
```

Código que calcula el entorno de la figura, para determinar sus puntos de snap. Es relativamente complejo.

```
360                             {
361                                 snapPoints
362                                 .add(new Point2D.Double(snapPoints
363                                       .lastElement().getX(), d));
364                             }
365                             if (Math.abs(c) > 10) // Second iteration, optional
366                             {
367                                 snapPoints.add(new Point2D.Double(
368                                       snapPoints.lastElement().getX(),
369                                       (d - c / 2)));
370                                 snapPoints.add(new Point2D.Double(
371                                       snapPoints.lastElement().getX(),
372                                       (d + c / 2)));
373                             }
374                           }
375
376                           // it's an horizontal line
377                           if (snapPoints.lastElement().getY() == pointPos[1])
378                           {
379                             System.out.println("una vertical!");
380                             double a, b, c, d;
381                             if (snapPoints.lastElement().getX() > pointPos[0]) {
382                                 a = snapPoints.lastElement().getX();
383                                 b = pointPos[0];
384                             } else if (pointPos[0] > snapPoints
385                                   .lastElement().getX()) {
386                                 a = pointPos[0];
387                                 b = snapPoints.lastElement().getX();
388                             } else
389                                 break;
390                             c = (a - b) / 2;
391                             d = a - c;
392
393                             if (Math.abs(a - b) > 10) // First iteration
394                             {
395                                 snapPoints.add(new Point2D.Double(d,
396                                       snapPoints.lastElement().getY()));
397                             }
398                             if (Math.abs(c) > 10) // Second iteration, optional
399                             {
400                                 snapPoints.add(new Point2D.Double(
401                                       (d - c / 2), snapPoints
402                                       .lastElement().getY()));
403                                 snapPoints.add(new Point2D.Double(
404                                       (d + c / 2), snapPoints
405                                       .lastElement().getY()));
406                             }
407                           }
408                       }
409                       newSnapPoint = new Point2D.Double(pointPos[0], pointPos[1]);
410                       if(!snapPoints.contains(newSnapPoint))
411                           snapPoints.add(newSnapPoint);
412
413                   }
414                   path.next();
415               }
```

```
416             }
417             generateAbsoluteSnapPoints();
418         }
419
420       }
421
422       /**
423        * For each snap point, generates a point that's located relative to this
      shape's container
424        */
425       private void generateAbsoluteSnapPoints()
426       {
427         NORTH_SNAP.setLocation(this.getX()+this.getWidth()/2,this.getY()+0);
428         SOUTH_SNAP.setLocation(this.getX()+this.getWidth()/2,this.getY()+this.getHei
      ght());
429         EAST_SNAP.setLocation(this.getX()+this.getWidth(),this.getY()+this.getHeight
      ()/2);
430         WEST_SNAP.setLocation(this.getX()+0,this.getY()+this.getHeight()/2);
431         if(absoluteSnapPoints.isEmpty())
432             for (int i = 0; i < snapPoints.size(); i++) {
433               absoluteSnapPoints.add(i,
434                     new Point2D.Double(snapPoints.elementAt(i).getX()
435                           + this.getX(), snapPoints.elementAt(i).getY()
436                           + this.getY()));
437             }
438         else
439             for (int i = 0; i < snapPoints.size(); i++) {
440
          absoluteSnapPoints.elementAt(i).setLocation(snapPoints.elementAt(i).getX()
441                     + this.getX(), snapPoints.elementAt(i).getY()
442                     + this.getY());
443             }
444
445       }
446
447       /*
448        * (non-Javadoc)
449        *
450        * @see interfaces.Snappable#getNearestSnapPoint(java.awt.Point)
451        */
452       @Override
453       public Point2D getNearestSnapPoint(Point2D p, int close_enough) {
454         assert p!=null:"P NULL!!!!";
455         //avoids infinite loop
456         if(close_enough>this.getWidth()) return null;
457
458         for (Point2D asp : absoluteSnapPoints) {
459             if (Math.abs(asp.getX() - p.getX()) <= close_enough
460                   && Math.abs(asp.getY()- p.getY()) <= close_enough) {
461
462               return asp;
463             }
464         }
465         return getNearestSnapPoint(p, close_enough + 2);
466       }
467
```

Se calculan las posiciones de los snaps relativas a la ventana

Este método determina qué snap está más cercano a un punto dado (pensar en el mouse arrastrando una línea cerca de la figura)

```
468       public Point2D getNearestAbsoluteSnapPoint(Point2D p, int close_enough)
469       {
470         //avoids infinite loop
471         if(close_enough>this.getWidth()) return null;
472
473         for(Point2D sp: snapPoints)
474         {
475             if(Math.abs((sp.getX()+this.getX())-
      p.getX())<=close_enough&&Math.abs((this.getY()+sp.getY())-p.getY())<=close_enough)
476             {
477               sp.setLocation(sp.getX() + this.getX(), sp.getY() + this.getY());
478               return sp;
479             }
480         }
481         return getNearestAbsoluteSnapPoint(p, close_enough + 2);
482       }
483
484       /**
485        * This method was intended as a replacement for absolute snap points. It's a
      TODO for future versions
486        *
487        */
488       public Point2D getAbsolutePositionForPoint(Point2D sp) {
489         Point2D app = new Point2D.Double(sp.getX(), sp.getY());
490         app.setLocation(app.getX() + this.getX(), app.getY() + this.getY());
491         return app;
492       }
493
494       public void updateSizes() {
495         super.updateSizes(this.getSize());
496         generateSnapPoints();
497       }
498
499       /*
500        * (non-Javadoc)
501        *
502        * @see interfaces.Moveable#locationChanged()
503        */
504       @Override
505       public void locationChanged() {
506         generateAbsoluteSnapPoints();
507       }
508
509       /**
510        * @param p2
511        * @return
512        */
513       public int getSnapPointIndex(Point2D p2) {
514         return absoluteSnapPoints.indexOf(getNearestSnapPoint(p2, 2));
515       }
516
517       /**
518        * @param index
519        * @return
520        */
521       public Point2D getAbsoluteSnapPointAt(int index) {
```

```
522         if (index >= 0)
523             return absoluteSnapPoints.elementAt(index);
524         return getCenter();
525       }
526
527       public Point2D getLinePointConnector()
528       {
529         return this.getAbsoluteSnapPointAt(0);
530       }
531
532       /**
533        * @param snap1
534        * @return
535        */
536       public int[] getCoordinatesFor(int index) {
537         int[] coord = { 0, 0, 0, 0 };
538         int delta = 1;
539         if (index >= 0) {
540             Point2D p = snapPoints.elementAt(index);
541             if (Math.abs(this.getY() - p.getY()) < delta)
542               coord[0] = 1; // NORTH
543             if (Math.abs(this.getWidth() - p.getX()) < delta)
544               coord[1] = 1; // EAST
545             if (Math.abs(this.getHeight() - p.getY()) < delta)
546               coord[2] = 1; // SOUTH
547             if (Math.abs(this.getX() - p.getX()) < delta)
548               coord[3] = 1; // WEST
549         }
550         return coord;
551       }
552
553
554       /* (non-Javadoc)
555        * @see framework.interfaces.Snapper#getSnapPoint(int)
556        */
557       @Override
558       public Point2D getSnapPoint(int i)
559       {
560         switch (i)
561         {
562         case N:
563             return NORTH_SNAP;
564         case S:
565             return SOUTH_SNAP;
566         case E:
567             return EAST_SNAP;
568         case W:
569             return WEST_SNAP;
570         }
571         return null;
572       }
573
574       /**
575        * @param p1
576        * @return
577        */
```

```
578       public Point2D getAbsoluteSnapPointAt(Point2D p)
579       {
580         if(snapPoints.contains(p))
581             return absoluteSnapPoints.elementAt(snapPoints.indexOf(p));
582         return null;
583       }
584
585       /* (non-Javadoc)
586        * @see framework.interfaces.Resizable#notifySelection()
587        */
588       @Override
589       public void notifySelection() {
590       }
591
592   }
```

Anexo B.

Aplicaciones del Framework

En este anexo se muestran capturas de pantallas de aplicaciones que utilizan el Framework.

Módulo Scheduling, CAPPS

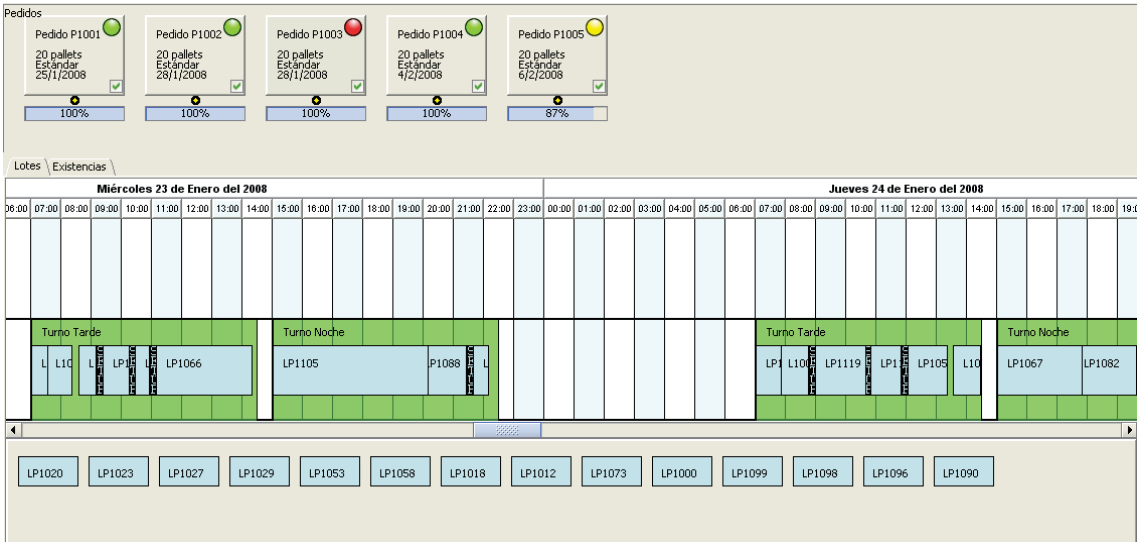

Figura Error! No text of specified style in document.**.1 :** ***Módulo Scheduling en ejecución. Prácticamente todos los elementos que se ven son extensiones de AbstractComponent. La aplicación contiene*** **3 secciones principales: Pedidos (arriba), Líneas (centro) y Lotes sin secuenciar (abajo).**

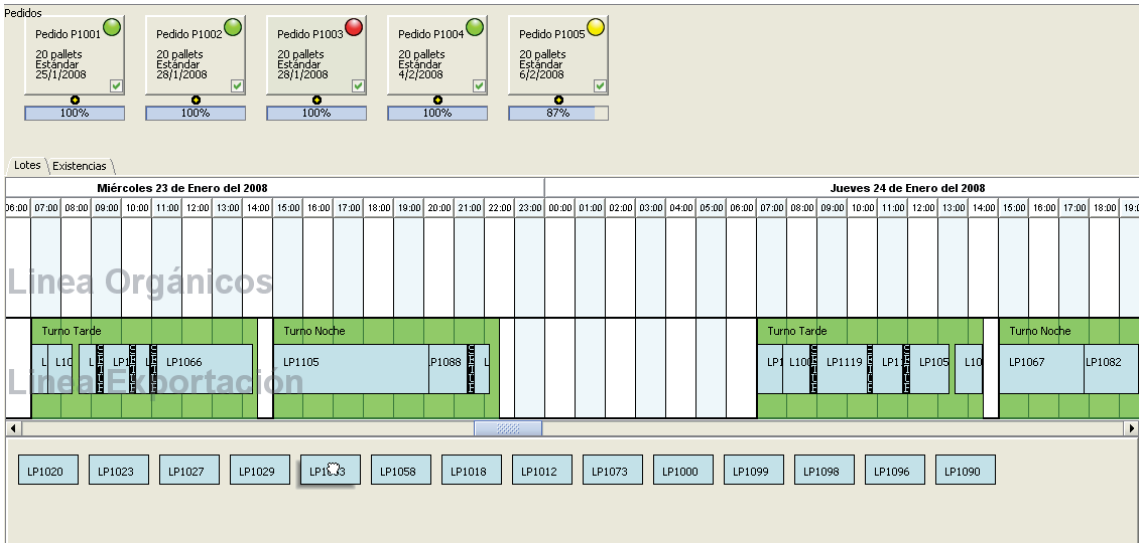

Figura Error! No text of specified style in document.**..2 :** ***Módulo Scheduling en ejecución. Una acción posible típica es tomar el lote no secuenciado y realizar un drag and drop en la línea correspondiente. Al presionar el lote, se muestran los nombres de las líneas en gris para que el usuario sepa cuál es cual.***

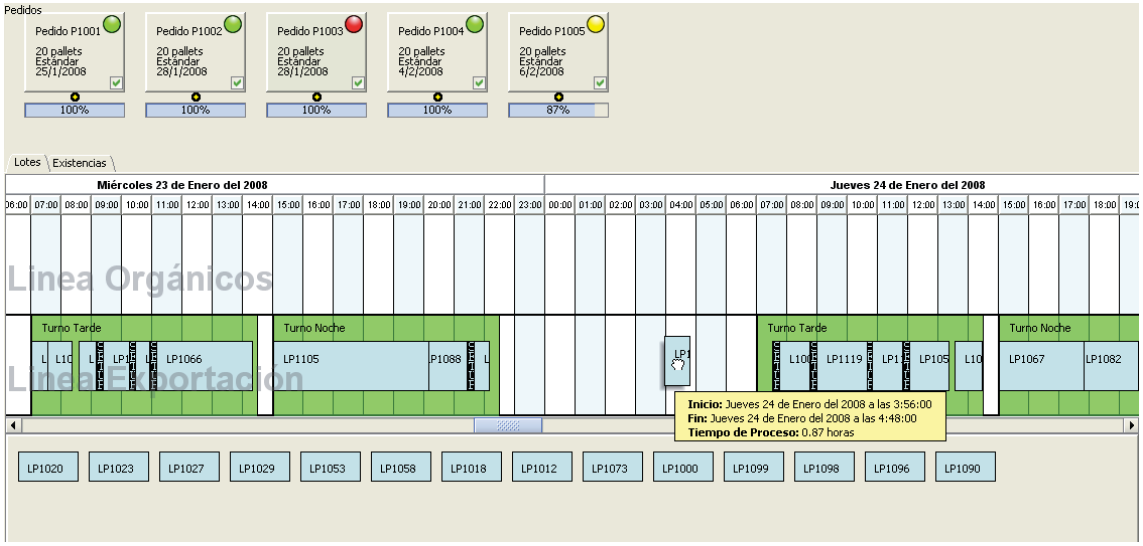

Figura Error! No text of specified style in document.**..3 :** ***Módulo Scheduling en ejecución. Dentro de la línea se realizan movimientos horizontales que significan cambiar la fecha de proceso del lote que se está arrastrando. El Framework permite usar tooltips propios que en este caso van mostrando la nueva fecha a medida que el usuario arrastra la "cajita".***

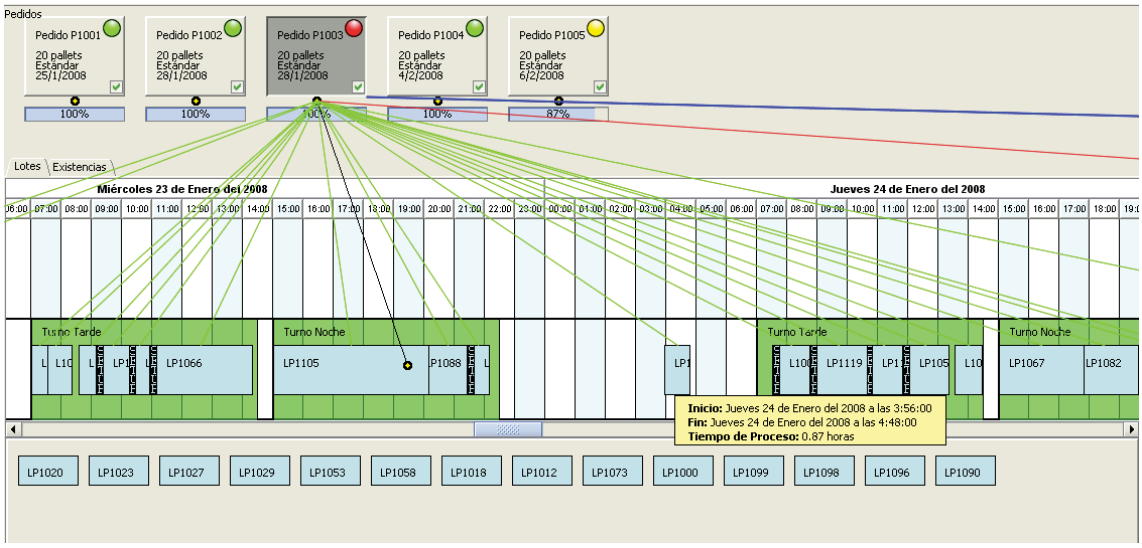

Figura Error! No text of specified style in document.**..4 :** ***Módulo Scheduling* en ejecución.** ***Al presionar en un pedido, se pueden ver gráficamente qué lotes están asignados a éste, y en rojo los que se procesan después de la fecha de entrega del pedido, y que por lo tanto lo atrasan y deben ser corregidos. Este tipo de dibujado "sobre todo" es posible gracias a la arquitectura por capas de la clase AbstractComponent del Framework. La línea negra con punta redonda es la que el usuario arrastra para asignar un lote a un pedido.***

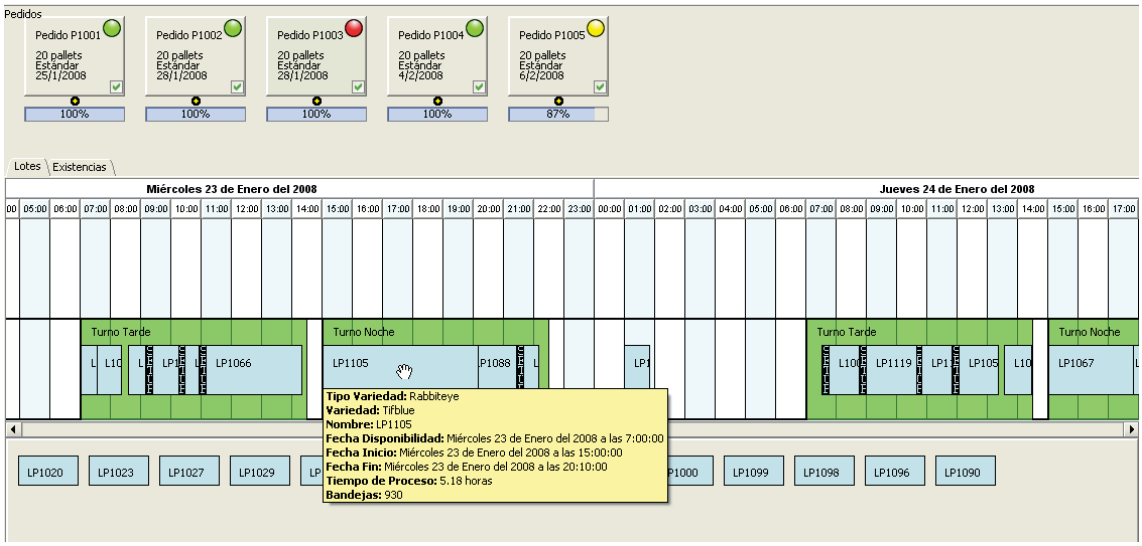

Figura Error! No text of specified style in document.**..5 :** ***Módulo Scheduling en ejecución. Todos los AbstractComponent tienen un tooltip que puede ser personalizado con colores, bordes, fuentes, y acepta también texto en formato html. En la captura se aprecia el tooltip típico de un lote, en este caso una "cajita".***

Prototipo software de diseño diagramas multi-agente

Este segundo caso es una aplicación tipo CASE para el mundo de los sistemas multi-agente y otros propósitos varios. Dada su naturaleza mucho más gráfica que el caso anterior, el aporte que hará al Framework será mucho mayor. Esta aplicación sólo llegó a etapas tempranas de desarrollo, pero de ella se obtuvieron algunas clases importantes del Framework, como *AbstractShape*, *DiagramCanvas*, y varias interfaces, como *Moveable*, *Resizable*, *Selectable*, etc.

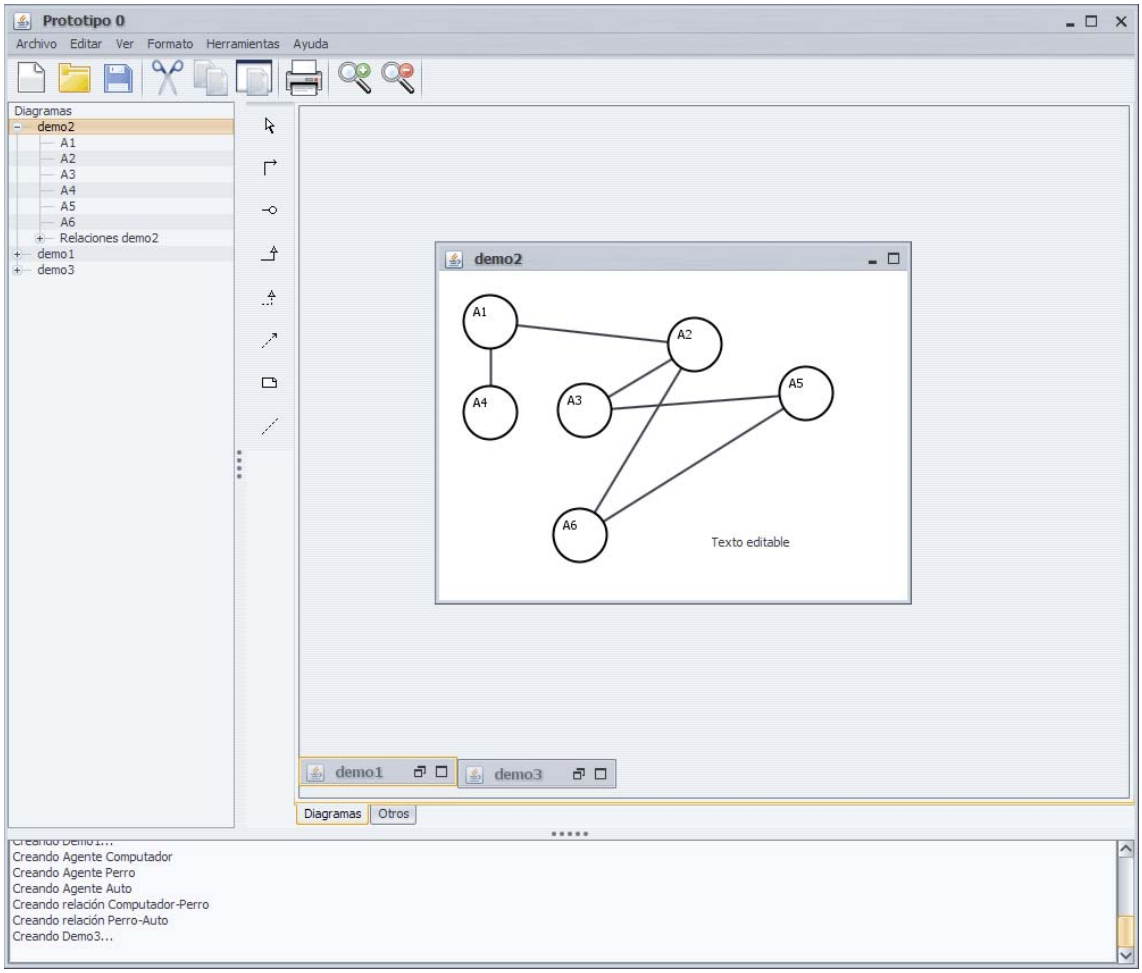

Figura Error! No text of specified style in document.**.6 :** ***Vista típica del prototipo. A la izquierda el árbol con los elementos del diagrama, a la derecha un contenedor de varios diagramas, cada uno en una ventana, y abajo un log de las acciones del usuario. Los agentes se representan por una clase GraphicAgent la cual extiende a AbstractShape, y todos están contenidos en un DiagramCanvas. Las relaciones se representan por líneas que unen dos agentes.***

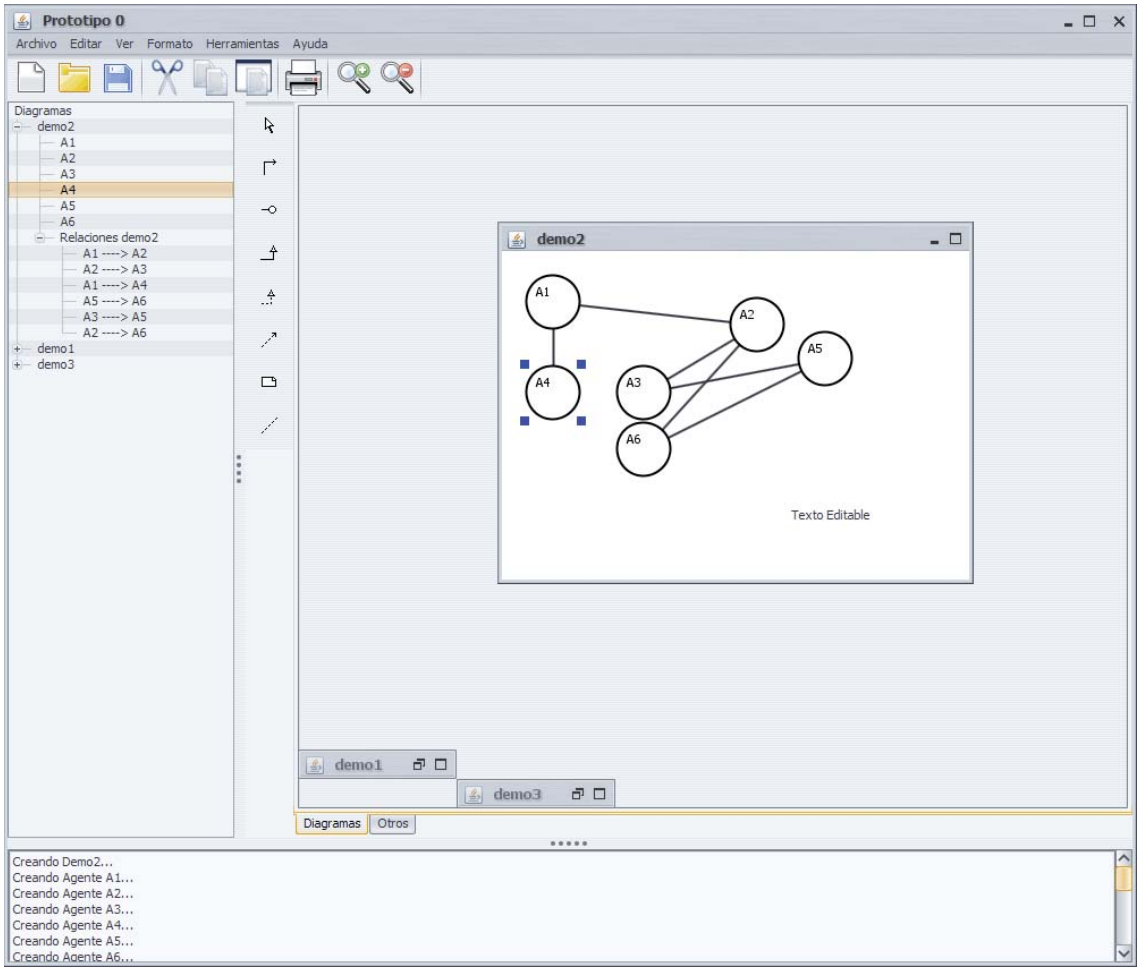

Figura Error! No text of specified style in document.**.7 :** ***Los comportamientos de cada AbstractShape se logran mediante el uso de interfaces. En este caso, la clase GraphicAgent es Selectable, Moveable y Resizable, por lo cual al hacer click sobre ella, se puede seleccionar, mover y agrandar/achicar.***

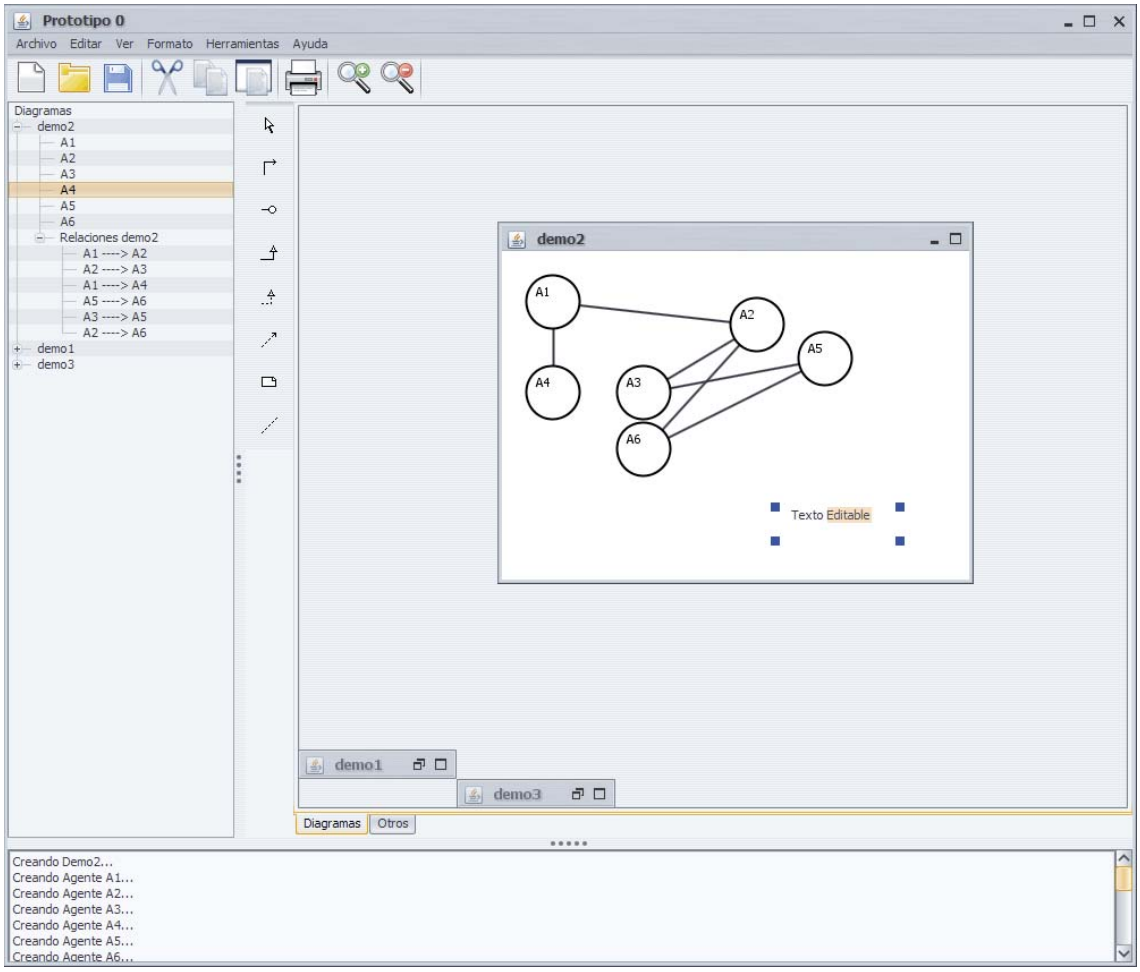

Figura Error! No text of specified style in document.**.8 :** ***El Framework provee elementos útiles que se esperan de cualquier herramienta gráfica, como lo es el texto editable sobre el gráfico. En este prototipo ya existe una versión de lo que sería este texto gráfico editable.***

UML-F Viewer

Como se vio en la sección 3.4, la ausencia de una herramienta de diseño de clases con perfil UML-F obligó al autor a tener que crear una propia. Denominada *UML-F Viewer*, la herramienta transforma código de clases a diagramas de clases con el perfil UML-F (ingeniería inversa).

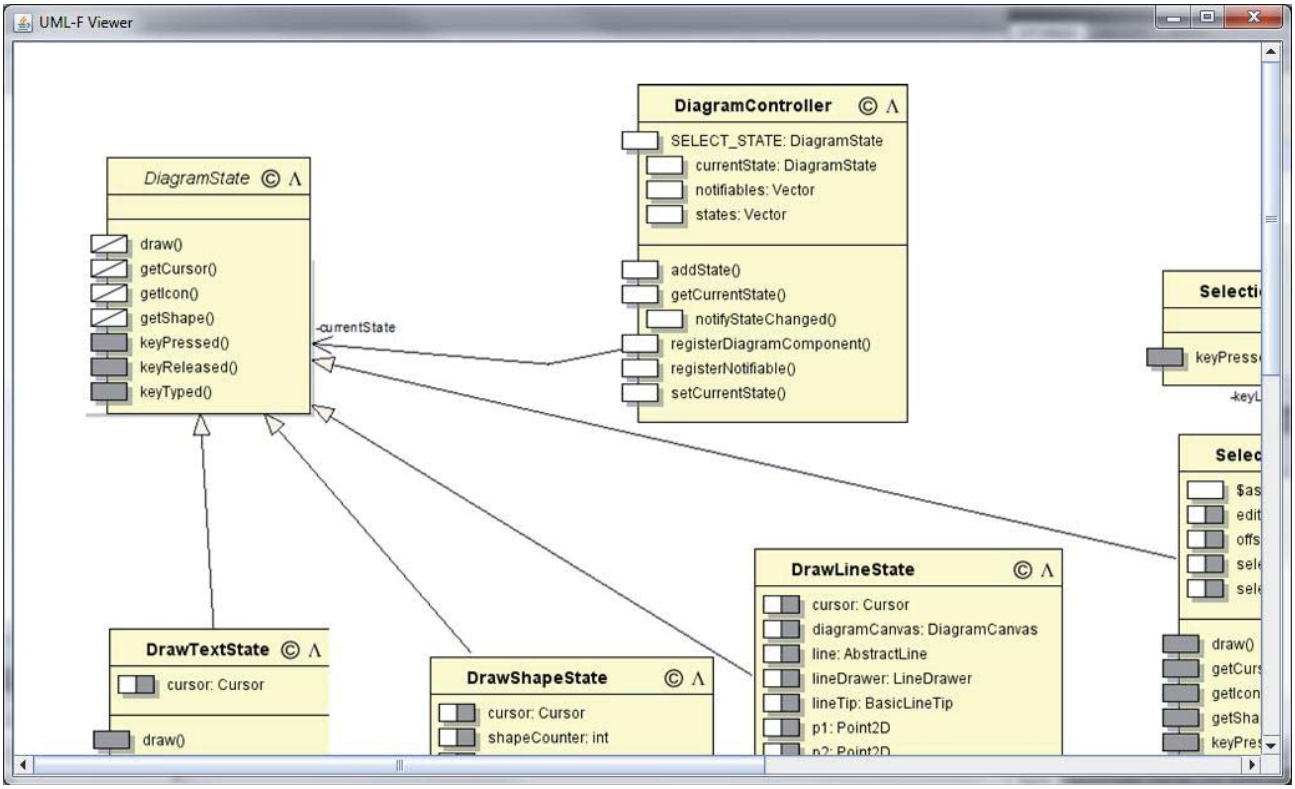

Figura Error! No text of specified style in document.**..9 :** ***UML-F Viewer mostrando clases del paquete state y sus relaciones, en formato UML-F***

Sistema de Estimación de Pérdidas 2.0 SAAM

Durante el 2010 se trabajó en un software para la empresa SAAM, específicamente su área de graneles, que estimaba y calculaba pérdidas y dimensiones de bodegas y bins para almacenaje de productos. Una de sus secciones contenía una representación gráfica de las dimensiones de una bodega y el producto almacenado, como se puede apreciar en la Figura B.40. Esta sección utiliza las clases *DiagramCanvas* y *AbstractShape* del Framework. Es un claro ejemplo de un modelo de datos completamente separado de su representación gráfica, y muestra una proporcionalidad metros-pixeles.

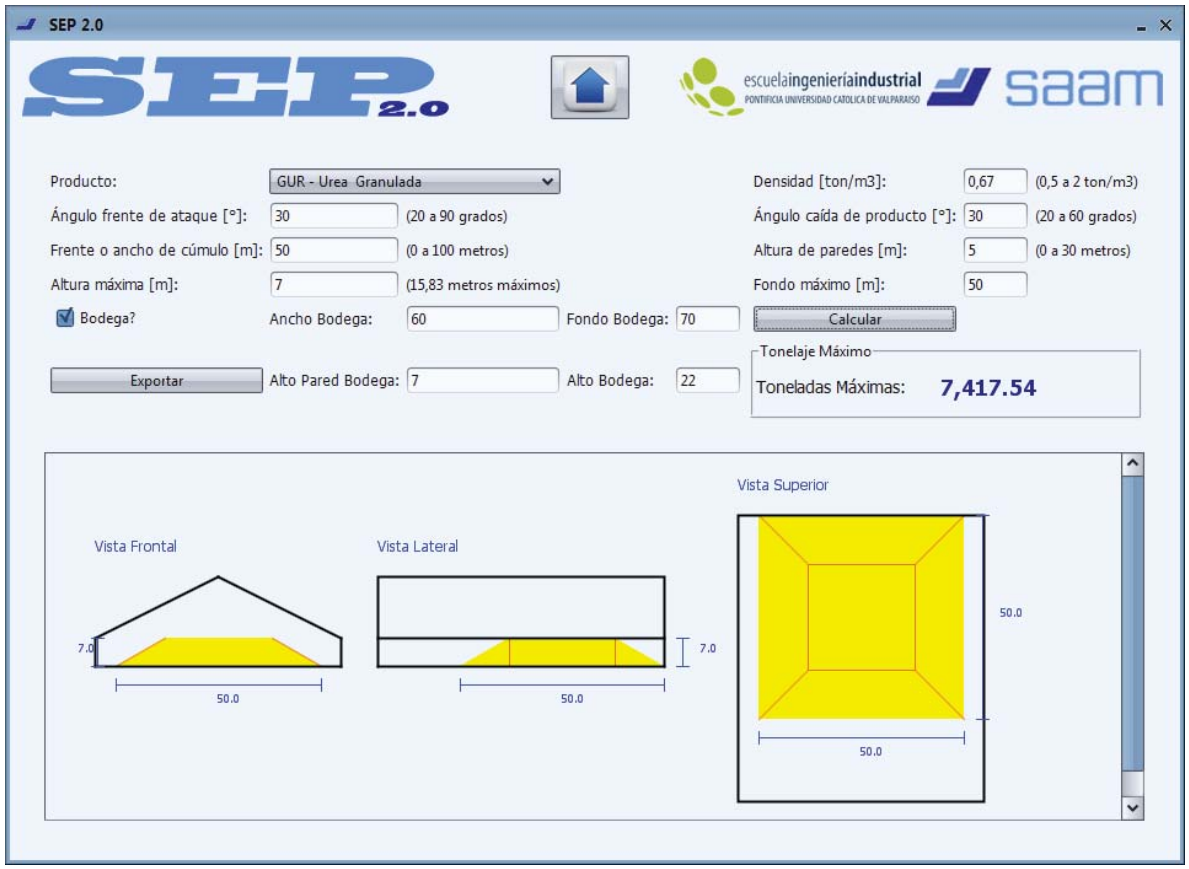

Figura Error! No text of specified style in document.**..10 :** ***SEP 2.0 mostrando una bodega y un cúmulo de producto con dimensiones dadas. El dibujo es proporcionado.***

Anexo C.

Diseño de las Pruebas

Diseño de Pruebas del Framework

Para probar el Framework como ayuda para facilitar el desarrollo de representaciones gráficas interactivas, se realizarán pruebas con usuarios. Los usuarios del Framework son, en general, desarrolladores en Java con algo de experiencia mínima. La prueba en sí consiste en desarrollar el código necesario para obtener un resultado en pantalla, el cual consistirá en un rectángulo y un círculo coloreados, unidos por una línea y que pueden ser movidos con el mouse.

Cada usuario tendrá una semana (5 días hábiles) para desarrollar la solución. Un grupo tendrá que lograrlo sólo con el uso de la API de Java y sus clases, y otro grupo tendrá que lograrlo utilizando el Framework. Previo a la entrega de la prueba, se le realiza a cada usuario un cuestionario que determinará más precisamente su perfil de desarrollador en Java, experiencia y conocimientos.

Una vez terminada la prueba y enviados los resultados, el usuario deberá completar otro cuestionario, en el cual explicará de cierto modo las facilidades o dificultades que tuvo a la hora de crear la solución. Además de esto se revisa el código de cada uno, para ver qué caminos se tomaron, si eran los esperados, y si se utilizó el Framework de manera adecuada.

Cuestionario Anterior

El cuestionario anterior a la prueba consiste en las siguientes preguntas con alternativas:

1.- ¿Hace cuanto tiempo programa en Java?

a. Menos de 1 año.

b. 1 año.

c. 2 años.

d. Más de 2 años.

Esta pregunta busca determinar el nivel de experiencia en general que tiene el usuario en Java, basándose en el tiempo total en que ha estado trabajando con este lenguaje.

2.- ¿Cuál de las siguientes clases considera más adecuada para construir una ventana en Java?

a. Window

b. JWindow

c. Frame

d. JFrame

Esta pregunta es clave para determinar los conocimientos del usuario en cuanto a gráfica en Java. Es básico y fundamental saber que JFrame (alternativa d) es la clase de Swing utilizada hoy en día para la creación de ventanas. Si marca Window o Frame, significa que sus conocimientos están limitados a AWT, lo cual implica que o aprendió desde material didáctico antiguo (por ejemplo de Java 1.2) o aprendió mal. JWindow es una alternativa correcta pero no es utilizada dad sus limitaciones (no tiene barra de título por ejemplo).

3.- ¿Qué clase se utiliza en Java para dibujar figuras geométricas?

a. Canvas

b. Component

c. JCanvas

d. JComponent

Si escoge la alternativa Canvas (a), significa que tiene conocimientos en Java2D, pero no actualizados, ya que esta clase solía utilizarse en un contexto de AWT. La opción correcta y actual es JComponent. JCanvas no existe, y Component puede utilizarse, pero jamás ha sido recomendada para este fin.

4.- ¿Qué método se utiliza en Java para dibujar figuras geométricas?

a. public void paint(Graphics g)

b. public void paint()

c. public void paintComponent(Graphics g)

d. public void paintComponents()

La alternativa correcta es la letra c. Si elige esta alternativa, significa que ya ha realizado métodos de pintado. Si escoge la letra a, significa que ha pintado, pero en un contexto de AWT. Las otras dos alternativas las marcará si no sabe o está tratando de adivinar.

5.-¿Qué clase está encargada de dibujar elementos gráficos en Java?

a. Graphics

b. Graphics2D

c. Drawer

d. Drawer2D

Las alternativas a y b son correctas, sin embargo, la letra b es más adecuada al contexto Java2D. Las otras dos alternativas no existen, y de ser elegidas, indicarían una clara falta de conocimientos por parte del usuario.

6.-¿Según UD, qué clase es la mejor para representar un rectángulo en Java? (todas estas clases existen)

a. Rectangle

b. Rectangle2D

c. Shape

d. RectangularShape

Todas las alternativas cumplen con el fin de poder representar un rectángulo (es decir, almacenan x,y,w,h). Sin embargo, la más versátil de ellas es Rectangle2D, que si es escogida, muestra conocimientos más actualizados por parte del usuario.

7.- Si tuviera que actualizar los contenidos de una pantalla, ¿a cuál método llamaría?

a. public void refreshContents()

b. public void repaintContents()

c. public void repaintAllContents(Window w)

d. public void repaint()

Las tres primeras alternativas son inventadas, no existen. Sólo la opción d es la correcta. Si escoge esta alternativa, hay una gran probabilidad de que sepa lo que está contestando, ya que las 3 primeras son mucho más llamativas para quien esté tratando de adivinar.

La Prueba

La prueba es la misma, para los dos casos. Se presenta sólo en forma de texto, sin ayudas visuales o resultados visuales esperados, para así analizar lo que el usuario entiende como resultado cumplido de la prueba. El texto de la prueba es como sigue:

“Utilizando (sólo Java AWT y Swing/ Framework), se requiere que construya una ventana cuyos contenidos sean un rectángulo rojo con borde negro y un círculo azul sin borde, unidos por una línea punteada. Además, deben poder ser movidos con el mouse por parte

del usuario. El movimiento debe ser independiente para cada uno y la conexión con la línea debe mantenerse.”

La prueba es simple, sin embargo requerirá el desarrollo de lo siguiente:

- La ventana contenedora de los elementos gráficos
- Los elementos gráficos por separado: el rectángulo y el círculo
- Pintar colores y bordes
- Crear una línea punteada
- Crear la unión de las figuras mediante la línea, es decir, relativizar las coordenadas de ésta
- Crear interacción mediante mouse
- Detectar el click del mouse (sobre la figura correspondiente)
- Responder al movimiento del mouse, alterando las coordenadas de la figura movida
- Actualizar el contenido de la pantalla, de forma de mostrar el movimiento de manera suave y sin “fantasmas”

El código final no debería exceder las 500 líneas, por lo cual es revisable línea a línea para cada código. Se pedirá además que el usuario intente ser ordenado, seguir una cierta lógica en la nomenclatura, y que comente su código. Para el caso de los que trabajen sólo con Java, se autoriza la utilización de tutoriales y ejemplos, mas no se autoriza el uso de bibliotecas externas que no pertenezcan a la API de Java, AWT y Swing.

Cuestionario Posterior

El cuestionario posterior repite algunas preguntas del cuestionario anterior, para ver el cambio, si es que existe, en los conocimientos del usuario para desarrollar elementos gráficos. El cuestionario posterior a la prueba consiste en las siguientes preguntas con alternativas:

1.- ¿Cuál de las siguientes clases considera más adecuada para construir una ventana en Java?

a. Window

b. JWindow

c. Frame

d. JFrame

2.- ¿Qué clase se utiliza en Java para dibujar figuras geométricas?

a. Canvas

b. Component

c. JCanvas

d. JComponent

3.- ¿Qué método se utiliza en Java para dibujar figuras geométricas?

a. public void paint(Graphics g)

b. public void paint()

c. public void paintComponent(Graphics g)

d. public void paintComponents()

5.-¿Qué clase está encargada de dibujar elementos gráficos en Java?

a. Graphics

b. Graphics2D

c. Drawer

d. Drawer2D

6.-¿Según UD, qué clase es la mejor para representar un rectángulo en Java? (todas estas clases existen)

a. Rectangle

b. Rectangle2D

c. Shape

d. RectangularShape

7.- Si tuviera que actualizar los contenidos de una pantalla, ¿a cuál método llamaría?

a. public void refreshContents()

b. public void repaintContents()

c. public void repaintAllContents(Window w)

d. public void repaint()

Hasta aquí, el usuario debería contestar todas las alternativas correctamente, como consecuencia de haber tenido la experiencia práctica dada por el desarrollo de la prueba. A continuación, van las preguntas que intentan rescatar las experiencias de los usuarios. Primero las preguntas comunes, y luego las preguntas para los que no utilizaron el Framework y los que sí.

Preguntas Comunes:

8.- ¿Cómo logró capturar la interacción con el mouse en Java?

a. Implementando la interfaz MouseListener

b. Extendiendo la clase MouseAdapter

c. Creando el método MouseClicked(MouseEvent e)

d. No hice nada para lograrlo

Las primeras 3 alternativas mostrarían que hubo trabajo por parte del usuario para capturar manualmente la interacción con mouse, que se espera que marquen quienes no utilizaron el Framework. Si alguien que utilizó el Framework no marca la alternativa d, significa que el Framework falló en manejar todas las interacciones automáticamente, o el usuario no comprendió que así lo hacía.

9.- ¿Cómo pintó el rectángulo de color rojo?

a. Utilizando Color.RED antes de pintar el rectángulo

b. Utilizando new Color(255,0,0) antes de pintar el rectángulo

c. Utilizando Color.red antes de pintar el rectángulo

d. Utilizando backgroundColor = Color.RED en las propiedades del rectángulo

Esta pregunta persigue los mismos objetivos que la pregunta anterior.

10.- ¿Qué tan complejo fue para UD el lograr los resultados de la prueba?

a. Muy complejo

b. Complejo

c. Poco Complejo

d. No logré los resultados

11.- Si UD no logró los resultados, ¿cuáles fueron estos resultados que NO logró? (marque todos los que correspondan)

a. Dibujar las figuras en pantalla

b. Dibujar la línea punteada

c. Lograr que la línea siguiera el movimiento

d. Lograr el movimiento de las figuras con el mouse

12.- ¿Considera que su código es fácilmente extensible (por ejemplo, agregar otro rectángulo o línea)?

a. Sí

b. No

Preguntas a usuarios Java:

13.- ¿Utilizó clases de Swing (javax.swing.*) , clases de AWT (java.awt.*) o una combinación de ambas?

a. Sólo Swing

b. Sólo AWT

c. Una combinación

d. No estoy seguro

14.- Elija la técnica que más se asemeja a la que utilizó para detectar el click de mouse:

a. Con el punto donde se realizó el click, revisé a cuál de las dos figuras estaba tocando

b. Hice que la figura misma detectara si estaba siendo tocada por el puntero del mouse

15.- Elija la técnica que más se asemeja a la que utilizó para pintar el rectángulo y su borde

a. Primero hice un fill() del color de fondo y luego un draw() del borde

b. Hice un componente, con background y border

c. Utilicé otra técnica : _____

16.-¿Utilizó un Pattern para lograr la línea punteada?

a. Sí

b. No

Preguntas a usuarios Framework:

13.- ¿Qué clase utilizó para el rectángulo y círculo?

a. AbstractShape

b. DiagramShape

c. AbstractComponent

d. Otra: _____

Con esta pregunta se puede verificar si utilizó la clase del Framework más adecuada para lograr los resultados de la prueba, que es DiagramShape.

14.- ¿Qué clase utilizó para la línea punteada?

a. BasicLine

b. DashedLine

c. Line2D

d. Otra:_____

La respuesta esperada es DashedLine, clase del Framework que entrega de forma inmediata una línea punteada. Line2D es de Java2D.

15.- ¿Cómo logró unir las dos figuras mediante la línea?

a. Utilicé los parámetros shape1 y shape2 de BasicLine o DashedLine

b. Actualicé los puntos x1,y1,x2,y2 de la línea cada vez que movía alguna de las figuras

c. Otro:_____

El Cuestionario termina con una sección de opiniones y comentarios sobre la prueba y el desarrollo de ésta. Los dos cuestionarios se realizan vía WEB, utilizando un motor de encuestas online (LimeSurvey), lo cual asegura que el anterior se realice antes de la prueba y el posterior una vez terminada y entregada la prueba, así como también asegurar que es respondida por quiénes corresponda una sola vez. La prueba y sus cuestionarios estarán disponibles en el sitio del Framework, para quienes se registren.

Anexo D.

Resultados Detallados Pruebas

Cuestionario Pre-Test Swing

Resultados

Número de registros en esta consulta:	4
Total de registros en esta encuesta:	4
Porcentaje del total:	100.00%

Resumen de campo para 001

¿Hace cuánto tiempo programa en Java?

Opción	Cuenta	Porcentaje
Menos de 1 año. (a)	2	50.00%
1 año. (b)	0	0.00%
2 años. (c)	1	25.00%
Más de 2 años. (d)	1	25.00%
Otro	0	0.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 002

¿Cuál de las siguientes clases considera más adecuada para construir una ventana en Java?

Opción	Cuenta	Porcentaje
Window (a)	0	0.00%
JWindow (b)	0	0.00%

Resumen de campo para 002

¿Cuál de las siguientes clases considera más adecuada para construir una ventana en Java?

Opción	Cuenta	Porcentaje
Frame (c)	0	0.00%
JFrame (d)	4	100.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 003

¿Qué clase se utiliza en Java para dibujar sobre ella?

Opción	Cuenta	Porcentaje
Canvas (a)	0	0.00%
Component (b)	0	0.00%
JCanvas (c)	3	75.00%
JComponent (d)	1	25.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 004

¿Qué método se utiliza en Java para dibujar figuras geométricas?

Opción	Cuenta	Porcentaje
public void paint(Graphics g) (a)	3	75.00%
public void paint() (b)	0	0.00%
public void paintComponent(Graphics g) (c)	0	0.00%
public void paintComponents() (d)	1	25.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 005

¿Qué clase está encargada de dibujar elementos gráficos en Java?

Opción	Cuenta	Porcentaje
Graphics (a)	2	50.00%
Graphics2D (b)	2	50.00%

Resumen de campo para 005

¿Qué clase está encargada de dibujar elementos gráficos en Java?

Opción	Cuenta	Porcentaje
Drawer (c)	0	0.00%
Drawer2D (d)	0	0.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 006

¿Según UD, qué clase es la mejor para representar un rectángulo en Java? (todas estas clases existen)

Opción	Cuenta	Porcentaje
Rectangle (a)	0	0.00%
Rectangle2D (b)	2	50.00%
Shape (c)	1	25.00%
RectangularShape (d)	1	25.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 007

Si tuviera que actualizar los contenidos de una pantalla, ¿a qué método llamaría?

Opción	Cuenta	Porcentaje
public void refreshContents() (a)	2	50.00%
public void repaintContents() (b)	0	0.00%
public void repaintAllContents(Window w) (c)	0	0.00%
public void repaint() (d)	2	50.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Cuestionario Post-Test Swing

Resultados

Número de registros en esta consulta: 4

Resultados

Total de registros en esta encuesta: 4

Porcentaje del total: 100.00%

Resumen de campo para 001

¿Cuál de las siguientes clases considera más adecuada para construir una ventana en Java?

Opción	Cuenta	Porcentaje
Window (a)	0	0.00%
JWindow (b)	0	0.00%
Frame (c)	0	0.00%
JFrame (d)	4	100.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 002

¿Qué clase se utiliza en Java para dibujar figuras geométricas?

Opción	Cuenta	Porcentaje
Canvas (a)	1	25.00%
Component (b)	1	25.00%
JCanvas (c)	0	0.00%
JComponent (d)	2	50.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 003

¿Qué método se utiliza en Java para dibujar figuras geométricas?

Opción	Cuenta	Porcentaje
public void paint(Graphics g) (a)	3	75.00%
public void paint() (b)	0	0.00%
public void paintComponent(Graphics g) (c)	1	25.00%
public void paintComponents() (d)	0	0.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 004

¿Qué clase está encargada de dibujar elementos gráficos en Java?

Opción	Cuenta	Porcentaje
Graphics (a)	3	75.00%
Graphics2D (b)	1	25.00%
Drawer (c)	0	0.00%
Drawer2D (d)	0	0.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 005

¿Según UD, qué clase es la mejor para representar un rectángulo en Java? (todas estas clases existen)

Opción	Cuenta	Porcentaje
Rectangle (a)	1	25.00%
Rectangle2D (b)	2	50.00%
Shape (c)	1	25.00%
RectangularShape (d)	0	0.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 006

Si tuviera que actualizar los contenidos de una pantalla, ¿a cuál método llamaría?

Opción	Cuenta	Porcentaje
public void refreshContents() (a)	2	50.00%
public void repaintContents() (b)	0	0.00%
public void repaintAllContents(Window w) (c)	0	0.00%
public void repaint() (d)	2	50.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 007

¿Cómo logró capturar la interacción con el mouse en Java?

Opción	Cuenta	Porcentaje
Implementando la interfaz MouseListener (a)	0	0.00%
Extendiendo la clase MouseAdapter (b)	1	25.00%
Creando el método MouseClicked(MouseEvent e) (c)	0	0.00%
No hice nada para lograrlo (d)	3	75.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 008

¿Cómo pintó el rectángulo de color rojo?

Opción	Cuenta	Porcentaje
Utilizando Color.RED antes de pintar el rectángulo (a)	3	75.00%
Utilizando new Color(255,0,0) antes de pintar el rectángulo (b)	0	0.00%
Utilizando Color.red antes de pintar el rectángulo (c)	1	25.00%
Utilizando backgroundColor = Color.RED en las propiedades del rectángulo (d)	0	0.00%
No lo hice (e)	0	0.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 009

¿Qué tan complejo fue para UD el lograr los resultados de la prueba?

Opción	Cuenta	Porcentaje
Muy complejo (a)	0	0.00%
Complejo (b)	1	25.00%
Poco Complejo (c)	1	25.00%
No logré los resultados (d)	2	50.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 010

Si UD no logró los resultados, ¿cuáles fueron estos resultados que NO logró? (marque todos los que correspondan)

Opción	Cuenta	Porcentaje
Dibujar las figuras en pantalla (a)	0	0.00%
Dibujar la línea punteada (b)	1	25.00%
Lograr que la línea siguiera el movimiento (c)	3	75.00%
Lograr el movimiento de las figuras con el mouse (d)	3	75.00%

Resumen de campo para 011

¿Considera que su código es fácilmente extensible (por ejemplo, agregar otro rectángulo o línea)?

Opción	Cuenta	Porcentaje
Sí (Y)	0	0.00%
No (N)	4	100.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 012

¿Utilizó clases de Swing (javax.swing.*) , clases de AWT (java.awt.*) o una combinación de ambas?

Opción	Cuenta	Porcentaje
Sólo Swing (a)	0	0.00%
Sólo AWT (b)	0	0.00%
Una combinación (c)	2	50.00%
No estoy seguro (d)	2	50.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 013

Elija la técnica que más se asemeja a la que utilizó para detectar el click de mouse:

Opción	Cuenta	Porcentaje

Resumen de campo para 013

Elija la técnica que más se asemeja a la que utilizó para detectar el click de mouse:

Opción	Cuenta	Porcentaje
Con el punto donde se realizó el click, revisé a cuál de las dos figuras estaba tocando (a)	1	25.00%
Hice que la figura misma detectara si estaba siendo tocada por el puntero del mouse (b)	0	0.00%
Sin respuesta	3	75.00%
No completada	0	0.00%

Resumen de campo para 014

Elija la técnica que más se asemeja a la que utilizó para pintar el rectángulo y su borde

Opción	Cuenta	Porcentaje
Primero hice un fill() del color de fondo y luego un draw() del borde (a)	3	75.00%
Hice un componente, con background y border (b)	1	25.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 015

¿Utilizó un Pattern para lograr la línea punteada?

Opción	Cuenta	Porcentaje
Sí (Y)	2	50.00%
No (N)	2	50.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Cuestionario Pre-Test Framework

Resultados

Número de registros en esta consulta: 4

Total de registros en esta encuesta: 4

Porcentaje del total: 100.00%

Resumen de campo para 001

¿Hace cuánto tiempo programa en Java?

Opción	Cuenta	Porcentaje
Menos de 1 año. (a)	0	0.00%
1 año. (b)	0	0.00%
2 años. (c)	1	25.00%
Más de 2 años. (d)	3	75.00%
Otro	0	0.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 002

¿Cuál de las siguientes clases considera más adecuada para construir una ventana en Java?

Opción	Cuenta	Porcentaje
Window (a)	0	0.00%
JWindow (b)	0	0.00%
Frame (c)	2	50.00%
JFrame (d)	2	50.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 003

¿Qué clase se utiliza en Java para dibujar sobre ella?

Opción	Cuenta	Porcentaje
Canvas (a)	2	50.00%
Component (b)	0	0.00%
JCanvas (c)	2	50.00%
JComponent (d)	0	0.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 004

¿Qué método se utiliza en Java para dibujar figuras geométricas?

Opción	Cuenta	Porcentaje
public void paint(Graphics g) (a)	1	25.00%
public void paint() (b)	1	25.00%
public void paintComponent(Graphics g) (c)	2	50.00%
public void paintComponents() (d)	0	0.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 005

¿Qué clase está encargada de dibujar elementos gráficos en Java?

Opción	Cuenta	Porcentaje
Graphics (a)	1	25.00%
Graphics2D (b)	2	50.00%
Drawer (c)	1	25.00%
Drawer2D (d)	0	0.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 006

¿Según UD, qué clase es la mejor para representar un rectángulo en Java? (todas estas clases existen)

Opción	Cuenta	Porcentaje
Rectangle (a)	2	50.00%
Rectangle2D (b)	1	25.00%
Shape (c)	0	0.00%
RectangularShape (d)	1	25.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 007

Si tuviera que actualizar los contenidos de una pantalla, ¿a qué método llamaría?

Opción	Cuenta	Porcentaje

Resumen de campo para 007

Si tuviera que actualizar los contenidos de una pantalla, ¿a qué método llamaría?

Opción	Cuenta	Porcentaje
public void refreshContents() (a)	0	0.00%
public void repaintContents() (b)	0	0.00%
public void repaintAllContents(Window w) (c)	1	25.00%
public void repaint() (d)	3	75.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Cuestionario Post-Test Framework

Resultados

Número de registros en esta consulta: 4

Total de registros en esta encuesta: 4

Porcentaje del total: 100.00%

Resumen de campo para 001

¿Cuál de las siguientes clases considera más adecuada para construir una ventana en Java?

Opción	Cuenta	Porcentaje
Window (a)	0	0.00%
JWindow (b)	0	0.00%
Frame (c)	0	0.00%
JFrame (d)	4	100.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 002

¿Qué clase se utiliza en Java para dibujar figuras geométricas?

Opción	Cuenta	Porcentaje
Canvas (a)	1	25.00%
Component (b)	0	0.00%

Resumen de campo para 002

¿Qué clase se utiliza en Java para dibujar figuras geométricas?

Opción	Cuenta	Porcentaje
JCanvas (c)	2	50.00%
JComponent (d)	1	25.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 003

¿Qué método se utiliza en Java para dibujar figuras geométricas?

Opción	Cuenta	Porcentaje
public void paint(Graphics g) (a)	1	25.00%
public void paint() (b)	1	25.00%
public void paintComponent(Graphics g) (c)	2	50.00%
public void paintComponents() (d)	0	0.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 004

¿Qué clase está encargada de dibujar elementos gráficos en Java?

Opción	Cuenta	Porcentaje
Graphics (a)	0	0.00%
Graphics2D (b)	2	50.00%
Drawer (c)	0	0.00%
Drawer2D (d)	2	50.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 005

¿Según UD, qué clase es la mejor para representar un rectángulo en Java? (todas estas clases existen)

Opción	Cuenta	Porcentaje
Rectangle (a)	0	0.00%
Rectangle2D (b)	3	75.00%

Resumen de campo para 005

¿Según UD, qué clase es la mejor para representar un rectángulo en Java? (todas estas clases existen)

Opción	Cuenta	Porcentaje
Shape (c)	0	0.00%
RectangularShape (d)	1	25.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 006

Si tuviera que actualizar los contenidos de una pantalla, ¿a cuál método llamaría?

Opción	Cuenta	Porcentaje
public void refreshContents() (a)	0	0.00%
public void repaintContents() (b)	0	0.00%
public void repaintAllContents(Window w) (c)	0	0.00%
public void repaint() (d)	4	100.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 007

¿Cómo logró capturar la interacción con el mouse en Java?

Opción	Cuenta	Porcentaje
Implementando la interfaz MouseListener (a)	0	0.00%
Extendiendo la clase MouseAdapter (b)	1	25.00%
Creando el método MouseClicked(MouseEvent e) (c)	0	0.00%
No hice nada para lograrlo (d)	3	75.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 008

¿Cómo pintó el rectángulo de color rojo?

Opción	Cuenta	Porcentaje

Resumen de campo para 008

¿Cómo pintó el rectángulo de color rojo?

Opción	Cuenta	Porcentaje
Utilizando Color.RED antes de pintar el rectángulo (a)	1	25.00%
Utilizando new Color(255,0,0) antes de pintar el rectángulo (b)	0	0.00%
Utilizando Color.red antes de pintar el rectángulo (c)	0	0.00%
Utilizando backgroundColor = Color.RED en las propiedades del rectángulo (d)	2	50.00%
No lo hice (e)	1	25.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 009

¿Qué tan complejo fue para UD el lograr los resultados de la prueba?

Opción	Cuenta	Porcentaje
Muy complejo (a)	0	0.00%
Complejo (b)	1	25.00%
Poco Complejo (c)	2	50.00%
No logré los resultados (d)	1	25.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 010

Si UD no logró los resultados, ¿cuáles fueron estos resultados que NO logró? (marque todos los que correspondan)

Opción	Cuenta	Porcentaje
Dibujar las figuras en pantalla (a)	2	50.00%
Dibujar la línea punteada (b)	1	25.00%
Lograr que la línea siguiera el movimiento (c)	2	50.00%
Lograr el movimiento de las figuras con el mouse (d)	3	75.00%

Resumen de campo para 011

¿Considera que su código es fácilmente extensible (por ejemplo, agregar otro rectángulo o línea)?

Opción	Cuenta	Porcentaje
Sí (Y)	3	75.00%
No (N)	1	25.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 012

¿Qué clase utilizó para el rectángulo y círculo?

Opción	Cuenta	Porcentaje
AbstractShape (a)	2	50.00%
DiagramShape (b)	0	0.00%
AbstractComponent (c)	1	25.00%
Otro	1	25.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 013

¿Qué clase utilizó para la línea punteada?

Opción	Cuenta	Porcentaje
BasicLine (a)	0	0.00%
DashedLine (b)	3	75.00%
Line2D (c)	1	25.00%
Otro	0	0.00%
Sin respuesta	0	0.00%
No completada	0	0.00%

Resumen de campo para 014

¿Cómo logró unir las dos figuras mediante la línea?

Opción	Cuenta	Porcentaje

Resumen de campo para 014

¿Cómo logró unir las dos figuras mediante la línea?

Opción	Cuenta	Porcentaje
Utilicé los parámetros shape1 y shape2 de BasicLine o DashedLine (a)	1	25.00%
Actualicé los puntos x1,y1,x2,y2 de la línea cada vez que movía alguna de las figuras (b)	1	25.00%
Otro	2	50.00%
Sin respuesta	0	0.00%
No completada	0	0.00%