UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"VERIFICACIÓN DE LA APLICACIÓN DE ALGUNOS PATRONES EN PROYECTOS DE CÓDIGO ABIERTO MEDIANTE EL ANÁLISIS DE CÓDIGO FUENTE"

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRA EN INGENIERÍA
(COMPUTACIÓN)

P R E S E N T A:

ARLEN PÉREZ ARANA

DIRECTOR DE TESIS:

M. EN C. MARÍA GUADALUPE ELENA IBARGÜENGOITIA GONZÁLEZ

México D.F. 2012.

AGRADECIMIENTOS

Muchas gracias a Dios y a mis padres, por su apoyo incondicional y amor infinito, sin ustedes no sería lo que soy ahora.

Mil gracias al amor de mi vida Joaquín Amozurrutia por haberme apoyado en cada momento, por tus ideas y sabiduría, sin ti no lo hubiera logrado.

Gracias a mi tutora Guadalupe Ibargüengoitia por su esfuerzo, apoyo y conocimiento.

Agradezco también a todas aquellas personas que me han brindado su amistad verdadera, gracias Laura Hernández, Vanessa Vega, Aaron Altamirano los quiero muchísimo.

Índice

INTRODUCCIÓN

Durante el proceso de desarrollo de software, nos encontramos con la difícil tarea de tomar decisiones. Algunas de ellas son: ¿cómo vamos a diseñar el sistema?, ¿cuál será la arquitectura del sistema? Además nos encontramos también con el problema de cómo aplicar el conocimiento adquirido anteriormente, cómo poder aplicar las lecciones aprendidas y la experiencia adquirida a lo largo de los años a los nuevos proyectos a desarrollar para que, de esta forma, se pueda producir software con buenas características como la flexibilidad, extensibilidad y eficiencia.

Las respuestas a estas preguntas se pueden encontrar en los *patrones de diseño*. Los expertos comúnmente resuelven los nuevos problemas con los que se enfrentan aplicando soluciones exitosas con las que han trabajado en el pasado. Se identifican partes de los problemas que son parecidas con las partes que se han encontrado anteriormente.Posteriormente aplican esas soluciones para los problemas actuales y generalizan esta solución. Finalmente, adaptan la solución general al contexto de nuevos problemas.

En este proyecto se pretende proponer criterios para la adecuada implementación[1] de algunos patrones de diseño de manera que estos patrones de diseño se implementen en futuros proyectos de forma adecuada. Se tomó la decisión de analizar específicamente los patrones *Observer, Decorator* y *Adapter*[2] debido a su estructura, diseño y aplicabilidad. Se sabe además que estos patrones de diseño son comúnmente implementados en proyectos de desarrollo de software (como se menciona en el libro "*Design Pattern*", pág. XV, (Gamma, 1996)) por lo que se pretende contribuir con la mejora de las prácticas de ingeniería de software al proponer criterios para su implementación; se sospecha que en algunos proyectos no se aplican adecuadamente, en el artículo "*Is Design Dead?*" (Fowler, 2000) se mencionan algunos fracasos en el diseño de software, de entre ellos la mala implementación de patrones de diseño; en el libro "*AntiPatterns: Refactoring Software, Architectures and Projects in Crisis*" (Brown, Malveau, "Skips" McCormick, & Mowbray, 1998) se describe en general la forma no correcta de diseñar software.

"El Antipatrón probablemente sea el resultado de que los desarrolladores no sepan otra manera mejor de diseñar software, probablemente no tienen suficiente conocimiento o experiencia en la resolución de un problema en particular, o en la aplicación de patrones de diseño en un buen contexto" (Brown, Malveau, "Skips" McCormick, & Mowbray, 1998).

Este proyecto se enfocará en proyectos escritos en código java. El alcance del proyecto de estudio se enfocará en tres patrones: *Observer*, *Adapter* y *Decorator*.

Se seleccionaron estos patrones debido a sus características (estas se describirán a detalle en el Capítulo 1 "Introducción a los patrones de diseño").

Estos patrones son interesantes debido a que su funcionalidad y características permiten que su aplicación sea de gran utilidad en muchos proyectos, es decir, existe una gran probabilidad de que nos encontremos con proyectos en los que se necesite o se pueda implementar estos patrones de

[1] Para los fines del presente trabajo el término "implementación" se refiere a la estructura estática de cada una de las instancias de los patrones de diseño, esta estructura se puede analizar por medio de un diagrama de clases, por lo que el análisis dinámico queda fuera del alcance de este trabajo.

[2] En el presente trabajo se mencionarán los patrones de diseño en inglés por convención.

manera que el proceso de desarrollo de software sea mejor (sea eficiente) y en consecuencia tener un producto de mejor calidad en un periodo de tiempo más corto.

En el caso del patrón *Observer*, recordemos que es el que envía mensajes a los objetos que están interesados en el mensaje, esta característica se puede encontrar en muchos de los requerimientos de proyectos a desarrollar ya que la comunicación "uno a muchos" es de gran utilidad en muchas ocasiones.

En el caso del patrón *Adapter*, este patrón también se puede implementar en muchos proyectos y a su vez se puede encontrar como una solución en muchas situaciones en las que se requiera utilizar componentes de software ya hechos que no sean compatibles con los nuevos componentes a implementar, es por esta razón que este patrón ha sido de mi interés y del interés de muchos expertos en el área.

En cuanto al patrón *Decorator*, recordemos que este patrón se utiliza en los casos en los que requieran que un objeto (con objeto me refiero a un componente de software, en la mayoría de los casos una instancia) tenga la flexibilidad de tener pocas o muchas características (objetos que decoran el mismo objeto) dependiendo de las necesidades, debido a lo anterior se puede asegurar que la implementación de este patrón podrá ser de gran ayuda en el desarrollo de diversos proyectos.

Asimismo, se estudiará la aplicación de estos patrones en dos proyectos de software libre, escritos en Java: *JHotDraw* y *Swing*.

La razón principal por la que se eligieron estos proyectos es porque éstos son de código abierto por lo que así se puede analizar libremente el código, su estructura y documentación.

La segunda razón por la que se eligieron es porque la estructura del código es la adecuada para poder realizar el análisis, es decir, la organización y las decisiones que se tomaron para escribir el código y a su vez desarrollar el proyecto fueron las correctas ya que el código y su documentación es fácil de comprender, las clases son fácilmente ubicables y su estructura permite que el comportamiento y tarea de cada clase e instancia sean fáciles de analizar.

Objetivos.

- Establecer criterios estáticos generales para detectar instancias e implementar los patrones de diseño: *Observer, Adapter* y *Decorator* en proyectos de código abierto donde se apliquen.
- Comparar el análisis realizado con otros proyectos realizados por colegas de la *École de Technologie Supérieure* de la *Université du Québec* para validar la aplicación de los patrones de diseño en los proyectos de *JHotDraw* y *Swing*.
- Que otras personas apliquen los criterios estáticos establecidos anteriormente en futuros proyectos con el objetivo de tener una mejor implementación de dichos patrones.
- Analizar la implementación de los patrones de diseño de *Observer*, *Adapter* y *Decorator* en los proyectos de *JHotDraw* y *Swing* con el objetivo de verificar que estos hayan sido aplicados correctamente de acuerdo a los criterios que se definirán en la misma investigación ya que se sospecha que los patrones no son adecuadamente aplicados en

algunos proyectos de código abierto, de acuerdo al artículo "I*s Design Dead?*" (Fowler, 2000) y en el libro "*AntiPatterns: Refactoring Software, Architectures and Projects in Crisis*" (Brown, Malveau, "Skips" McCormick, & Mowbray, 1998).

Contribución y relevancia.

Existen muchos proyectos, en los que los patrones de diseño no son implementados adecuadamente, de acuerdo al artículo "I*s Design Dead?*" (Fowler, 2000) y en el libro "*AntiPatterns: Refactoring Software, Architectures and Projects in Crisis*" (Brown, Malveau, "Skips" McCormick, & Mowbray, 1998), ya que no se cumplen los criterios generales para aplicarlos o no están bien definidos ni establecidos de acuerdo a características estandarizadas de dichos patrones.

Con este proyecto se pretenden establecer criterios generales (en este proyecto de tesis únicamente se estudiarán tres patrones) y de acuerdo a éstos detectarlos y evaluar si son aplicados adecuadamente en proyectos de código abierto; con lo anterior se pretende contribuir con una guía para detectar estos patrones de diseño, así como contribuir a las buenas prácticas de ingeniería de software al implementar los patrones de diseño.

Metodología.

La metodología está basada en iteraciones. La razón por la cual se pensó en trabajar así es porque al analizar el primer patrón se obtiene experiencia que se usa en las siguientes iteraciones, a la vez que se va refinando el análisis. Además es porque no se puede predecir el resultado de las iteraciones antes de llevar a cabo la experimentación.

Primera iteración:

- Encontrar las instancias del patrón *Observer* en cada uno de los proyectos de código abierto (*JHotDraw* y *Swing*).
- Establecer criterios para una buena aplicación del patrón *Observer* a partir de algunas referencias confiables que contienen información valiosa acerca de este patrón.
- Validar cada una de las instancias del patrón *Observer* contra los criterios obtenidos anteriormente. Esta validación se realiza mediante el análisis de la estructura del código fuente.
- Usar algunas herramientas que ayudarán a realizar la validación de la estructura de los elementos en el patrón *Observer*. Estas herramientas son "*Enterprise Architect"* y técnicas empleadas en el análisis.
- Documentar los resultados obtenidos, determinar si el patrón es aplicado adecuadamente o no a ambos proyectos de acuerdo a los criterios establecidos y explicar las razones de por qué se llegó a esa conclusión.

Segunda iteración:

- De igual forma en que se procedió con el patrón *Observer* se procederá con el patrón *Adapter* y/o *Decorator* e inclusive se puede continuar con el patrón *Observer*. Esto se planeó así porque no se puede predecir que va a pasar con la primera iteración (el análisis del patrón *Observer*) no es muy útil decir cuál de los patrones serán estudiados en las siguientes iteraciones.

Tercera y cuarta iteración:

- Escribir los capítulos correspondientes en la tesis según lo obtenido en la iteración pasada.
- Se realizarán una o dos iteraciones más dependiendo del resultado de iteraciones anteriores y siguiendo los mismos pasos que en la primera iteración, los patrones estudiados en dichas iteraciones dependerán del criterio del punto anterior.

Quinta iteración:

- Integrar la documentación y los resultados obtenidos en las iteraciones anteriores, así como documentar los resultados generales.
- Acabar de escribir el resto de los capítulos para completar la tesis.

Estructura del trabajo.

El presente trabajo está compuesto por una primera parte en la que se introduce al lector en los conceptos básicos que se mencionan en el resto del trabajo, posteriormente se muestra el análisis realizado así como las conclusiones a las que se llegó de acuerdo al mismo.

En el capítulo 1 “Introducción a los patrones de diseño” se describe brevemente qué son los patrones de diseño, cómo se clasifican y algunos ejemplos de ellos. Se describen también los patrones de diseño analizados en el presente trabajo; específicamente se define la aplicabilidad, descripción, implementación (diagrama de clases), ventajas, desventajas, patrones relacionados, diversas definiciones e implementaciones obtenidas de diversas fuentes.

En el capítulo 2 “Proyectos de código abierto analizados” se describen brevemente los proyectos que se seleccionaron para el análisis los cuales son *JHotDraw* 7.5.1 y la biblioteca *Swing*. Se describen también las razones por las que se seleccionaron estos proyectos así como la estructura de los mismos.

En el capítulo 3 “Técnicas y herramientas empleadas en el análisis” como su nombre lo indica, se describen las técnicas empleadas para la búsqueda de instancias de cada uno de los patrones. Además se describen las características de la aplicación *Enterprise Architect*, el cual fue utilizado para realizar ingeniería inversa para obtener el diagrama de clases de ambos proyectos y a partir de estos y del código fuente realizar el análisis.

En el capítulo 4 “Criterios para la evaluación de las instancias de los patrones” es la parte del trabajo donde se establecen, describen y analizan los criterios para la evaluación de las instancias

de los patrones de diseño seleccionados, de acuerdo a la experiencia adquirida durante el análisis se decidió dividir los criterios en tres categorías: criterios de estructura, criterios de interrelación y otros criterios; estos últimos pueden ser cualquier criterio que no entre en las dos clasificaciones anteriores.

En el capítulo 5 “Evaluación y análisis de las instancias” es la parte del trabajo en la que se describe el análisis realizado en cada una de las instancias de los patrones de diseño seleccionados, se determina también si cada uno de los criterios fueron validados o no y finalmente se muestran las conclusiones a las que se llegó de acuerdo al análisis realizado anteriormente en cada una de las iteraciones. Además se muestran los resultados generales obtenidos de acuerdo a los resultados obtenidos en el análisis de cada una de las instancias.

Los nombres de los patrones de diseño están en inglés por convención.

Finalmente se presentan las conclusiones generales del trabajo.

Capítulo 1

INTRODUCCIÓN A LOS PATRONES DE DISEÑO Y A LOS PROYECTOS DE CÓDIGO ABIERTO ANALIZADOS

La idea detrás de los patrones de diseño es el desarrollar una forma estandarizada de representar una solución general a problemas encontrados comúnmente en el desarrollo de software. Los patrones de diseño nos dan una forma efectiva de compartir experiencias dentro de la comunidad de programación orientada a objetos.

1.1 Historia de los patrones.

La inspiración de los patrones de diseño en el desarrollo de software es atribuido a Christopher Alexander, un profesor de arquitectura en U.C. Berkeley. A finales de los 70's, publicó algunos libros en los que introduce el concepto de patrón y provee un conjunto de patrones para el diseño de la arquitectura.

Kent Beck y Ward Cunningham discutieron un conjunto de patrones de diseño de *Smalltalk* en una presentación en la conferencia OOPSLA en 1987. Asimismo escribieron un libro acerca de los patrones para el desarrollo en C++ en los años 90's.

Probablemente la mejor contribución a la popularidad de los patrones de diseño fue en 1995 con el libro "*Design Patterns: Elements of Reusable Object-Oriented Software*" (Gamma, 1996), también conocido como "*Gang of Four*" o GoF. Dicho libro introduce un conjunto de patrones y tiene ejemplos de dichos patrones en el lenguaje C++. Otro de los libros que le dio impulso a los patrones fue el libro "*Pattern-Oriented Software Architecture, A System of Patterns*" (Buschmann, 1996).

La tecnología Java fue adquiriendo popularidad al mismo tiempo que los patrones de diseño, por lo que fue inevitable que los desarrolladores de Java estuvieran interesados en aplicar los patrones de diseño en sus proyectos y en la actualidad la aplicación de los patrones de diseño en proyectos Java siguen siendo muy populares.

1.2 Patrones de comportamiento.

Los patrones de comportamiento son aquéllos que se encargan del flujo del control en un sistema. Algunas formas de organizar el control dentro de un sistema pueden lograr beneficios tanto en el mantenimiento como en la eficiencia del mismo. Los patrones de comportamiento introducen la esencia de las prácticas probadas en una forma práctica de entenderlo y asimismo brindan una forma sencilla de aplicar heurísticas. (Steling & Maassen, 2001)

El patrón de comportamiento que se analizará es el patrón *Observer*, el cual se describe a continuación.

1.2.1 Observador (*Observer*).

> *Es un patrón de comportamiento, cuyo propósito es proveer flexibilidad para que un componente envíe mensajes a los receptores que estén interesados en dicho mensaje.* (Steling & Maassen, 2001).

Otras definiciónes del patrón *Observer* son las siguientes:

> *"Patrón Observer define una dependencia "uno a muchos" entre objetos de manera que cuando cambia el estado de un objeto todos los objetos que dependen de él son notificados y actualizados automáticamente"*. (Freeman, Freeman, & Bates, 2004)

> *"El acoplamiento ligero entre dos objetos nos permite construir sistemas OO flexibles que pueden manejar cambios debido a que minimizan la independencia entre objetos".* (Freeman, Freeman, & Bates, 2004).

Aplicabilidad.

El *Observer* es generalmente apropiado cuando:

- Al menos existe un mensaje que enviar.
- Existe uno o más receptores de mensajes que pueden variar en una aplicación o entre aplicaciones.
- Este patrón es frecuentemente empleado en situaciones donde el mensaje enviado no necesita o no quiere saber cómo es que los receptores actúan con la información que éste provee, sino que lo único que le concierne al mensaje es enviar la información a los receptores necesarios.
- Pruebas. Se puede diseñar un "*Observer*" el cual pueda mostrar el comportamiento del observado.
- Desarrollo incremental. Es sencillo construir *Observers* adicionales al mismo tiempo que se desarrollan los primeros *Observers*.

Descripción.

Algunos productores de mensajes (componentes que envían mensajes) siguen un modelo simple de comunicación punto a punto, crean mensajes destinados a uno o más receptores. En estos

casos el manejo de eventos es bastante sencillo. Para otro tipo de productores de mensajes una acción puede desencadenar una serie de reacciones variables y probablemente involucra al productor de mensajes y uno o más receptores.

En el patrón *Observer*, los productores de mensajes (*Subject*, componentes observables) envían mensajes que generan eventos. Uno o más receptores de mensajes (*Observers*) reciben y actúan de acuerdo a los acontecimientos. La responsabilidad de los componentes observables es el transmitir eventos a cualquier *Observer* interesado, es decir, todos los *Observers* que se encuentren "registrados" con el componente observable. Una interfaz que "escucha" aloja un componente observable para indicar qué evento ha ocurrido y posiblemente para proveer detalles a los *Observers*.

Ventajas

El objeto observable puede ser relativamente sencillo, es decir no tiene en sí una cantidad excesiva de código para implementarse.

- Este patrón permite agregar nuevos "*Observers*" en cualquier momento que se necesite. Esto es posible debido a que, de lo único de lo que depende el "*Subject*" es de uno o más objetos que implementan la interfaz *Observer*.

- Nunca se necesitará modificar el "*Subject*" para agregar nuevos tipos de *Observers*, esto es, supóngase que se desea agregar una clase abstracta como "*Observer*" no se necesita hacer ningún cambio al *Subject* sino que todo lo que se tiene que hacer es implementar la interfaz *Observer* en la nueva clase y registrarla como *Observer*. El *Subject* únicamente liberará notificaciones a cualquier objeto que implemente la interfaz *Observer*.

- Es posible reusar *Subjects* u *Observer*s sin que unos dependan de otros, esto debido a que ambos se encuentran ligeramente acoplados. Éstos son libres de realizar cambios sin que dejen de realizar sus obligaciones como *Subject* u *Observer* dependiendo del caso.

Desventajas

El reto principal de este patrón es la implementación del modelo de mensajes, se pueden usar estrategias en las que los mensajes son específicos o generales, aunque ambas estrategias tienen desventajas. En el caso de que se implemente la estrategia de mensajes genéricos. Se vuelve más complicado determinar que está pasando para un componente observable en específico.

En el caso de que se decida implementar mensajes específicos, se tiene que el hecho de construir mensajes más específicos aumenta la cantidad de requerimientos en la construcción del componente observable. Esto se debe a que éstos deben producir un conjunto de notificaciones diseñadas bajo condiciones específicas. Además posiblemente se deberá diseñar *Observers* más complejos debido a que los *Observers* deberán manejar una gran variedad de mensajes.

Patrones relacionados.

Uno de los patrones que comúnmente se emplean para implementar la comunicación entre el *Observer* y el Observable es el patrón *Remote Proxy*.

Implementación.

La Figura 1 muestra el diagrama de clases del patrón *Observer*:

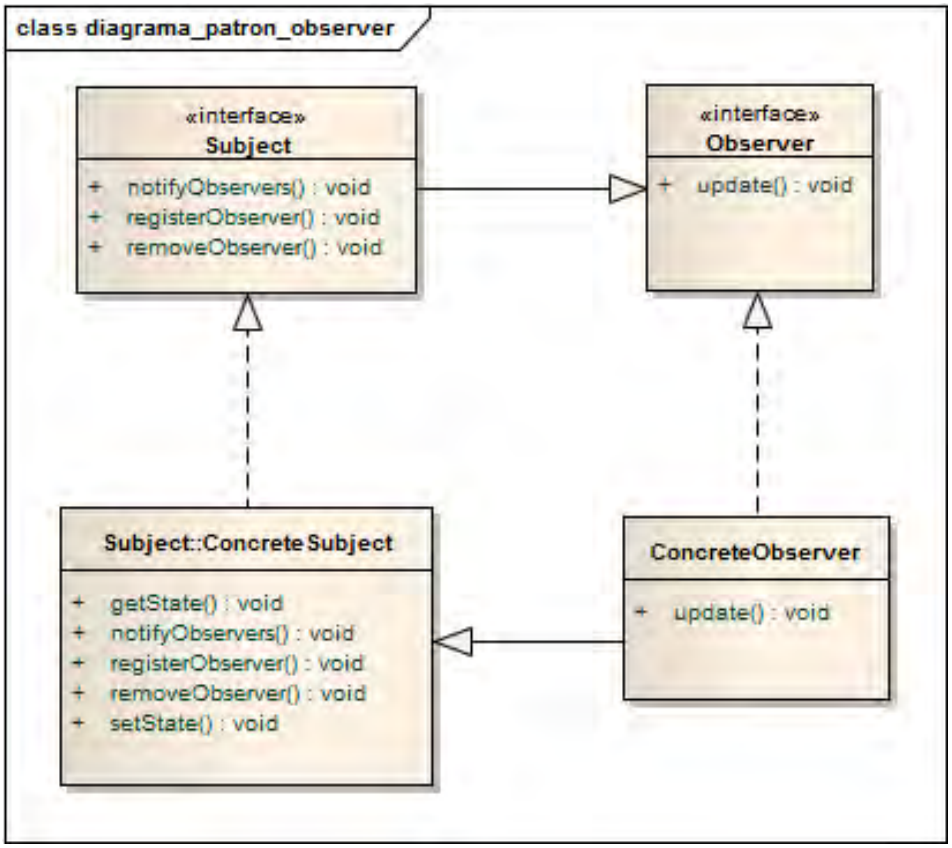

Figura 1 Diagrama de clases del patrón *Observer* (Freeman, Freeman, & Bates, 2004).

Los componentes son los siguientes:

-*Subject:* (u Observable como se encuentra en "*Applied Java Patterns*" (Steling & Maassen, 2001)) Es una interfaz que define cómo es que los clientes (*Observers)* pueden interactuar con un *Subject*. Sus métodos incluyen poner y quitar *Observers*, y uno o más métodos de notificaciones para enviar información a través de *Subject* a sus respectivos clientes.

-*ConcreteSubject:* (*ConcreteObservable* como se encuentra en "*Applied Java Patterns*" (Steling & Maassen, 2001)) La clase que provee implementaciones para cada uno de los métodos en la interfaz *Subject*, necesita mantener una colección de *Observers*.

-*Observer*: Es una interfaz que define cómo es que los clientes/*Observers* pueden interactuar con un *Subject*. Sus métodos incluyen poner y quitar *Observers*, y uno o más métodos de notificaciones para enviar información a través de un *Subject* a sus respectivos *Observers*.

-*ConcreteObserver*: Implementa la interfaz Observer y determina en cada método de implementación como responder a los mensajes recibidos de *Subject*.

Este patrón permite que exista un ligero acoplamiento entre dos objetos, de manera que puedan interactuar entre ellos a pesar de que un objeto tiene muy poco conocimiento del otro y viceversa. Este acoplamiento se debe a que lo único que el "*Subject*" sabe del *Observer* es que implementa una interfaz (la interfaz *Observer*), no necesita saber la clase concreta del *Observer*, qué hace, o algo más que la interfaz.

Al comparar las deficiones de diferentes autores del patrón *Observer* se puede notar que no difieren en mucho, las diferencias son únicamente en cómo se nombra a la interfaz *Observable* y *Subject* que en realidad son la misma nombradas de diferente forma en cada bibliografía.

La diferencia radica en la forma en que el *Subject* se comunica con el *Observer*, por ejemplo, comparando la definiciones de los libros "*Applied Java Patterns*" (Steling & Maassen, 2001) y "*Head First Design Patterns*" (Freeman, Freeman, & Bates, 2004) en el caso de la primer definición se puede notar que la comunicación entre el *Observer* y el *Observable* (o *Subject*) se realiza entre la interfaz *Observer* y la clase *ConcreteObservable*, en cambio en la segunda definición se puede notar que la comunicación se realiza entre las interfaces *Subject* y *Observer* y las clases *ConcreteSubject* y *ConcreteObserver*.

Como se puede observar, las descripciones de los patrones varían de acuerdo al autor, debido a que cada autor tiene su propia perspectiva de un mismo concepto.

Para el caso que se estudiará se tomó como referencia la bibliografía "*Head First Design Patterns*" (Freeman, Freeman, & Bates, 2004), debido a que las definiciones que se establece en dicha bibliografía son prácticas y sencillas de implementar.

1.3 Patrones de estructura.

Los patrones de estructura son aquellos que describen una manera efectiva de dividir y combinar los elementos de una aplicación. La forma en que los patrones de estructura afectan a las aplicaciones puede ser de diferentes formas.

Por ejemplo, el patrón *Adapter* permite que dos sistemas incompatibles se puedan comunicar entre sí.

Algunos patrones de estructura son:

- ***Adapter**: Este patrón actúa como un intermediario entre dos clases, transforma la interfaz de una clase de manera que pueda ser usada por la otra.*
- ***Bridge:** Es implementado para dividir un componente complejo en dos componentes separados pero relacionados por jerarquías inherentes: la abstracción y la implementación interna.*
- ***Composite:** Es implementado para crear formas flexibles de construir árboles de cualquier grado de complejidad y al mismo tiempo lograr que cada elemento del árbol pueda trabajar como una interfaz uniforme.*
- ***Decorator:** Es implementado para proveer una forma flexible de agregar o quitar componentes a otro componente sin cambiar la estructura interna de ese componente, la apariencia externa o su función.*

- ***Facade:*** *Es implementado para proveer una interfaz simple a un conjunto de subsistemas o un subsistema complejo.*
- ***Flyweight:*** *Es implementado en los casos en los que se necesite reducir el número de objetos de muy bajo nivel u objetos con mucho detalle, esto lo realiza compartiendo objetos.*
- ***Half-Object Plus Protocol (HOPP)****: Para proveer una sola entidad que se encuentre alojada en dos o más espacios de memoria.*
- ***Proxy****: Para proveer una representación de otros objetos, las razones por las que se desee tener una representación diferente pueden ser muchas, de entre ellas: el acceso al objeto, velocidad, seguridad.*
  (Steling & Maassen, 2001)

En la siguiente sección se describirá el patrón *Adapter* que es uno de los patrones que se analizarán.

1.3.1 Adaptador (*Adapter*).

> *"El patrón Adapter transforma la interfaz de una clase en otra interfaz que el cliente espera encontrar. El Adapter permite que las clases que son parcial o totalmente incompatibles trabajen a la par".* (Freeman, Freeman, & Bates, 2004).

Otra definición de este patrón es la siguiente:

> *"Es un patrón de estructura, su propósito es actuar como un intermediario entre dos clases modificando la interfaz de una clase de manera que pueda ser usada por otra".* (Steling & Maassen, 2001).

Aplicabilidad.

El *Adapter* es generalmente apropiado cuando:

- Se desea usar un objeto en un ambiente en el que se espera una interfaz diferente de la interfaz del objeto.
- La traducción de una interfaz entre las múltiples fuentes puede ser requerida.
- Un objeto debe actuar como un intermediario de un grupo de clases, y no es posible conocer qué clases serán usadas antes de la ejecución.

Descripción.

En ocasiones se desea usar una clase en una nueva aplicación sin volver a codificar para que coincida con el nuevo entorno. En estos casos, se debe diseñar una clase *Adapter* para que actúe como un traductor. Estas clases reciben llamadas del entorno y modifican dichas llamadas para que sean compatibles con una clase *Adaptee.*

Ambientes típicos donde un *Adapter* puede ser útil incluyen aplicaciones que soportan comportamientos tipo "*plug-in*" como gráficos, textos o editores. Los navegadores web son otro ambiente en donde el *Adapter* puede ser útil. Aplicaciones que tienen que ver con internacionalizaciones, es decir, aplicaciones en las que se requiera un determinado componente de software dependiendo de la región donde se desee utilizar, así como aplicaciones que usan componentes que son añadidos sobre la marcha.

La forma en que este patrón funciona es la siguiente:

- El cliente hace una petición al *Adapter* llamando a un método de sí mismo usando la interfaz que implementa.
- El *Adapter* traduce esa petición en una o más llamadas al *Adaptee* usando la interfaz *Adaptee*.
- El cliente recibe el resultado de la llamada y nunca se entera de si existe algún adaptador haciendo la traducción.

Implementación.

La Figura 2 muestra el diagrama de clases del patrón *Adapter*:

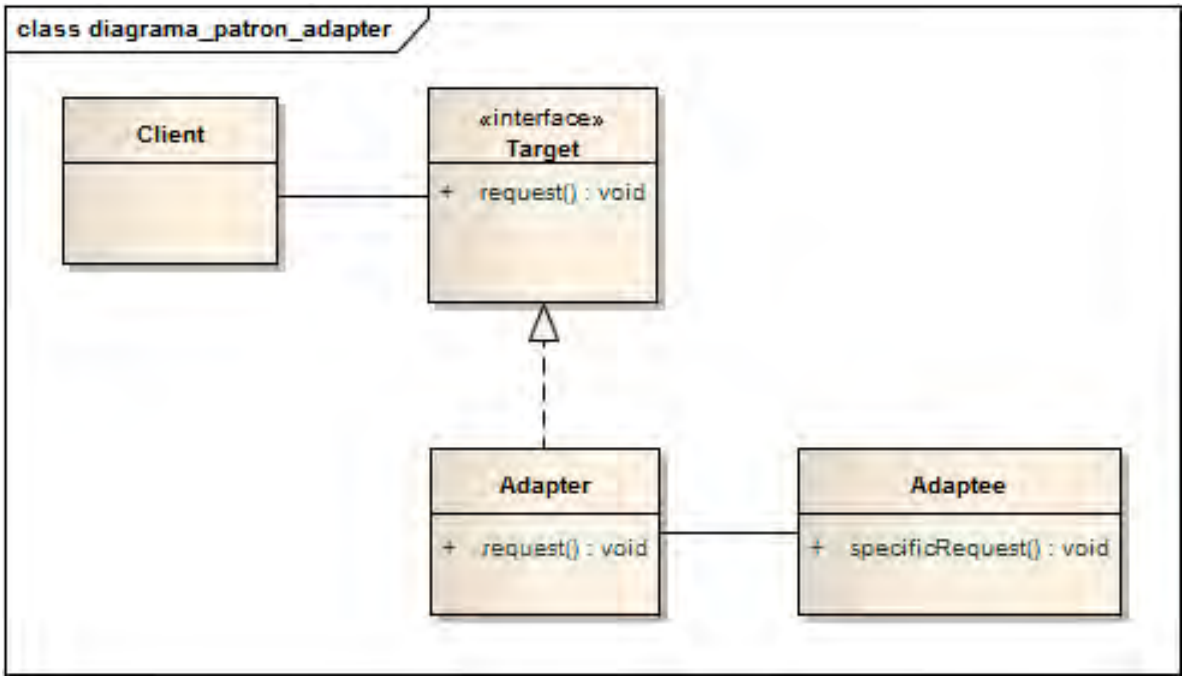

Figura 2, Diagrama de clases del patrón *Adapter* (Freeman, Freeman, & Bates, 2004).

Los componentes (roles) son los siguientes:

-*Client*: Es la entidad que es incompatible.

-*Tarjet*: Interfaz con la que el *Client* se comunica de manera que el *Adapter* reciba una llamada o petición a través de éste.

-*Adapter*: Implementa la interfaz *Target* y mantiene una instancia del *Adaptee* de manera que el *Client* pueda comunicarse con el *Adaptee* a través de éste.

-*Adaptee*: Entidad con la que se desea comunicar el *Client*.

El objetivo principal de la aplicación de este patrón es el desacoplar el cliente de la interfaz implementada, y también es muy común encontrarla en los casos en que se espera que la interfaz cambie con el tiempo. El *Adapter* encapsula esos cambios de manera que el cliente no tiene que ser modificado cada vez que éste necesita operar con una interfaz diferente.

Una de las ventajas de la aplicación de este patrón es que se puede usar un *Adapter* con cualquier subclase del *Adaptee*. Además nótese que este patrón forza a que el cliente se comunique con una interfaz pero no tiene que implementar ninguna ni tampoco deberá ser modificado en ningún momento.

Cabe destacar que el patrón *Adapter* es uno de los patrones que se puede aplicar en un nivel más alto en comparación con el resto de los patrones, es decir, por lo general no se aplica para comunicar objetos o clases, sino que, permite en muchas ocasiones la comunicación entre dos aplicaciones que funcionan en su propio entorno por separado. Es por lo anterior que su implementación puede variar significativamente entre un proyecto y otro, y a su vez ésta es la razón por la que la definición y el diagrama de clases de este patrón varía tanto en cada una de las bibliografías.

1.3.2 Decorador (Decorator).

"El patrón *Decorator* agrega responsabilidades adicionales a uno o más objetos de manera dinámica. Los decoradores proveen una alternativa flexible para crear subclases de manera que puedan extender funcionalidades". (Freeman, Freeman, & Bates, 2004)

El objetivo principal de este patrón es proveer una forma en que las clases puedan ser fácilmente extendidas para incorporar nuevo comportamiento sin modificar el código ya existente obteniendo así diseños que son resistentes al cambio y al mismo tiempo lo suficientemente flexibles para adoptar nuevas funcionalidades cuando existan cambios en los requerimientos.

Lo descrito anteriormente permite tener la combinación entre rigidez y flexibilidad que en muchas situaciones se necesitan, al aplicar este patrón se debe de ser cuidadosos al momento de escoger las partes de código que se desea extender. Si esto no se realiza con cuidado se pueden cometer errores como diseñar código innecesario y complejo, y en consecuencia difícil de entender.

Otra definición del patrón *Decorator* es la siguiente:

> *"Es un patrón de estructura, su propósito es el proveer una manera flexible de aumentar o quitar una funcionalidad sin cambiar la apariencia externa o función".* (Steling & Maassen, 2001).

Aplicabilidad.

El *Decorator* es generalmente apropiado cuando:

- Se requiere hacer cambios dinámicos que son transparentes a los usuarios, sin las restricciones de las subclases.
- Las capacidades de los componentes pueden ser añadidos o quitados en tiempo de ejecución.
- Existen diversas características independientes que se deben aplicar dinámicamente, además, éstas se pueden utilizar en cualquier componente.

Descripción.

El patrón decorador funciona permitiendo que las capas se agreguen o eliminen de un objeto base. Cada capa puede proveer comportamiento (métodos) y estado (variables) para aumentar el objeto base. Las capas pueden ser encadenadas y asociadas libremente con este patrón, permitiéndonos así, crear objetos con comportamiento de un conjunto de bloques de construcción muy sencilla.

Características importantes.

Los *decorators* tienen el mismo súper tipo que los objetos que decoran.

- Se pueden usar uno o más decoradores para envolver a un objeto, entiéndase por envolver al proceso de agregarle funcionalidad a un objeto dependiendo de las necesidades del mismo.
- Dado que el *Decorator* tiene el mismo súper tipo que el objeto que decora, se puede pasar un objeto decorado en lugar del objeto original (el objeto ya envuelto).
- El *Decorator* agrega su propio comportamiento antes y/o después de delegar funcionalidades al objeto que decora.
- Los objetos pueden ser decorados en cualquier momento que se requiera, de manera que se pueden decorar objetos dinámicamente en tiempo de ejecución con cuantos *decorators* se desee.

Existen muchas clases en las bibliotecas de Java en las que se aplica el patrón *Decorator*, pero en especial los paquetes *java.io* (de entrada y salida) son las clases en las que este patrón es aplicado constantemente. A continuación se verá un ejemplo, es un conjunto de objetos que utilizan el patrón *Decorator* para agregar funcionalidad para leer datos de un archivo:

FileInputStream. Es una clase que maneja archivos de texto para que puedan ser leídos. Es el componente decorado.

BufferedInputStream. Esta clase funciona como el *ConcreteDecorator*. Esta clase agrega funcionalidades de dos formas, una de ellas es almacenar en un buffer la información de entrada para mejorar el procedimiento, y además agrega el método *readLine()* a la interfaz para leer una entrada basada en caracteres una línea a la vez.

LineNumberInputStream. Es también un concrete *Decorator*. Esta clase adiciona la habilidad de contar las líneas al mismo tiempo que lee la información.

BufferedInputStream y *LineNumberInputStream* extienden de la clase *FilterInputStream* la cual actúa como la clase abstracta *Decorator*.En la Figura 3 se muestra la representación de la aplicación del patrón Decorator en algunas clases de la plataforma Java:

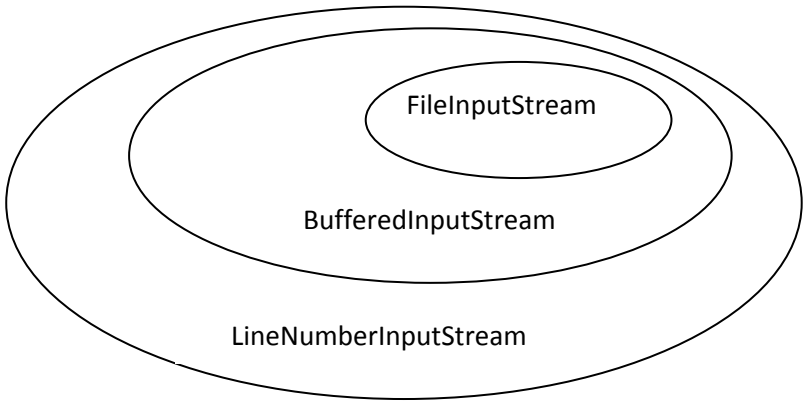

Figura 3, Ejemplo de la aplicación del patrón *Decorator* en la plataforma Java (Freeman, Freeman, & Bates, 2004)

Como se puede notar en la Figura 3 las clases concretas envuelven agregando funcionalidad a la clase *FileInputStream*.

En el ejemplo anterior se puede notar también una de las desventajas de este patrón, los diseños utilizando este patrón comúnmente resultan muchas clases pequeñas lo cual puede resultar abrumador para los desarrolladores, es por lo anterior que este patrón debe de usarse cuidadosamente.

Implementación.

La Figura 4 muestra el diagrama de clases del patrón *Decorator*:

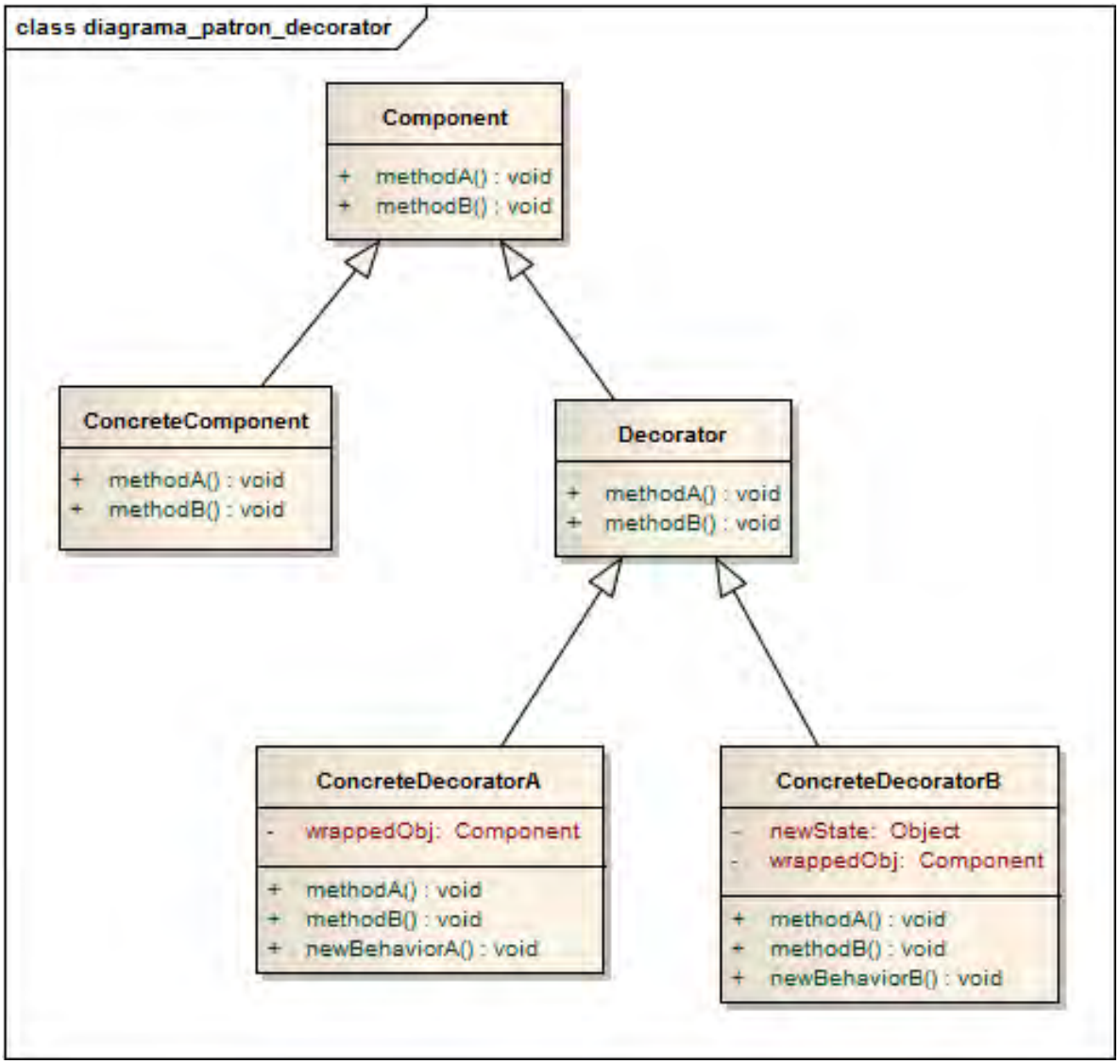

Figura 4, Diagrama de clases del patrón *Decorator* (Freeman, Freeman, & Bates, 2004).

Los componentes (roles) que se muestran en la Figura 4 se describen a continuación:

Component- Representa el componente que contiene el comportamiento general, puede ser una clase abstracta o una interfaz.

ConcreteComponent- Es la clase que implementa la interfaz *Component* o hereda de ella, de forma que pueda existir más de un *Component*.

Decorator- Define el comportamiento estándar esperado de todos los *Decorators*. *Decorator* puede ser una clase abstracta o una interfaz. El *Decorator* provee soporte para la contención, es decir, mantiene una referencia al *Component*, el cual puede ser un *ConcreteComponent* u otro *Decorator*.

ConcreteDecorator- Cada subclase de *Decorator* necesita soporte para el encadenamiento. Detrás del requerimiento base, cada *Decorator* puede definir métodos adicionales y/o variables para extender el componente.

1.4 JHotDraw 7.5.1

Una forma de reducir el tiempo de desarrollo de software y, al mismo tiempo, mejorar la calidad del software es utilizando un *framework*. Los *frameworks* son diseñados para reutilizar componentes ya que en ellos se pueden encontrar componentes de software ya construidos que se pueden utilizar como bloques para construir otra aplicación y, a su vez, se pueden emplear patrones de diseño para modelar la arquitectura del software.

> *JHotDraw es un GUI framework de código abierto escrito en Java. JHotDraw fue diseñado para dibujar y crear gráficos técnicos así como estructurados. Además, ofrece un soporte mucho mejor para los editores de aplicaciones para propósitos relacionados con la creación de gráficos.*
>
> *El objetivo principal de su creación fue el realizar un "ejercicio de diseño". Es por lo anterior que su diseño se llevó a cabo implementando varios patrones de diseño, reconocidos por los beneficios que conlleva su implementación.*
>
> (Thomas Eggenschwiler, 2007)

Lo descrito anteriormente es la razón por la que se decidió analizar este proyecto, los patrones de diseño que se van a analizar en este caso son *Observer*, *Decorator* y *Adapter*, los cuales fueron implementados en *JHotDraw*.

JHotDraw es un proyecto que se ha ido mejorando con el tiempo y han sido muchos desarrolladores quienes han estado involucrados en este proceso. Sin embargo, los autores originales de este proyecto fueron Erich Gamma y Thomas Eggenschwiler.

Las razones por las que JHotDraw es un proyecto de código abierto son las siguientes:

- *Lograr que una cantidad importante de desarrolladores se interesen por este framework y lo utilicen.*
- *Crear nuevas aplicaciones utilizando JHotDraw.*
- *Lograr que el desarrollo de aplicaciones realizadas con JHotDraw permita, a su vez, la influencia hacia los usuarios de este framework para mejorar y al mismo tiempo continuar con el desarrollo del framework mismo.*
- *Que los desarrolladores puedan agregar nuevas funcionalidades, elementos o componentes para mejorar el proyecto inicial.*
- *Impulsar el desarrollo de software.*
- *Identificar nuevos patrones de diseño así como identificar los componentes que pueden reconstruirse.*
- *Para analizar la relevancia e importancia que tiene el desarrollo de nuevas API para JHotDraw.*
- *Para crear un ejemplo de un buen diseño de un framework, utilizando patrones de diseño.*

(Thomas Eggenschwiler, 2007) y (Infoworld Inc, 2011).

1.5 Swing.

Swing es una biblioteca gráfica escrita en Java. Es un conjunto de herramientas que incluye un Interfaz de Usuario Gráfica (GUI, por sus siglas en inglés). Esta biblioteca es parte de *Java Foundation Classes* (JFC) y está destinada a programadores Java.

La principal razón por la que se creó Swing fue para construir un conjunto de componentes GUI extensibles de manera que los desarrolladores puedan crear componentes gráficos potentes de forma rápida y eficiente para aplicaciones comerciales.

Algunos de los componentes (*widgets*) que se pueden encontrar en esta biblioteca son cajas de texto, botones, tablas, entre otros.

Una de las ventajas que Swing tiene sobre *Abstract Window Toolkit* (AWT) es que todos los componentes de Swing corren igual en todas las plataformas y, en cambio, AWT está ligado a un sistema de ventanas de la plataforma subyacente. Swing no utiliza las facilidades de las plataformas nativas sino que, por el contrario, se aproxima a emularlos teniendo así un comportamiento uniforme en todas las plataformas.

La desventaja que tiene Swing es que, debido a que sus componentes son ligeros, su ejecución puede ser lenta.

En diciembre de 1996 *Netscape Communications Corporation* anunció la primera liberación de las *Internet Foundation Classes* (IFC), las cuales eran unas bibliotecas gráficas para Java, estas bibliotecas no incluían *Swing* como lo conocemos en la actualidad, sin embargo, son similares ya que a partir de éstas se creó *Swing*.

En abril de 1996 *Sun Microsystems* y *Netscape Communications Corporation* anunciaron que querían crear las *Java Foundation Classes* (JFC) combinando las IFC con otras tecnologías.

Posteriormente la creación de la biblioteca *Swing* logró introducir mecanismos que permitían que cada componente, originalmente creados por IFC, pudiera ser alterado sin hacer cambios sustanciales al código de la aplicación.

La biblioteca *Swing* favorece los componentes gráficos relativos, es decir, aquéllos que especifican las relaciones posicionales entre componentes; y desfavorece a los componentes gráficos absolutos, es decir, aquéllos componentes que especifican la posición, el tamaño exacto de los componentes. Las aplicaciones que utilizan la biblioteca Swing trabajan y se visualizan de forma correcta. Es por lo anterior que la biblioteca se pensó para favorecer los componentes gráficos relativos, ya que así, se puede lograr una visualización correcta sin tener que preocuparnos por aspectos como el color, fuentes, lenguajes, tamaños o dispositivos de entrada/salida de sistemas subyacentes.

La desventaja de que *Swing* favorezca a los componentes relativos es que esto provoca que el diseño de la pantalla sea complicado.

Una de las razones por las que se decidió analizar esta biblioteca Java es porque *Swing* utiliza un modelo publicador-suscriptor. Para los fines de este trabajo, este modelo no es más que la aplicación del patrón *Observer* en la creación de esta biblioteca. Este modelo funciona de la siguiente manera:

Los *listeners* (escuchadores) son los elementos que están a la espera de que ocurra un evento específico, es decir, los *Observers* (observadores); por otra parte están los publicadores, es decir, los *Subject* (sujetos) quienes se encargan de notificar a los observadores que un determinado evento ha ocurrido, algunos ejemplos de eventos son: presionar un botón o ingresar texto.

La utilización de la biblioteca *Swing* permite que en las aplicaciones exista un acoplamiento débil, contribuyendo así a que *Swing* tenga una curva de aprendizaje pronunciada.

1.5.1 Arquitectura

Los componentes Swing poseen un diseño Modelo-Vista-Controlador (MVC). En este trabajo no se analizará el patrón de diseño MVC, sin embargo, es importante describir la arquitectura general de la biblioteca. Como se sabe, el MVC se compone de tres partes:

- Un Modelo que representa los datos de la aplicación.
- La Vista que es la representación visual de esos datos.
- Un Controlador que controla los datos introducidos en la vista y, a su vez, convierte esos datos en cambios en el modelo de datos.

Pero posteriormente los desarrolladores de Swing descubrieron que esta forma de separar los componentes no funcionaba correctamente debido a que los componentes de la vista y el controlador requieren de un acoplamiento fuerte. Debido a lo anterior se decidió colapsar estas dos partes en una sola llamada *UI (User-Interface) object*, la Figura 5 muestra la arquitectura de la biblioteca Swing como quedó finalmente.

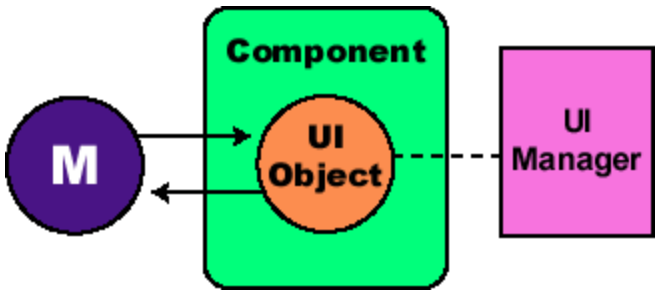

Figura 5, Modelo de la arquitectura de la biblioteca Swing (Oracle Corporation, 2010).

Como se puede ver en el diagrama anterior, finalmente Swing se basó en un modelo parecido al MVC pero no estrictamente el mismo, este modelo a veces es llamado “Arquitectura de Modelo Separable”.

Capítulo 2

TÉCNICAS Y HERRAMIENTAS EMPLEADAS EN EL ANÁLISIS

2.1 *Enterprise Architect* 7.5.

2.1.1 Breve descripción.

Enterprise Architect (EA) es una herramienta para diseño y análisis UML. Es un software desarrollado por *Sparx Systems* que puede ser empleado en el desarrollo de software desde el levantamiento de requerimientos, análisis, modelos de diseño, pruebas y mantenimiento.

"EA es una herramienta multiusuario, basada en Windows, diseñada para ayudar a construir software robusto y fácil de mantener. Ofrece salida de documentación flexible y de alta calidad." (Sparx Systems Pty Ltd., 2007).

Algunas características son:

- Soporta diagramas de comportamiento como: casos de uso, actividades, estado, interacción, secuencia y comunicación.
- Soporta diagramas de estructura: paquetes, clases, objetos, composición, componentes y despliegue.
- Posee una interfaz de usuario intuitiva.
- Soporte para transformaciones MDA (Arquitectura Dirigida por Modelos), el cual permite transformar elementos simples en complejos.
- Ingeniería inversa y directa, es posible generar código fuente a partir de los diagramas de clases. Por el contrario, generar los diagramas de clases a partir de código fuente.
- Modelado de base de datos.
- Es posible implementar *Plug-ins* para establecer vínculos con Visual Studio.NET y Eclipse.
- Soporte para control de versiones.
- Soporte para la administración de proyectos: detallar ítems de riesgo, unir tipos de métricas especializadas a cualquier elemento del modelo, métricas de casos de usos.
- Soporte para pruebas: de unidad, de integración, de sistema, de aceptación, escenarios.

2.1.2 ¿Por qué utilizar *Enterprise Architect*?

EA posee todas las características necesarias para la búsqueda, análisis y evaluación de los patrones de diseño en el presente proyecto, es decir, con la ayuda de EA se puede realizar ingeniería inversa lo cual permite analizar la estructura del código a partir de los diagramas de clases y no del código en sí, lo cual permite que el análisis se lleve a cabo de forma más rápida y eficaz; EA posee herramientas de gran utilidad como el "Buscador", el cual busca en nombres de archivos (clases) y dentro de ellos coincidencias de palabras clave. Además EA permite ver el código fuente con marcas y con código específico resaltado, como por ejemplo, nombres de funciones tipos de dato, etc, lo cual permite que el análisis sea más rápido y sencillo.

2.2 Técnicas empleadas y utilización de *Enterprise Architect* en el proyecto.

EA es un software con diversos componentes y herramientas para la administración de proyectos y desarrollo de software, no fue necesario emplear alguna otra herramienta. A continuación se describirán las técnicas empleadas para el desarrollo de este proyecto:

Para los fines de este proyecto, EA se empleó para realizar ingeniería inversa a partir del código fuente del proyecto *JHotDraw* y *Swing*, es decir, a partir del código fuente se generaron los diagramas de clases.

Una de las técnicas que se emplearon en este proyecto es el buscar palabras clave, como por ejemplo la palabra "*Listener*", la cual se sabe por los autores que es una palabra que se usa en ambos proyectos para darle nombre a las clases que funcionan como *Observers*. De esta forma fue rápido y sencillo encontrar las instancias del patrón *Observer,* así como del patrón *Decorator* y *Adapter* buscando las palabras clave *"Decorator", "decorate"*, "*Adapter*" respectivamente.

Las técnicas empleadas en el presente trabajo podrían no ser útiles en los proyectos en los que no se les haya dado un nombre a las clases que estuviera relacionado con la entidad que representa. Por ejemplo, en el caso de una clase que sea *Observer*, existe la posibilidad de que se le haya dado un nombre diferente de las palabras "*Listener*", "*Observer*", "*Catcher*", etc.; por lo que su búsqueda no podría realizarse a través de esta técnica y deberá de utilizarse otras técnicas como por ejemplo, la búsqueda de métodos que coincidan con la definición de estos o el análisis de las interrelaciones entre clases.

2.2.1 Patrón Observer.

Una vez encontrados las clases *Observer* se procedió a buscar clases con nombre similar al *Observer*, por ejemplo, en una de las instancias del patrón *Observer* el *Subject,* en donde el *Observer* es *"CompositeFigureListener",* tiene un nombre similar al *Observer* (*CompositeFigure*) y esto sucede en casi todas las instancias de este patrón. Siguiendo este procedimiento fue más rápido encontrar las instancias de este patrón.

Una vez obtenidos los *Observers* y *Subjects* se verificó que los *Subject* tuvieran los métodos *registerObserver()*, *removeObserver()* y *notifyObserver()* y a su vez, los *Observer* tuvieran el método *update()*.

Posteriormente para encontrar los *ConcreteSubject* y *ConcreteObserver*, se agregaron todas las subclases de estos al diagrama de clases, en general los *Observers* y *Subjects* son interfaces por lo que las subclases son implementaciones de éstas. Completando así el diagrama de clases para posteriormente establecer los criterios apropiados para una buena aplicación.

Una vez obtenidas las instancias, después del análisis de las mismas se determinó que los criterios deberán ser establecidos de acuerdo a la siguiente clasificación:

- Criterios de estructura.
- Criterios de interrelación.
- Otros criterios.

Estos criterios se describen con mayor detalle en el capítulo 5.

2.2.2 Patrón Decorator.

Una vez determinadas las clases que son posibles *decorators* se procedió a analizar todas las interrelaciones que poseen estos.

En algunos casos se observó que muchos de los nombres de las clases que tienen la palabra "*decorator*" o similar no son instancias del patrón *Decorator*, por lo que se decidió emplear otra estrategia, la cual se describirá a continuación:

En el proyecto *Swing*, en cada uno de los componentes, (véase el apartado 2.2 para mayor detalle acerca de los componentes de la biblioteca *Swing*) se analizó su estructura y las interrelaciones que poseen para encontrar alguna coincidencia con la estructura e interrelaciones propias del patrón *Decorator* (como se muestra en el apartado 4.2.3 y 4.2.4).

Posteriormente se buscó un posible *Component*, es decir, una clase de la que herede o una interfaz que implemente el posible *Decorator*. Es importante mencionar que esta técnica no se utilizó en todas las instancias debido a que algunas instancias actúan como decoradores pero no como una instancia en sí del patrón *Decorator*, por lo que la estructura en algunas instancias no es propia de una instancia de este patrón, esto se explica más a detalle en el capítulo 5.

Otra técnica que se utilizó fue el análisis de los métodos que agregan nuevo comportamiento a los *ConcreteDecorators* de forma que si estos no existían esto se pudiera ver reflejado en la verificación de los criterios y conclusiones en cada una de las instancias.

2.2.3 Patrón Adapter.

De forma similar al patrón *Decorator* se observó que muchos de los nombres de las clases que tienen la palabra "*adapter*" o similar no son instancias del patrón *Adapter*, por lo que también se decidió emplear otra técnica para encontrar instancias de este patrón.

Se realizó la búsqueda de clases que pudieran tener el mismo propósito pero que no pudieran tener comunicación si no es por medio de un *Adapter*, como por ejemplo la instancia *ADP_ComponentUI*, la cual tiene clases (*Target* y *Adaptees*) que tienen comunicación consigo mismo por medio de los *Adapters plaf::ButtonUI*, *plaf::ColorChooserUI* y *plaf::ComboBoxUI*, véase el apartado 5.1.4 para mayor detalle acerca de esta instancia.

Capítulo 3

CRITERIOS PARA LA EVALUACIÓN DE LAS INSTANCIAS DE LOS PATRONES

En el presente capítulo se establecerán y analizaran los criterios para la evaluación de las instancias de los patrones de diseño. Estos criterios se obtuvieron a partir de la estructura del diagrama de clases de los patrones de diseño, así como del análisis de las instancias encontradas.

Los criterios obtenidos se analizarán y evaluarán en tres categorías de acuerdo a la experiencia obtenida en cada una de las iteraciones, estas categorías se describirán a continuación:

- Criterios de estructura: Estos criterios muestran todos los roles que la instancia deberá tener así como los métodos que cada uno de los roles debe tener.
- Criterios de interrelación: Estos criterios muestran las relaciones que deben existir entre cada uno de los roles, estas relaciones pueden ser implementación de una interfaz, herencia o alguna llamada de un método en otro.
- Otros criterios: Estos criterios son principalmente los criterios que se consideran importantes pero que no entran en ninguna de las dos clasificaciones anteriores, pueden ser criterios dinámicos, en los que se describe algún comportamiento esperado de algún rol o algún método.

3.1 Observer.

3.1.1 Roles.

Observer, ConcreteObserver, Subject, ConcreteSubject.

3.1.2 Descripción de la solución.

Modificar el estado de un objeto cuando un evento determinado ocurra y, a su vez, notificar y actualizar todos los objetos dependientes de este objeto de la ocurrencia de este evento.

3.1.3 Criterios.

La Figura 6 muestra los criterios de estructura, de interrelación y otros criterios. Cada uno de los elementos numerados deberán existir en la instancia candidata del patrón.

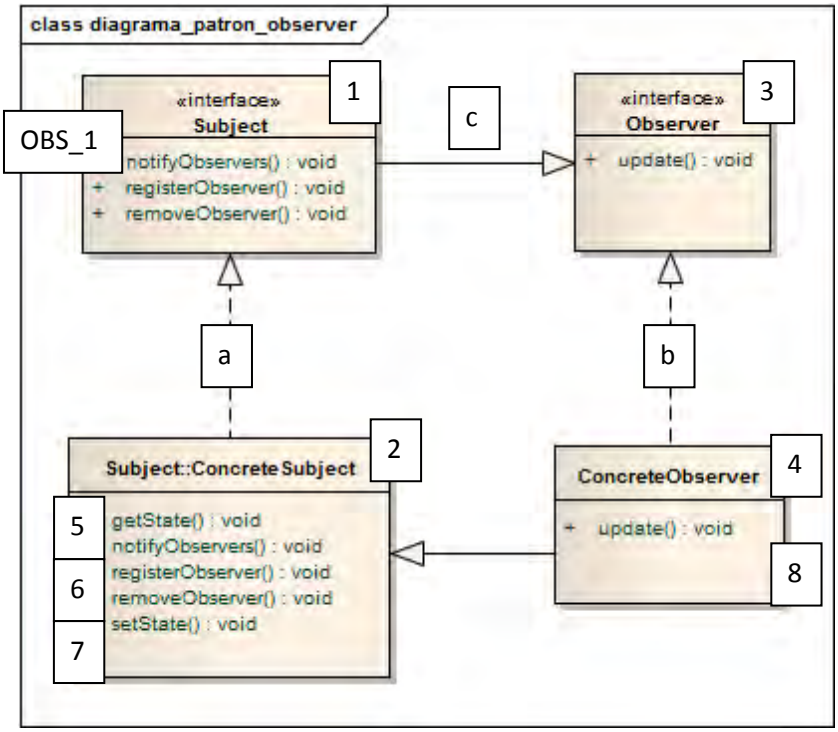

Figura 6, Diagrama general de criterios del patrón *Observer*.

Nota: Los *getState()* y *setState()* pueden ser parte de la instancia para guardar y obtener el estado del *Subject*. Sin embargo, estos métodos no son considerados como parte de los criterios.

Criterios de estructura.

La Tabla 1 muestra la relación de los números de la Figura 6 con los roles o métodos que forman parte de los criterios de estructura. Todos los elementos mencionados deberán existir en cada una de las instancias de este patrón, como se puede notar se tienen tres tipos de elementos:

- Rol: Es la clase que tendrá un papel específico (descrito a detalle en la sección 1.2.1) dentro de la instancia.
- Método: Son los métodos que deberán tener cada una de las clases de acuerdo a la Figura 6 y la descripción de los métodos en la sección 1.2.1.
- Atributo: Son los atributos que deberán tener cada una de las clases de acuerdo a la Figura 6 y la descripción de los métodos en la sección 1.2.1.

Elemento de la Figura 6	Nombre de elemento	Tipo de elemento
1	Subject	Rol
2	ConcreteSubject	Rol
3	Observer	Rol

4	ConcreteObserver	Rol
5	notifyObservers()	Método
6	registerObserver()	Método
7	removeObserver()	Método
8	update()	Método

Tabla 1, Relación de los elementos de estructura del patrón *Observer*.

Criterios de interrelación.

La Tabla 2 muestra la descripción de los criterios de interrelación del patrón *Observer*:

Elemento de la Figura 6	DESCRIPCIÓN
a	El *ConcreteSubject* debe implementar o heredar del *Subject* o, en su defecto, el *ConcreteSubject* y el *Subject* pueden ser representados (colapsados) como una sola clase.
b	El *ConcreteObserver* debe implementar o heredar del *Observer* o, en su defecto, el *ConcreteObserver* y el *Observer* pueden ser representados (colapsados) como una sola clase.
c	El *Subject* debe hacer una llamada al *Observer* en el método *notifyObservers()* llamando el método *update()*.

Tabla 2, Criterios de interrelación del patrón *Observer*.

Otros criterios.

La Tabla 3 muestra la descripción de otros criterios del patrón *Observer*:

Elemento de la Figura 6	DESCRIPCIÓN
OBS_1	El mensaje o mensajes enviados por el método o métodos *notifyObservers()* deberá(n) ser enviado(s) a todos los *Observers* del *Subject*.

Tabla 3, Otros criterios del patrón *Observer*.

Observaciones importantes.

- El *Subject* puede tener uno o más *Observers* y, a su vez, un *Observer* puede tener uno o más *Subjects*, se tomará como instancia a un *Subject* con sus respectivos *Observers* pero estos *Observers* podrán aparecer en alguna otra instancia del patrón *Observer*.
- Pueden existir en una misma instancia clases que tenga el rol de *Observer* o *ConcreteObserver* y al mismo tiempo tengan el rol de *Subject* o *ConcreteSubject*. Sin embargo, esta situación no determina si una instancia es válida o no.

3.2 Decorator.

3.2.1 Roles.

Decorator, ConcreteDecorator, Component, ConcreteComponent.

3.2.2 Descripción de la solución.

Agregar dinámicamente nuevas responsabilidades a un objeto de manera que este objeto pueda tener una o más responsabilidades de las que tenía inicialmente y, en consecuencia, un componente pueda tener dinámicamente un comportamiento diferente dependiendo de las necesidades que puedan existir en un tiempo determinado.

3.2.3 Criterios.

La Figura 7 muestra los criterios de estructura, de interrelación y otros criterios, cada uno de los elementos numerados deberán existir en la instancia candidata del patrón.

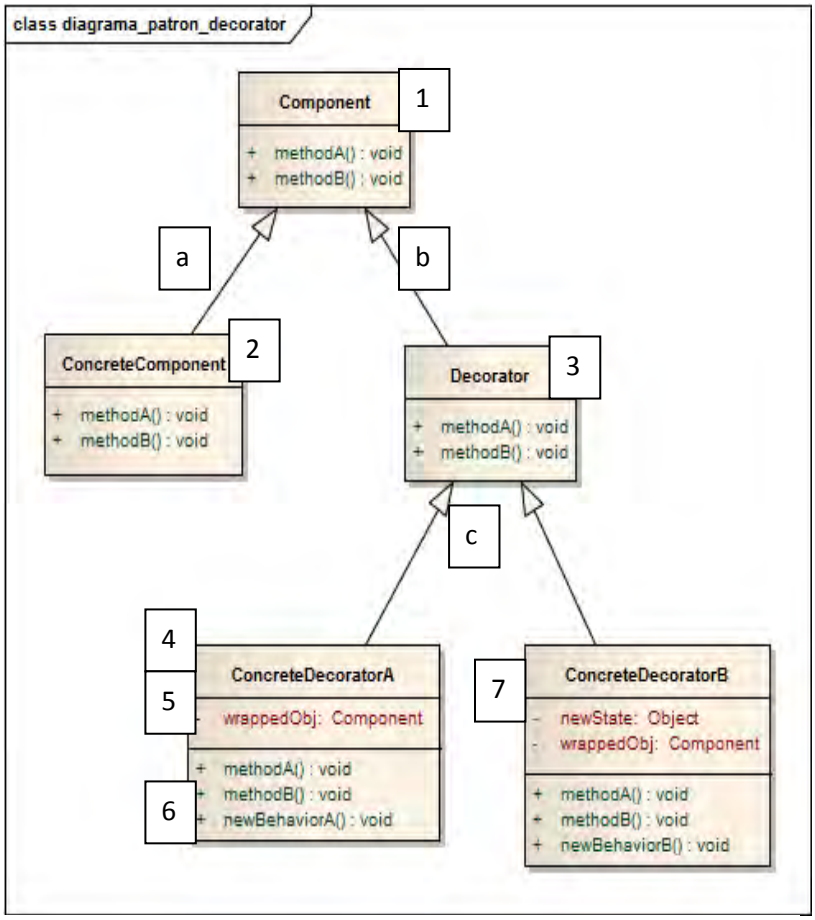

Figura 7, Diagrama general de criterios del patrón *Decorator*.

Criterios de estructura.

La Tabla 4 muestra la relación de los números de la Figura 7 con los roles o métodos que forman parte de los criterios de estructura. Todos los elementos mencionados deberán existir en cada una de las instancias de este patrón. La descripción de los criterios de estructura y la forma en la que analizan es similar a la descripción de los criterios del patrón *Observer*.

Elemento de la Figura 7	Nombre de elemento	Tipo de elemento
1	Component	Rol
2	ConcreteComponent	Rol
3	Decorator	Rol
4	ConcreteDecorator	Rol
5	wrappedObj	Atributo
6	newBehavior()	Método
7	newState	Atributo

Tabla 4, Relación de los elementos de estructura del patrón *Decorator*.

Criterios de interrelación.

La Tabla 5 muestra la descripción de los criterios de interrelación del patrón *Decorator:*

Elemento de la Figura 7	DESCRIPCIÓN
a	El *ConcreteComponent* debe implementar o heredar del *Component* o, en su defecto, el *ConcreteComponent* y el *Component* pueden ser representados (colapsados) como una sola clase.
b	El *Decorator* debe implementar o heredar del *Component*.
c	El *ConcreteDecorator* debe implementar o heredar del *Decorator*.

Tabla 5, Criterios de interrelación del patrón *Decorator*.

Otros criterios.

La Tabla 6 muestra la descripción de otros criterios del patrón *Decorator:*

CRITERIO	DESCRIPCIÓN
DEC_1	El o los métodos del *ConcreteDecorator* que son heredados o implementan el *Decorator* deberán proveer nuevo comportamiento al *ConcreteComponent*, de forma que, al llamar estos métodos se pueda hacer uso de este nuevo comportamiento.
DEC_2	El *Decorator* debe envolver a un solo *Component*.

Tabla 6, Otros criterios del patrón *Decorator*.

3.3 Adapter.

3.3.1 Roles.

Client, Target, Adapter, Adaptee.

3.3.2 Descripción de la solución.

Convertir una interfaz en otra interfaz que el cliente espera, de manera que el cliente pueda interactuar con componentes incompatibles.

3.3.3 Criterios.

La **¡Error! No se encuentra el origen de la referencia.** muestra los criterios de estructura, de interrelación y otros criterios, cada uno de los elementos numerados deberán existir en la instancia candidata del patrón.

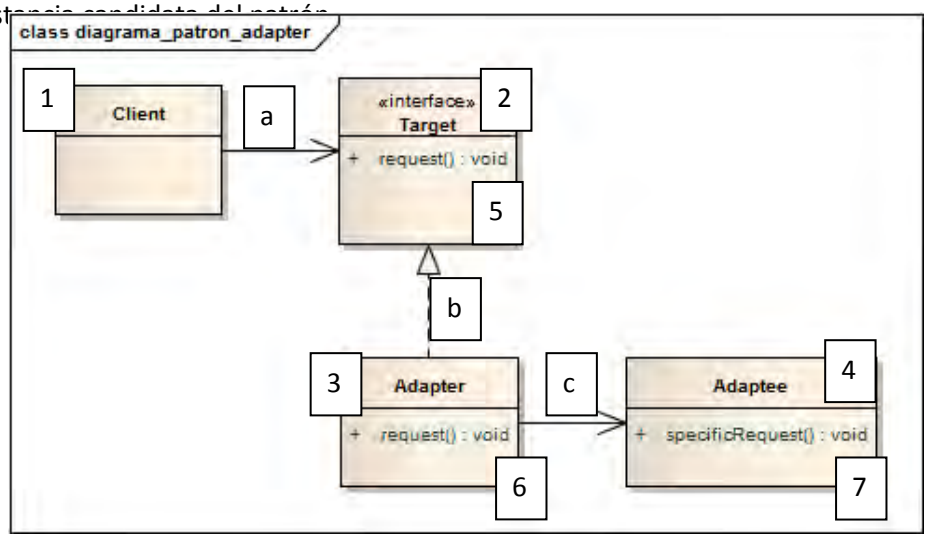

Figura 8, Diagrama general de criterios del patrón *Adapter*.

Criterios de estructura.

La Tabla 7 muestra la relación de los números de la **¡Error! No se encuentra el origen de la referencia.** con los roles o métodos que forman parte de los criterios de estructura. Todos los elementos mencionados deberán existir en cada una de las instancias de este patrón.

Elemento de la ¡Error! No se encuentra el origen de la referencia.	Nombre de elemento	Tipo de elemento

1	Client	Rol
2	Target	Rol
3	Adapter	Rol
4	Adaptee	Rol
5	request() (Target)	Método
6	request() (Adapter)	Método
7	specificRequest()	Método

Tabla 7, Relación de los elementos de estructura del patrón *Adapter*.

Criterios de interrelación.

La Tabla 8 muestra la descripción de los criterios de interrelación del patrón *Adapter:*

Elemento de la ¡Error! No se encuentra el origen de la referencia.	**DESCRIPCIÓN**
a	El *Client* debe hacer una llamada al *Adapter* llamando a un método en él a través de la interfaz *Target*.
b	El *Adapter* debe implementar la interfaz *Target* o, en su defecto, heredar de ésta.
c	El *Adapter* debe tener una instancia del *Adaptee* o, implementar o heredar del *Adaptee*.

Tabla 8, Criterios de interrelación del patrón *Adapter*.

Otros criterios.

La Tabla 9 muestra la descripción de otros criterios del patrón *Adapter:*

CRITERIO	**DESCRIPCIÓN**
ADP_1	Todas las llamadas del *Client* quedan delegadas al *Adaptee*.
ADP_2	El *Target* y el *Adaptee* deben estar conceptualmente relacionados, es decir, deben tener las mismas tareas pero cada uno debe de llevarlas a cabo de forma diferente.

Tabla 9, Otros criterios del patrón *Adapter*.

Capítulo 4

EVALUACIÓN Y ANÁLISIS DE LAS INSTANCIAS

En el capítulo anterior se describieron los criterios que serán utilizados para el análisis de cada una de las instancias de los patrones de diseño. En el presente capítulo se utilizarán dichos criterios para realizar el análisis y evaluación de cada una de las instancias de los patrones de diseño aplicando los criterios y, al mismo tiempo, comparándolos con el comportamiento de cada una de las instancias.

El proceso para el análisis será el siguiente:

- Primero se mostrará el diagrama de clases de la instancia a analizar.
- Se realizará la verificación de los criterios de estructura: Se mostrará una tabla donde se relacionará el número mostrado en la respectiva figura, el nombre del rol o método del patrón de diseño y el nombre del rol o método encontrado en la instancia.
- Se realizará la verificación de los criterios de interrelación: Se mostrará una tabla donde se relacionará la letra que representa la interrelación, la descripción de este criterio aplicado a la instancia y la verificación determinando con un "Sí" si fue verificada y con un "No" si no lo fue.
- Se realizará la verificación de otros criterios. Se mostrará una tabla donde se relaciona el criterio, la descripción de éste aplicado a la instancia y la verificación procediendo de la misma forma como se procedió en el punto anterior.
- Por último se describirá la conclusión a la que se llegó de acuerdo al análisis anterior.

A continuación se describirán algunos ejemplos del análisis de los criterios de estructura, de interrelación y otros criterios del patrón *Observer*:

Ejemplo del análisis de los criterios de estructura. En la Figura 6 el *Subject* tiene el número 1, lo que significa que la instancia a analizar debe tener alguna clase que juegue el rol de *Subject* y, por ejemplo, en el *ConcreteSubject* se tiene el método *notifyObservers()* (con el número 5) lo que significa que se deberá encontrar este método también en el *ConcreteSubject* en cada una de las instancias potenciales del patrón *Observer*.

Ejemplo del análisis de los criterios de interrelación. En la Figura 6 la implementación del *Subject* en el *ConcreteSubject* tiene la letra "a" lo que indica que es una relación que deberá cumplirse en las instancias, además la letra "c" representa la llamada que el *Subject* hace al *Observer* llamando al método *update()* en el método *notifyObservers()*.

Ejemplo del análisis de otros criterios. Un criterio que es importante analizar y que además, no entra en los criterios de estructura ni de interrelación, es el hecho de que el mensaje enviado por el *Subject* deberá enviarse a todos los *ConcreteObservers*, a diferencia de los criterios descritos anteriormente, el comportamiento de este criterio no se puede apreciar en el diagrama por lo que algunos criterios dentro de "Otros criterios" no se mostrarán en su respectivo diagrama general.

4.1 Observer.

4.1.1 Instancia OBS_*CompositeFigure* (JHotDraw).

Diagrama de clases.

La Figura 8 muestra el diagrama de clases de la instancia.

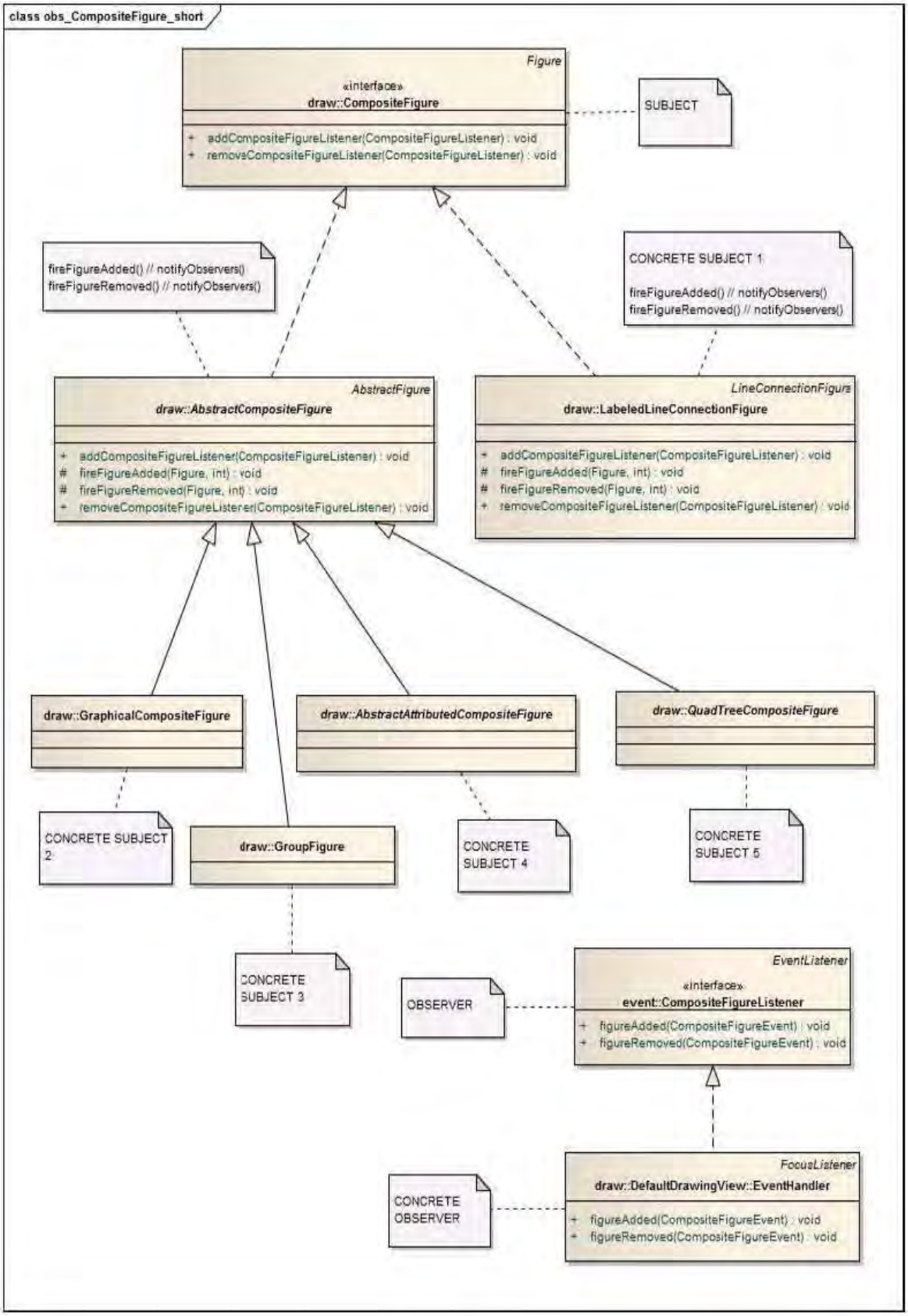

Figura 8, Diagrama de clases de la instancia *OBS_CompositeFigure*.

Verificación de los criterios de estructura.

La Tabla 10 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 1 y la Figura 6, identificando cada uno de los elementos de la Tabla 1 en la Figura 8.

Elemento de la Figura 6	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Subject	CompositeFigure
2	ConcreteSubject	LabeledLineConnectionFigure, AbstractCompositeFigure (Abstract class): GraphicalCompositeFigure, AbstractAtributedCompositeFigure, GroupFigure, QuadTreeCompositeFigure
3	Observer	CompositeFigureListener
4	ConcreteObserver	EventHandler
5	*notifyObservers()*	*fireFigureAdded(), fireFigureRemoved()*
6	*registerObserver()*	*addCompositeFigureListener(CompositeFigureListener listener)*
7	*removeObserver()*	*removeCompositeFigureListener(CompositeFigureListener listener)*
8	*update()*	*figureAdded(), figureRemoved()*

Tabla 10, Verificación de los criterios de estructura de la instancia *OBS_CompositeFigure*.

Verificación de los criterios de interrelación.

La Tabla 11 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 2 y la Figura 6, identificando cada uno de los elementos en la Figura 8.

Elemento de la Figura 6	Descripción	Verificado
a	El *AbstractCompositeFigure* y el *LabeledLineConnectionFigure* implementan el *CompositeFigure*.	*Sí*
b	El *EventHandler* implementa el *CompositeFigureListener*.	*Sí*
c	Los métodos *figureAdded()* y *figureRemoved()* son llamados en el método *fireFigureAdded()* y el *fireFigureRemoved()* respectivamente, como se muestra en las siguientes líneas de código encontradas en los métodos *fireFigureAdded()* y *fireFigureRemoved()*: *((CompositeFigureListener)listeners[i+1]).figureAdded(event)* *((CompositeFigureListener)listeners[i+1]).figureRemoved(event)*	*Sí*

Tabla 11, Verificación de los criterios de interrelación de la instancia *OBS_CompositeFigure*.

Otros criterios.

La Tabla 12 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 3 y la Figura 6, identificando cada uno de los elementos de la tabla en la Figura 8 y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
OBS_1	Los mensajes enviados por *fireFigureAdded()* y *fireFigureRemoved()* son siempre enviados a todos los *ConcreteObservers*, a continuación se muestran las líneas de código de estos métodos donde se puede notar que todos los "*listeners"* son notificados: *for (int i = listeners.length-2; i>=0; i-=2){* *…* *((CompositeFigureListener) listeners[i + 1]).figureRemoved(event);* *…}*	*Sí*

Tabla 12, Verificación de otros criterios de la instancia *OBS_CompositeFigure*.

Conclusiones.

Como se puede notar en el criterio OBS_1, el *for* que es utilizado para notificar a todos los *listeners* utiliza el incremento "*i-=2*", esto se explicará a detalle a continuación:

En el siguiente método se muestra como cada *listener* es agregado a la lista de *listeners* utilizando el método *add* de un objeto (*listenerList*) de la clase *EventListenerList* que es parte del paquete *Javax.Swing.Event:*

```
AbstractCompositeFigure
public void addCompositeFigureListener(CompositeFigureListener listener) {
        listenerList.add(CompositeFigureListener.class, listener);
    }
```

A continuación se muestra el cuerpo del método *add* de la clase *EventListenerList* en donde esencialmente se puede observar que el método *add* guarda en el primer elemento que encuentra disponible el nombre de la clase de la que se trata el *listener* y en el siguiente elemento el *listener* mismo:

```
/**
     * Adds the listener as a listener of the specified type.
     * @param t the type of the listener to be added
     * @param l the listener to be added
     */
```

```
 public synchronized <T extends EventListener> void add(Class<T> t, T l) {
if (l==null) {
    // In an ideal world, we would do an assertion here
    // to help developers know they are probably doing
    // something wrong
    return;
}
if (!t.isInstance(l)) {
    throw new IllegalArgumentException("Listener " + l +
                    " is not of type " + t);
}
if (listenerList == NULL_ARRAY) {
    // if this is the first listener added,
    // initialize the lists
    listenerList = new Object[] { t, l };
} else {
    // Otherwise copy the array and add the new listener
    int i = listenerList.length;
    Object[] tmp = new Object[i+2];
    System.arraycopy(listenerList, 0, tmp, 0, i);

    tmp[i] = t;
    tmp[i+1] = l;

    listenerList = tmp;
}
 }
```

Por lo anterior se puede notar que los *listener* se guardan cada i+1 elemento, donde *listeners.length-2 > i >= 0*, es por esta razón que el barrido del arreglo se realiza con el incremento "*i-=2*", concluyendo así que la notificación si se realiza a todos los *listeners*.

Por otra parte se puede notar que esta instancia cumple con todos los criterios y la estructura es similar al diagrama general del patrón *Observer* (Figura 6).

Se puede notar también que el nombre de los métodos dan una buena idea acerca de su propósito, por ejemplo, los métodos *addCompositeFigureListener()* y *removeCompositeFigureListener()* dan una idea clara y concisa: *"agregar y remover un escuchador (o listener, como los autores llaman a los observers) de un CompositeFigure"*. Además se tiene que la palabra "*fire*" da la idea de "disparar" algo. Para el caso de este proyecto, los métodos que

tienen un nombre con esta palabra disparan una notificación cuando algo sucede. En este caso cuando un evento ocurre, de esta manera se puede tener una idea clara del comportamiento del método.

En este caso existe una cierta variación respecto al diagrama de clases del patrón Observer, tenemos una clase abstracta AbstractCompositeFigure la cual contiene los métodos notifyObservers, recordemos que el objetivo de una clase abstracta es encapsular características que tengan más de una clase, por lo que esta generalización no afecta a la estructura de la instancia ni el comportamiento del patrón.

Debido a lo mencionado anteriormente se puede decir que, dar nombres que describen el comportamiento del método se puede tomar como una buena práctica en el desarrollo de software, haciendo así, más fácil la comprensión del código y para este caso el análisis de las instancias de los patrones.

Finalmente se puede concluir que esta instancia es una instancia válida del patrón *Observer* de acuerdo a los criterios estáticos establecidos.

4.1.2 Instancia OBS_DrawingView (JHotDraw).

Diagrama de clases.

La Figura 9 muestra el diagrama de clases de la instancia:

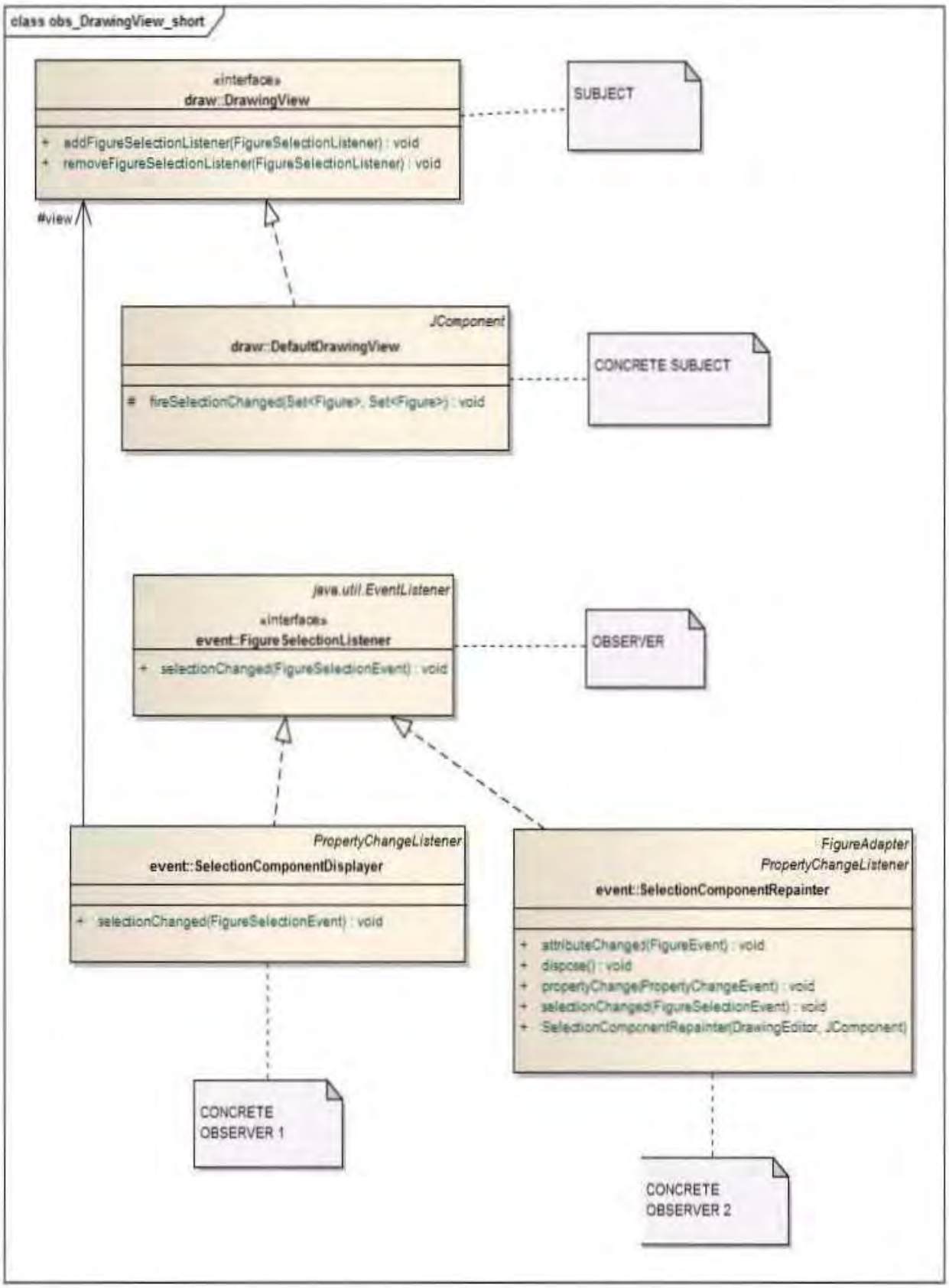

Figura 9, Diagrama de clases de la instancia *OBS_DrawingView*.

Verificación de los criterios de estructura.

La Tabla 13 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 1 y la Figura 6, identificando cada uno de los elementos de la Tabla 1 en la Figura 9.

Elemento de la Figura 6	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Subject	DrawingView
2	ConcreteSubject	DefaultDrawingView
3	Observer	FigureSelectionListener
4	ConcreteObserver	SelectionComponentDisplayer, SelectionComponentRepainter
5	*notifyObservers()*	*fireSelectionChanged()*
6	*registerObserver()*	*addFigureSelectionListener()*
7	*removeObserver()*	*removeFigureSelectionListener()*
8	*update()*	*selectionChanged()*

Tabla 13, Verificación de los criterios de estructura de la instancia OBS_DrawingView.

Verificación de los criterios de interrelación.

La Tabla 14 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 2 y la Figura 6, identificando cada uno de los elementos en la Figura 9.

Elemento de la Figura 6	Descripción	Verificado
a	El *DefaultDrawingView* implementa la interfaz *DrawingView*.	*Sí*
b	El *SelectionComponentDisplayer* y el *SelectionComponentRepainter* implementan el *FigureSelectionListener*.	*Sí*
c	En el *DefaultDrawingView* en el método *fireSelectionChanged()* está la siguiente línea: *((FigureSelectionListener) listeners[i + 1]).selectionChanged(event);* llamando al método *update()*.	*Sí*

Tabla 14, Verificación de otros criterios de la instancia OBS_DrawingView.

Otros criterios.

La Tabla 15 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 3 y la Figura 6, identificando cada uno de los elementos de la tabla en la Figura 9 y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
OBS_1	Los mensajes enviados por *fireSelectionChanged()* son siempre enviados a todos los *ConcreteObservers*, a continuación se muestran las líneas de código de este método donde se puede notar que todos los *listeners* son notificados: *for (int i = listeners.length-2; i>=0; i-=2){* *…* *((FigureSelectionListener) listeners[i + 1]).selectionChanged(event);* *…}*	*Sí*

Tabla 15, Verificación de otros criterios de la instancia *OBS_DrawingView*.

Conclusiones.

Como se puede observar, esta instancia es muy similar a la anterior ya que posee la misma estructura e incluso los nombres de los atributos y métodos son similares también.

Además, en el criterio OBS_1 en el "*for*" también se tiene la misma situación, sólo que en este caso el método que es llamado es el "*selectionChanged()*" que es el método *update()* en este caso.

Prácticamente se pueden adoptar las conclusiones de la instancia anterior a ésta, por lo que se puede concluir que ésta también se puede considerar como una instancia válida del patrón *Observer* de acuerdo a los criterios establecidos.

4.1.3 Instancia OBS_Figure (JHotDraw).

Diagrama de clases.

La Figura 10 muestra el diagrama de clases de la instancia.

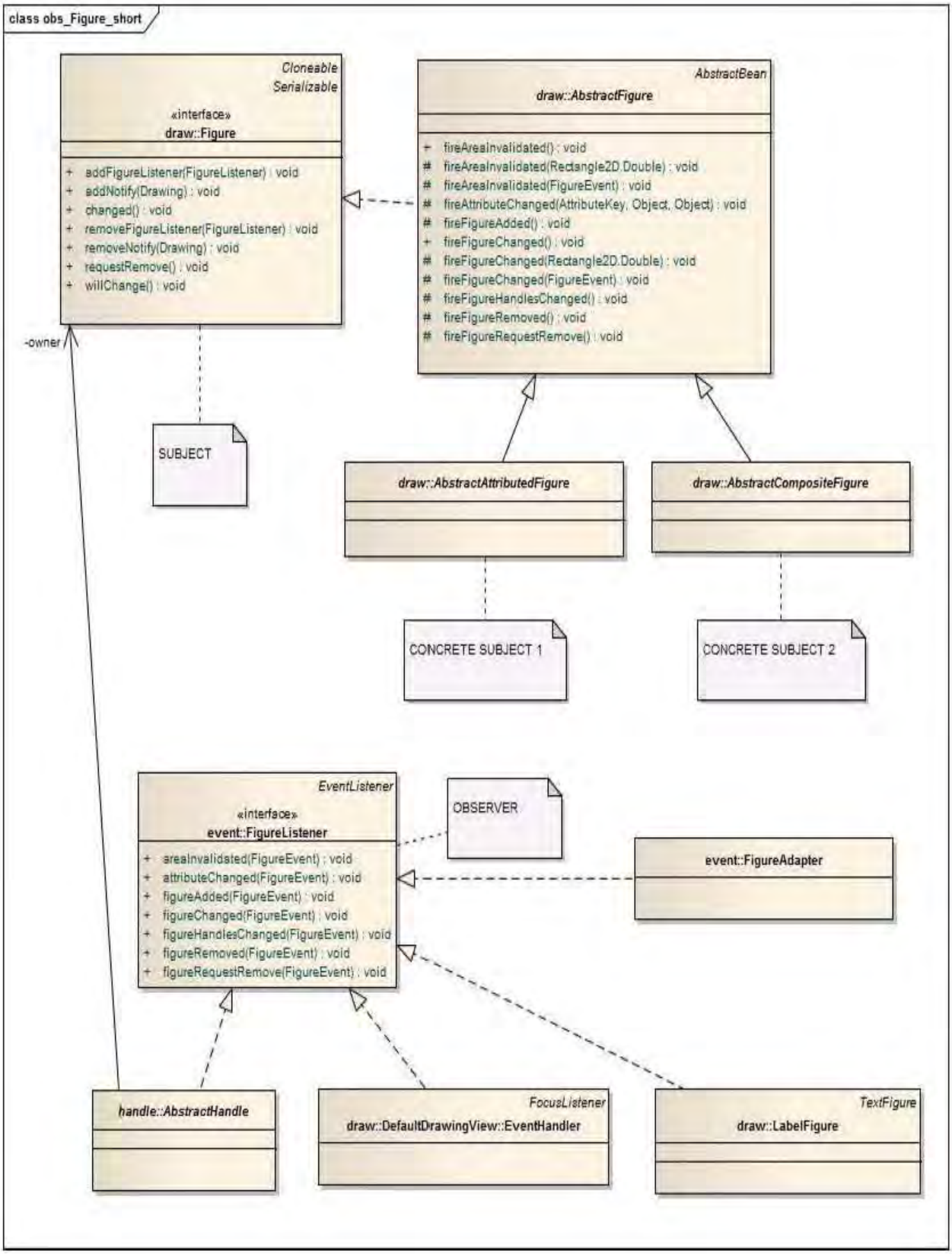

Figura 10, Diagrama de clases de la instancia *OBS_Figure*.

Verificación de los criterios de estructura.

La Tabla 16 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 1 y la Figura 6, identificando cada uno de los elementos de la Tabla 1 en la Figura 10.

Elemento de la Figura 6	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Subject	Figure
2	ConcreteSubject	*AbstractFigure* (Abstract class) AbstractAttributedFigure, AbstractCompositeFigure
3	Observer	FigureListener
4	ConcreteObserver	FigureAdapter, AbstractHandle, EventHandle, LabelFigure
5	*notifyObservers()*	*fireAreaInvalidated(), fireAttributeChanged(), fireFigureAdded(), fireFigureChanged(), fireFigureHandlesChanged(), fireFigureRemoved(), fireFigureRequestRemove()*
6	*registerObserver()*	*addFigureListener()*
7	*removeObserver()*	*removeFigureListener()*
8	*update()*	*areaInvalidated(), attributedChanged(), figureAdded(), figureChanged(), figureHandlesChanged(), figureRemoved(), figureRequestRemove()*

Tabla 16, Verificación de los criterios de estructura de la instancia *OBS_Figure.*

Verificación de los criterios de interrelación.

La Tabla 17 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 2 y la Figura 6, identificando cada uno de los elementos en la Figura 10.

Elemento de la Figura 6	Descripción	Verificado
a	El *AbstractFigure* implementa la interfaz *Figure*.	*Sí*
b	*FigureAdapter*, *AbstractHandle*, *EventHandle* y *LabelFigure* implementan el *FigureListener*.	*Sí*
c	En el *AbstractFigure* en los métodos "*notifyObservers()*" está la siguiente línea: *((FigureListener) listeners[i + 1]).[nombre_metodo_update];* llamando al método *update()*.	*Sí*

Tabla 17, Verificación de los criterios de interrelación de la instancia *OBS_Figure.*

Otros criterios.

La Tabla 18 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 3 y la Figura 6, identificando cada uno de los elementos de la tabla en la Figura 8 y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
OBS_1	Los mensajes enviados por *fireSelectionChanged()* son siempre enviados a todos los *ConcreteObservers*, a continuación se muestran las líneas de código de este método donde se puede notar que todos los *listeners* son notificados: *for (int i = listeners.length-2; i>=0; i-=2){* *…* *((FigureSelectionListener) listeners[i + 1]).selectionChanged(event);* *…}*	*Sí*

Tabla 18, Verificación de otros criterios de la instancia *OBS_Figure.*

Conclusiones.

Como se puede notar esta instancia es similar a las instancias anteriores, ya que las tres poseen la misma estructura y la misma esencia, se puede decir que las tres instancias fueron pensadas y desarrolladas de la misma forma, el *"for"* cumple también con las mismas características descritas anteriormente.

Por lo anterior se puede concluir que ésta también es una instancia válida del patrón *Observer* de acuerdo a los criterios establecidos.

4.1.4 Instancia OBS_Handle (JHotDraw).

Diagrama de clases.

La Figura 11 muestra el diagrama de clases de la instancia.

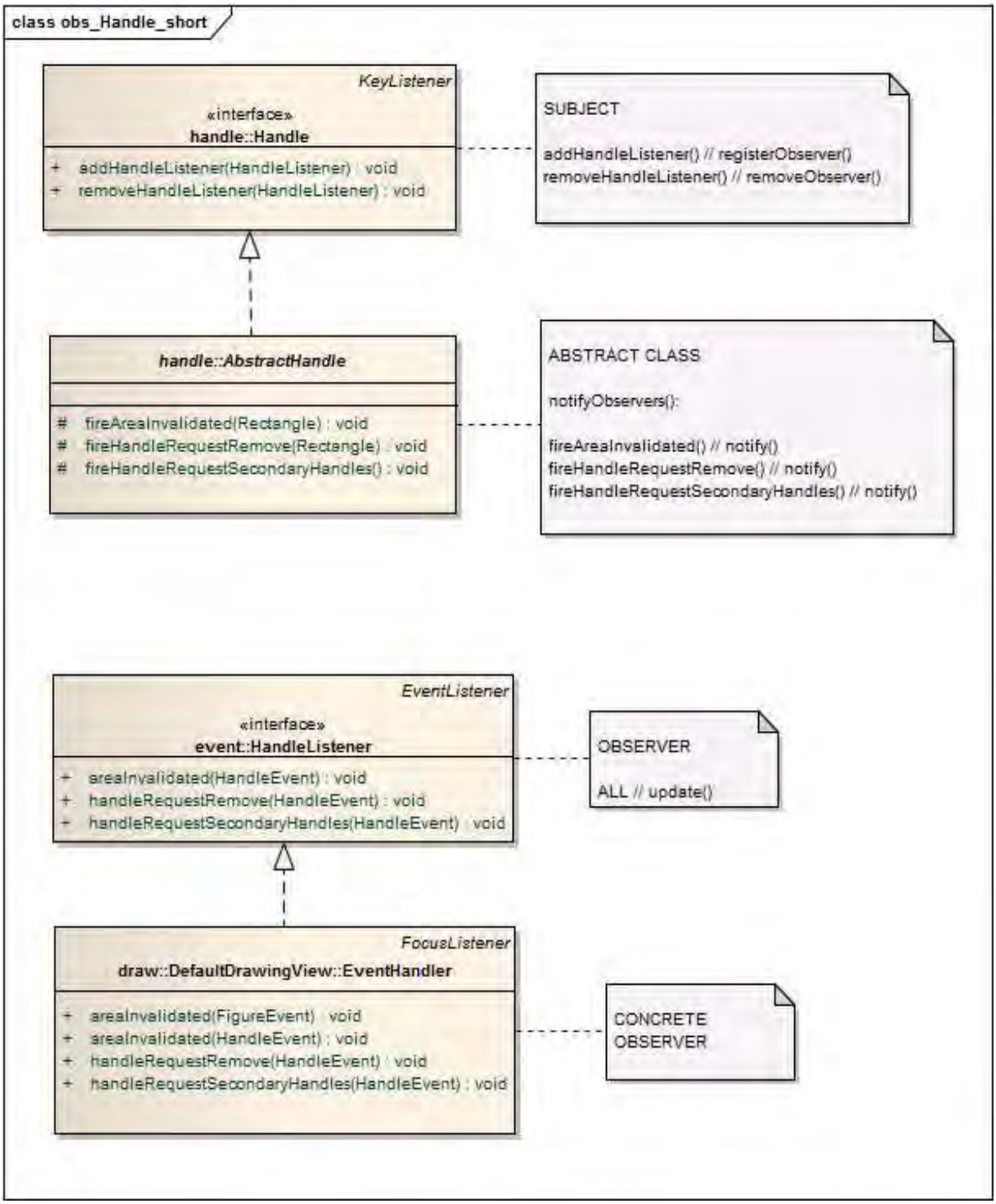

Figura 11, Diagrama de clases de la instancia *OBS_Handle.*

Nota: Los *ConcreteSubject* no se muestran en el diagrama debido a que son demasiados, sin embargo se mencionan en la "verificación de los criterios de estructura".

Verificación de los criterios de estructura.

La Tabla 19 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 1 y la Figura 6, identificando cada uno de los elementos de la Tabla 1 en la Figura 11.

Elemento de la Figura 6	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Subject	Handle
2	ConcreteSubject	*AbstractHandle* (Abstract class) BezierNodeHandle, SVGRectRadiusHandle, SVGPathOutlineHandle, ODGPathOutlineHandle, LocatorHandle, LinkHandle, ConvexHullOutlineHandle, DragHandle, OrientationHandle, RoundRectangleRadiusHandle, BezierControlPointHandle.
3	Observer	HandleListener
4	ConcreteObserver	EventHandler
5	*notifyObservers()*	*fireAreaInvalidated(), fireHandleRequestRemove(), fireHandleRequestSecondaryHandles().*
6	*registerObserver()*	*addHandleListener()*
7	*removeObserver()*	*removeHandleListener()*
8	*update()*	*areaInvalidated(), handleRequestRemove(), handleRequestSecondaryHandles().*

Tabla 19, Verificación de los criterios de estructura de la instancia *OBS_Handle.*

Verificación de los criterios de interrelación.

La Tabla 20 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 2 y la Figura 6, identificando cada uno de los elementos en la Figura 11.

Elemento de la Figura 6	Descripción	Verificado
a	El *AbstractHandle* implementa la interfaz *Handle*.	*Sí*
b	El *HandleListener* implementa el *HandleListener*.	*Sí*
c	En el *AbstractHandle* en los métodos "*notifyObservers()*" está la siguiente línea: *((HandleListener) listeners[i + 1]).[nombre_metodo_update];* llamando al método *update()*.	*Sí*

Tabla 20, Verificación de los criterios de interrelación de la instancia *OBS_Handle.*

Otros criterios.

La Tabla 21 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 3 y la Figura 6, identificando cada uno de los elementos de la tabla en la Figura 11 y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
OBS_1	Los mensajes enviados por los métodos "*notifyObservers()"* son siempre enviados a todos los ConcreteObservers, a continuación se muestran las líneas de código de estos métodos donde se puede observar que todos los *listeners* son notificados: *for (int i = listeners.length-2; i>=0; i-=2){* *…* *((HandleListener) listeners[i + 1]) .[nombre_metodo_update];* *…}*	*Sí*

Tabla 21, Verificación de otros criterios de la instancia *OBS_Handle.*

Conclusiones.

La interfaz *Handle* fue diseñada para modificar un objeto *Figure* de modo que esta interfaz pueda manejar los eventos ocurridos en un objeto *Figure*.

En este caso se tiene que el *Subject* posee varias notificaciones, es decir, maneja diversos eventos: si existe un área inválida, cuando se solicita remover algún manejo de evento (*handle)* y cuando se solicita agregar un manejo secundario de un evento (*secondary handle*) y, por lo tanto, se tienen también tres métodos *update()*: *areaInvalidated(), handleRequestRemove()* y *handleRequestSecondaryHandles()*.

Por ejemplo el método *fireAreaInvalidated()* fue creado para notificar a los *listeners* interesados de un evento en el que una figura (es decir, un objeto Figure) se quiere dibujar en un área en la que no es válida y por ello un "*drawing view*" (es decir, un objeto *DrawingView*) requiere ser dibujado nuevamente, este evento es el parámetro de esta función y, a su vez, en ella se llama al método *areaInvalidated()* el cual es unos de los métodos *update()*, los cuales realizan diferentes acciones dependiendo de sus respectivos objetivos de acuerdo a un evento en particular.

Como se puede notar de acuerdo a lo descrito anteriormente, a la similitud que existe con las otras instancias y a que esta instancia cumple con todos los criterios; se puede decir que ésta es una instancia válida del patrón *Observer* de acuerdo a los criterios establecidos.

4.1.5 Instancia OBS_ListModel (Swing).

Diagrama de clases.

La Figura 13 muestra el diagrama de clases de la instancia.

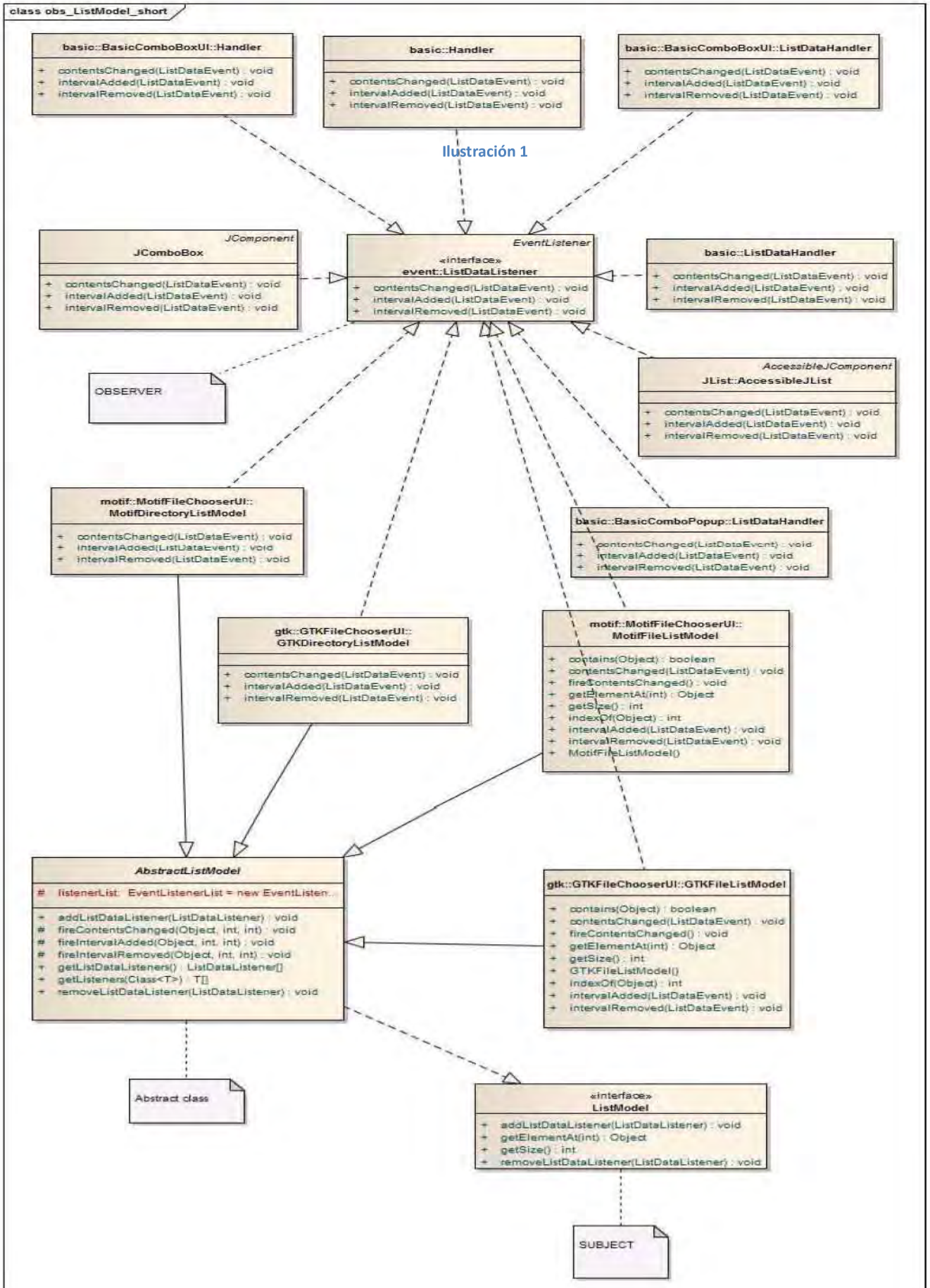

Figura 12, Diagrama de clases de la instancia *OBS_ListModel.*

Nota: Los *ConcreteSubject* únicamente se mencionan en la "Verificación de los criterios de estructura", no se encuentran en el diagrama de clases.

Verificación de los criterios de estructura.

La Tabla 22 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 1 y la Figura 6, identificando cada uno de los elementos de la Tabla 1 en la Figura 13.

Elemento de la Figura 6	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Subject	ListModel
2	ConcreteSubject	gtk::GTKFileChooserUI::DirectoryComboBoxModel, gtk::GTKFileChooserUI::FilterComboBoxModel, **gtk::GTKFileChooserUI::GTKDirectoryListModel, gtk::GTKFileChooserUI::GTKFileListModel, motif::MotifFileChooserUI::MotifFileListModel, motif::MotifFileChooserUI::MotifDirectoryListModel**, motif::MotifFileChooserUI::FilterComboBoxModel, windows::WindowsFileChooserUI::DirectoryComboBoxModel, windows::WindowsFileChooserUI::FilterComboBoxModel, metal::MetalFileChooserUI::FilterComboBoxModel, metal::MetalFileChooserUI::DirectoryComboBoxModel, basic::BasicDirectoryModel, DefaultListModel, DefaultComboBoxModel.
3	Observer	ListDataListener
4	ConcreteObserver	basic::BasicComboBoxUI::Handler, basic::BasicComboBoxUI::ListDataHandler, basic::BasicComboPopup::ListDataHandler, basic::ListDataHandler, basic::Handler, **gtk::GTKFileChooserUI::GTKDirectoryListModel gtk::GTKFileChooserUI::GTKFileListModel, motif::MotifFileChooserUI::MotifFileListModel, motif::MotifFileChooserUI::MotifDirectoryListModel,** JComboBox, JList::AccessibleJList.
5	*notifyObservers()*	*fireContentsChanged(Object, int, int), fireIntervalAdded(Object, int, int), fireIntervalRemoved(Object, int, int).*
6	*registerObserver()*	*addListDataListener(ListDataListener).*
7	*removeObserver()*	*removedListDataListener(ListDataListener).*
8	*update()*	*contentsChanged(ListDataEvent), intervalAdded(ListDataEvent), intervalRemoved(ListDataEvent).*

Tabla 22, Verificación de los criterios de estructura de la instancia *OBS_ListModel.*

Verificación de los criterios de interrelación

La Tabla 23 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 2 y la Figura 6, identificando cada uno de los elementos en la Figura 13.

Elemento de la Figura 6	Descripción	Verificado
a	Todas las clases que son ConcreteSubject implementan la interfaz ListModel.	*Sí*
b	Todas las clases que son ConcreteObserver implementan la interfaz ListDataListener.	*Sí*
c	En la clase AbstractListModel en los métodos *fireContentsChanged(Object, int, int), fireIntervalAdded(Object, int, int) y fireIntervalRemoved(Object, int, int)* se encuentra la siguiente línea: *((ListDataListener) listeners[i + 1]).[nombre_metodo_update];* llamando al método *update()*.	*Sí*

Tabla 23, Verificación de los criterios de interrelación de la instancia *OBS_ListModel.*

Otros criterios.

La Tabla 24 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 3 y la Figura 6, identificando cada uno de los elementos de la tabla en la Figura 13 y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
OBS_1	Los mensajes enviados por los métodos "*notifyObservers()"* son siempre enviados a todos los ConcreteObservers, a continuación se muestran las líneas de código de estos métodos donde se puede observar que todos los *listeners* son notificados: *for (int i = listeners.length-2; i>=0; i-=2){* *…* *((ListDataListener) listeners[i + 1]) .[nombre_metodo_update];* *…}*	*Sí*

Tabla 24, Verificación de otros criterios de la instancia *OBS_ListModel.*

Conclusiones.

La clase *ListModel* es la clase encargada del manejo de las listas que son usadas en la biblioteca Swing. Recuérdese que una lista es una colección de objetos, por lo que esta clase está destinada

al manejo de estos objetos, es decir, a la agregación, eliminación y modificación de cada uno de los objetos de las listas, también disparará notificaciones a los *Observers* involucrados en esta instancia.

Por otro lado, la clase *ListDataListener* fue creada como *"Listener"* para la clase *ListModel*, esta clase estará a la escucha de cualquier modificación que se realice en la clase *ListModel*, de forma que se actualizarán a todos los *Observers* en caso de que ocurra un cambio en algún objeto *ListModel*.

Como se puede notar en el diagrama de clases, la estructura de esta instancia es muy similar a las anteriores, incluso es similar a las instancias del proyecto *JHotDraw*. Sin embargo, en esta instancia se tienen cuatro clases que son *ConcreteObservers* y *ConcreteSubject* al mismo tiempo, estas clases son las siguientes:

gtk::GTKFileChooserUI::GTKDirectoryListModel,

gtk::GTKFileChooserUI::GTKFileListModel,

motif::MotifFileChooserUI::MotifFileListModel,

motif::MotifFileChooserUI::MotifDirectoryListModel.

Como se puede notar en el diagrama de clases, estas clases tienen los métodos *notifyObservers()* y los métodos *update()* al mismo tiempo. A simple vista esto puede parecer excesivo ya que la misma clase que está notificando a los *Observers* de que ocurrió algún cambio en la clase, es la que recibe la notificación de que debe realizar alguna acción. Sin embargo, analizando esta situación detenidamente se puede notar que la notificación del evento se realiza a todos los *Observers* y no únicamente a la clase en la que se realizó algún cambio. Por esta razón es factible que una clase pueda ser *Subject* y *Observer* al mismo tiempo.

Finalmente, de acuerdo a la estructura de esta instancia, al análisis descrito anteriormente y a las conclusiones anteriores se puede decir que esta es una instancia válida del patrón *Observer* de acuerdo a los criterios establecidos.

4.1.6 Instancia OBS_JList (Swing).

Diagrama de clases.

La Figura 13 muestra el diagrama de clases de la instancia.

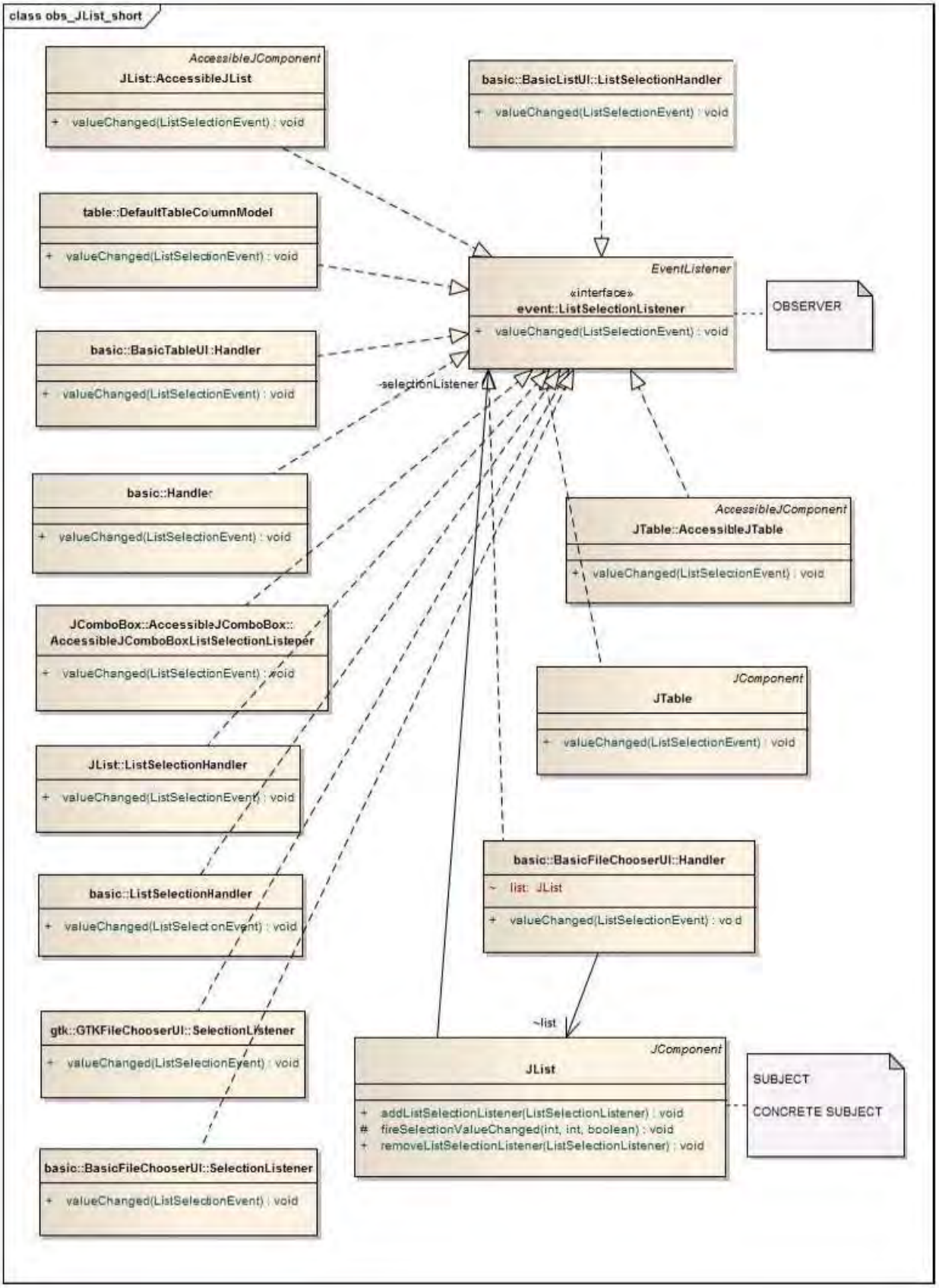

Figura 13, Diagrama de clases de la instancia *OBS_JList.*

Verificación de los criterios de estructura.

La Tabla 25 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 1 y la Figura 6, identificando cada uno de los elementos de la Tabla 1 en la Figura 13.

Elemento de la Figura 6	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Subject	JList
2	ConcreteSubject	JList
3	Observer	ListSelectionListener
4	ConcreteObserver	JList::AccessibleList, JList::ListSelectionHandler, table::DefaultTableColumnModel, basic::BasicTableUI::Handler, basic::Handler, basic::BasicFileChooserUI::Handler, basic::BasicFileChooserUI::SelectionListener, basic::BasicListUI::ListSelectionHandler, basic::ListSelectionHandler, JComboBox::AccessibleJComboBoxListSelectionListener, gtk::GTKFileChooserUI::SelectionListener, JTable, JTable::AccessibleTable.
5	*notifyObservers()*	*fireSelectionValueChanged(int, int, boolean)*
6	*registerObserver()*	*addListSelectionListener(ListSelectionListener)*
7	*removeObserver()*	*removeListSelectionListener(ListSelectionListener)*
8	*update()*	*valueChanged(ListSelectionEvent)*

Tabla 25, Verificación de los criterios de estructura de la instancia *OBS_JList.*

Verificación de los criterios de interrelación.

La Tabla 26 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 2 y la Figura 6, identificando cada uno de los elementos en la Figura 13.

Elemento de la Figura 6	Descripción	Verificado
a	*JList* es el *Subject* y el *ConcreteSubject* al mismo tiempo.	*Sí*
b	Todos los *ConcreteObservers* mencionados anteriormente implementan la interfaz *ListSelectionListener*.	*Sí*
c	En *JList* en el método "*notifyObservers()*" está la siguiente línea: *((ListSelectionListener) listeners[i + 1]).valueChanged(e);* llamando al método *valueChanged()*.	*Sí*

Tabla 26, Verificación de los criterios de interrelación de la instancia *OBS_JList.*

Otros criterios.

La Tabla 27 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 3 y la Figura 6, identificando cada uno de los elementos de la tabla en la Figura 13 y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
OBS_1	El mensaje enviado por el método "*notifyObservers()*" es siempre enviado a todos los *ConcreteObservers*, a continuación se muestran las líneas de código de este método donde se puede notar que todos los *listeners* son notificados: *for (int i = listeners.length-2; i>=0; i-=2){* *…* *((ListSelectionListener) listeners[i + 1]).valueChanged(e);* *…}*	*Sí*

Tabla 27, Verificación de otros criterios de la instancia *OBS_JList.*

Conclusiones.

La clase *JList*, que en esta instancia es el *Subject*, es un componente de la biblioteca *Swing* que muestra una lista de objetos y permite al usuario seleccionar uno o más objetos de la lista, además de notificar a los *Observers* de algún cambio en los objetos de esta clase. Por otro lado, la clase *ListSelectionListener*, que en esta instancia es el *Observer*, es la clase encargada de esperar algún evento, que en este caso es algún cambio en el valor de la selección de una lista *JList*, para actualizar el estado de los objetos de esta clase.

Además, en esta instancia también se cumplieron todos los criterios, por lo que la estructura es similar también a las instancias anteriores. De acuerdo a lo anterior se puede concluir que ésta es una instancia válida del patrón *Observer* de acuerdo a los criterios establecidos.

4.1.7 Instancia OBS_VetoableChange (Swing).

Diagrama de clases.

La Figura 14 muestra el diagrama de clases de la instancia.

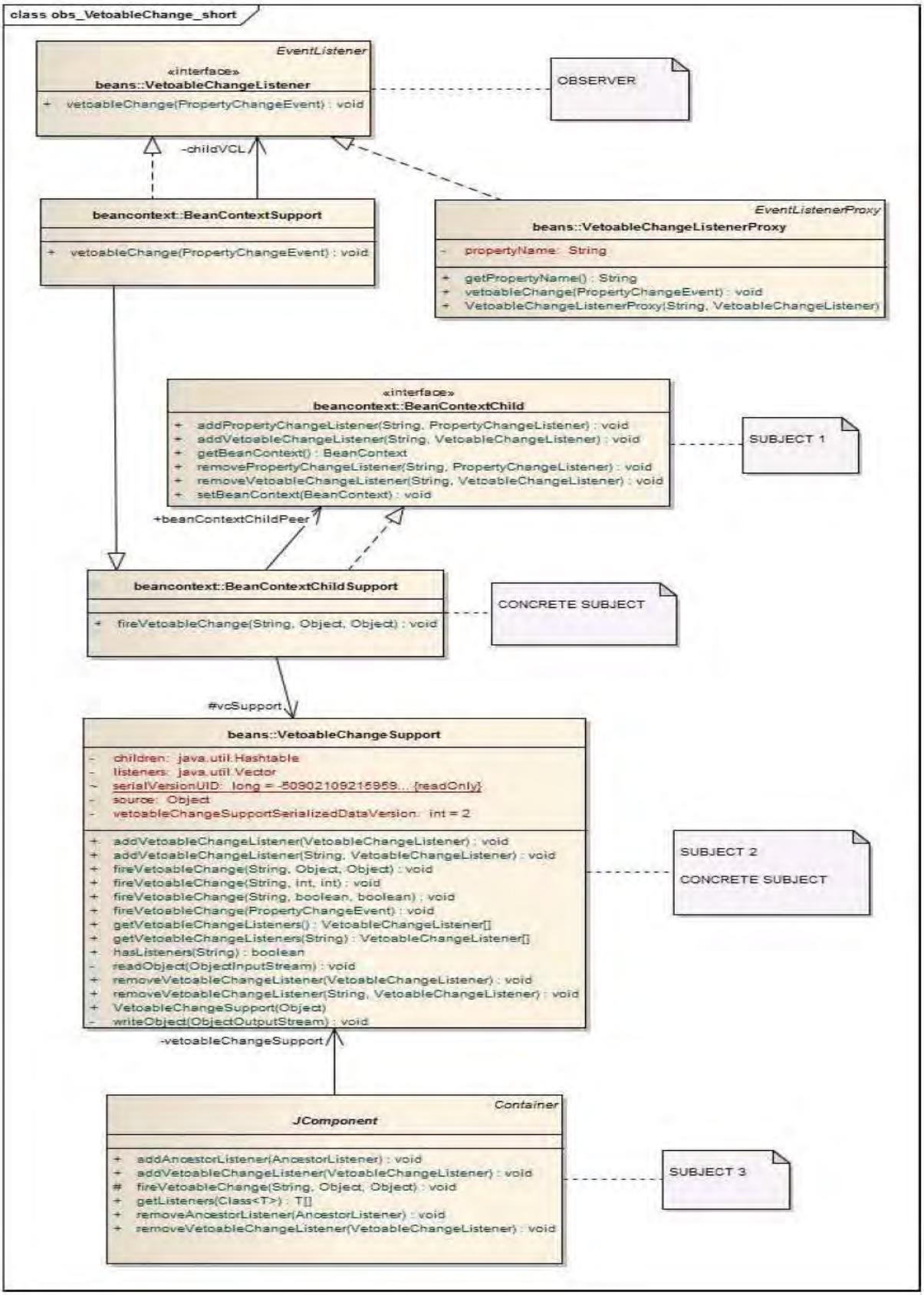

Figura 14, Diagrama de clases de la instancia *OBS_VetoableChange*.

Verificación de los criterios de estructura.

La Tabla 28 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 1 y la Figura 6, identificando cada uno de los elementos de la Tabla 1 en la Figura 14.

Elemento de la Figura 6	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Subject	**1** beancontext::BeanContextChild **2** beans::VetoableChangeSupport **3** JComponent
2	ConcreteSubject	**1** beancontext::BeanContextChildSupport **2** beans::VetoableChangeSupport **3** JComponent
3	Observer	VetoableChangeListener
4	ConcreteObserver	beancontext::BeanContextSupport, beans::VetoableChangeListenerProxy
5	*notifyObservers()*	*fireVetoableChange(String, Object, Object)*
6	*registerObserver()*	*addVetoableChangeListener(String, VetoableChangeListener)*
7	*removeObserver()*	*removeVetoableChangeListener(String, VetoableChangeListener)*
8	*update()*	*vetoableChange(PropertyChangeEvent)*

Tabla 28, Verificación de los criterios de estructura de la instancia *OBS_VetoableChange*

Verificación de los criterios de interrelación.

La Tabla 29 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 2 y la Figura 6, identificando cada uno de los elementos en la Figura 14.

Elemento de la Figura 6	Descripción	Verificado
a	*BeanContextChildSupport* implementa la interfaz *BeanContextChild*.	*Sí*
b	*beancontext::BeanContextSupport* y *beans::VetoableChangeListenerProxy* implementan la interfaz *VetoableChangeListener*.	*Sí*
c	En el *ConcreteSubject* ***beancontext::BeanContextChildSupport*** , en el método "*notifyObservers()*" existe una llamada al método *vetoableChange(PropertyChangeEvent)*.	*Sí*
	En ***beans::VetoableChangeSupport***, en el método "*notifyObservers()*" existe una llamada al método *vetoableChange(PropertyChangeEvent)*.	*Sí*
	En ***JComponent***, en el método "*notifyObservers()*" existe una llamada al método *vetoableChange(PropertyChangeEvent)*.	*Sí*

Tabla 29, Verificación de los criterios de interrelación de la instancia *OBS_VetoableChange.*

Otros criterios.

La Tabla 30 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 3 y la Figura 6, identificando cada uno de los elementos de la tabla en la Figura 13 y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
OBS_1	Los mensajes enviados por el método "*notifyObservers()"* en la clase ***beancontext::BeanContextChildSupport*** son siempre enviados a todos los *ConcreteObservers*. La estructura del método *fireVetoableChange()* es la siguiente: *protected VetoableChangeSupport vcSupport;* *public void fireVetoableChange(String name, Object oldValue, Object newValue) throws PropertyVetoException {* *vcSupport.fireVetoableChange(name, oldValue, newValue);* *}*	*Sí*
	Los mensajes enviados por el método "*notifyObservers()"* en la clase ***beans::VetoableChangeSupport*** son siempre enviados a todos los *ConcreteObservers*, a continuación se muestran las líneas de código de estos métodos donde se puede observar que todos los *listeners* son notificados: *java.util.Vector targets = null;* *targets = (java.util.Vector) listeners.clone();* *for (int i = 0; i < targets.size(); i++) {* *VetoableChangeListener target =* *(VetoableChangeListener)targets.elementAt(i);* *target.vetoableChange(evt);* *}*	*Sí*

	Los mensajes enviados por el método "*notifyObservers()"* en la clase ***JComponent*** son siempre enviados a todos los *ConcreteObservers*. La estructura del método *fireVetoableChange()* es la siguiente: *protected void fireVetoableChange(String propertyName, Object oldValue, Object newValue)* *throws java.beans.PropertyVetoException* *{* *if (vetoableChangeSupport == null) {* *return;* *}* *vetoableChangeSupport.fireVetoableChange(propertyName, oldValue, newValue);* *}*	*Sí*

Tabla 30, Verificación de otros criterios de la instancia *OBS_VetaobleChange.*

Conclusiones.

La interfaz *VetoableChangeListener* fue diseñada para crear *Listeners* para los objetos que estén interesados en dar soporte a propiedades restringidas, en esta instancia se tiene que existen más de un *Subject* (3 *Subject*), a continuación se analizará cada uno de ellos:

1 Subject: *beancontext::BeanContextChild*,

ConcreteSubject: *beancontext::BeanContextChildSupport*.

De acuerdo al diagrama de clases descrito anteriormente, se puede notar que este *Subject* tiene los métodos: *notifyObservers(), revomeObserver() y registerObserver(),* sin embargo, este *Subject* no tiene una llamada directa al método *update()* (*vetoableChange(PropertyChangeEvent)*), a continuación se muestra la estructura del método *fireVetoableChange()*:

```
protected VetoableChangeSupport vcSupport;
public void fireVetoableChange(String name, Object oldValue, Object
newValue) throws PropertyVetoException {
      vcSupport.fireVetoableChange(name, oldValue, newValue);
    }
```

Como se puede notar existe un objeto de la clase *VetoableChangeSupport*, el cual es utilizado para llamar el método *update()*. Los autores de esta clase, y en particular de este método, reutilizaron la clase destinada a dar soporte a los cambios ocurridos construyendo un objeto de ésta y utilizándola para llamar al método *update()* cuando se requiera.

2 Subject: *beans::VetoableChangeSupport*,

ConcreteSubject: *beans::VetoableChangeSupport*

En este caso se tiene que la clase *beans::VetoableChangeSupport* es el *Subject* principal ya que es la clase encargada de dar soporte a algún cambio de interés que pueda ocurrir para esta instancia. A continuación se mostrará el código del método *fireVetoableChange()*:

```
public void fireVetoableChange(PropertyChangeEvent evt)
                              throws PropertyVetoException {
…
for (int i = 0; i < targets.size(); i++) {
                  VetoableChangeListener target =
                        (VetoableChangeListener)targets.elementAt(i);
                  target.vetoableChange(evt);
              }
… }
```

Como se puede observar, en este método existe una llamada directa al método *vetoableChange()* a diferencia del *Subject* anterior en el que los autores crearon un objeto de la clase *VetoableChangeSupport* y a partir de éste se realizó la llamada al método *vetoableChange()*. Los autores reutilizaron este método para el resto de los *Subject* en esta instancia.

3 Subject: *JComponent*

ConcreteSubject: *JComponent*

Al igual que el *Subject beancontext::BeanContextChild* se tiene que en el método *fireVetoableChange()* los autores de esta clase utilizaron un objeto de la clase *VetoableChangeSupport* para hacer una llamada al método *fireVetoableChange()* y a su vez al método *vetoableChange()* por medio de este método.

A continuación se muestra el código fuente del método *fireVetoableChange()*:

```
private VetoableChangeSupport vetoableChangeSupport;
protected void fireVetoableChange(String propertyName, Object oldValue,
Object newValue)
        throws java.beans.PropertyVetoException
    {
        if (vetoableChangeSupport == null) {
            return;
        }
```

```
        vetoableChangeSupport.fireVetoableChange(propertyName,  oldValue,
newValue);
    }
```

Finalmente se puede concluir que esta es una instancia válida del patrón *Observer* de acuerdo a los criterios establecidos ya que estos criterios se cumplieron satisfactoriamente, además de que la estructura de la instancia es la adecuada.

4.2 Decorator.

4.2.1 Instancia DEC_decoratedFigure (JHotDraw).

Diagrama de clases.

La Figura 15 muestra el diagrama de clases de la instancia.

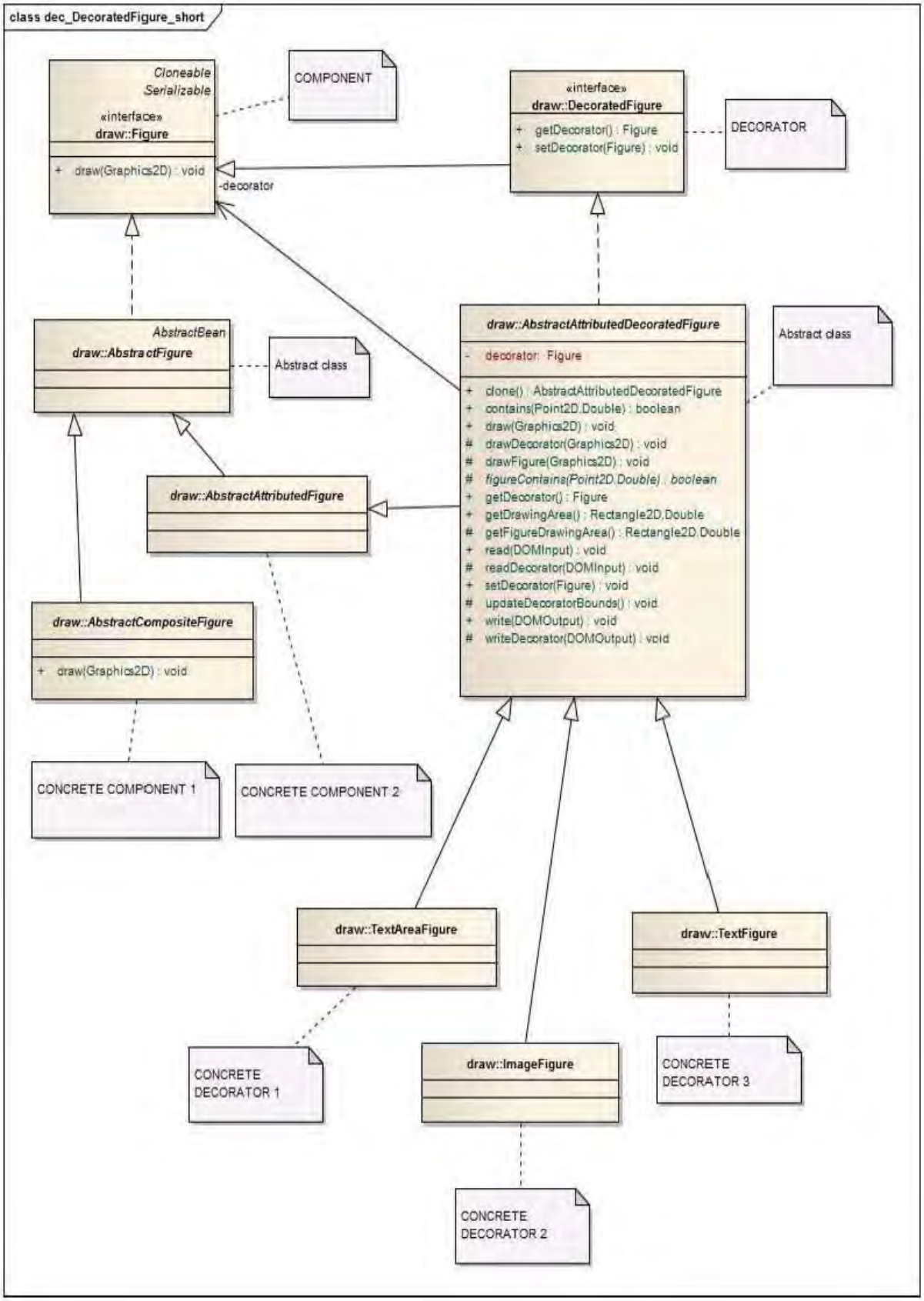

Figura 15, Diagrama de clases de la instancia *DEC_DecoratedFigure*.

Verificación de los criterios de estructura.

La Tabla 31 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 4 y la Figura 7, identificando cada uno de los elementos de la Tabla 4 en la Figura 15.

Elemento de la Figura 7	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Component	Figure
2	ConcreteComponent	AbstractAtributedFigure, AbstractCompositeFigure
3	Decorator	DecoratedFigure
4	ConcreteDecorator	TextFigure, ImageFigure, TextAreaFigure
5	wrappedObj	Figure decorator
6	newBehavior()	clone(), contains(), draw(), drawDecorator(), drawFigure(), figureContains(), write()
7	newState	---

Tabla 31, Verificación de los criterios de estructura de la instancia *DEC_DecoratedFigure*.

Verificación de los criterios de interrelación.

La Tabla 32 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 5 y la Figura 7, identificando cada uno de los elementos en la Figura 15.

Elemento de la Figura 7	Descripción	Verificado
a	El *AbstractFigure* implementa la interfaz *Figure*.	*Sí*
b	La interfaz *DecoratedFigure* hereda de la interfaz *Figure*.	*Sí*
c	The *AbstractAttributedDecoratedFigure* implementa la interfaz *DecoratedFigure*.	*Sí*

Tabla 32, Verificación de los criterios de interrelación de la instancia *DEC_DecoratedFigure*.

Otros criterios.

La Tabla 33 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 6 y la Figura 7, identificando cada uno de los elementos de la tabla en la Figura 15 y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
DEC_1	La clase abstracta *AbstractAtributtedDecoratedFigure* y las clases que implementan esta clase tienen varios métodos que proveen comportamiento diferente a los *ConcreteComponents*, por ejemplo, el método *drawImage()* provee de nuevo comportamiento para la clase *ImageFigure*.	*Sí*

DEC_2	La interfaz *DecoratedFigure* solamente envuelve a un solo objeto *Figure*, como se puede observar en el método *setDecorator(Figure)*.	*Sí*

Tabla 33, Verificación de otros criterios de la instancia *DEC_DecoratedFigure*.

Conclusiones.

Figure es una interfaz destinada a la creación de elementos gráficos que se pueden dibujar solamente en un "dibujo" a la vez, el "dibujo" en este caso es un objeto de la clase *Drawing*, una figura (objeto *Figure*) puede ser, a su vez, decorada por otra figura, esto se puede notar observando la interfaz *DecoratedFigure* que es una figura que hereda de la interfaz *Figure*, y que además, decora al *Component* que en este caso es la misma interfaz *Figure* como se puede notar en el atributo "*Decorator: Figure*" en la Figura 15 y en el criterio DEC_2 el método *setDecorator(Figure newValue)* envuelve al componente "*newValue"* decorándolo, y como *newValue* puede ser una sola figura o una figura ya decorada se puede deducir que la figura puede ser decorada, es decir envuelta, cuantas veces se requiera, cumpliendo así con uno de los criterios más importantes del patrón *Decorator*: envolver a un solo objeto.

Por otro lado, se tiene que la interfaz que está destinada a decorar a un *Component* fue nombrada como *DecoratedFigure* (figura decorada), en términos prácticos este nombre no describe correctamente el comportamiento de esta interfaz ya que esta clase está jugando el rol de *Decorator* no de *Component*, recordar que el *Component* es el objeto que es decorado y el *Decorator* el objeto que lo decora o agrega dinámicamente nuevo comportamiento, llamarla *FigureDecorator* (decorador de figuras) pudo haber sido una mejor forma de nombrar a esta interfaz.

Sin embargo, se puede notar que esta instancia cumple con todos los criterios, además el comportamiento de ésta, como se dijo anteriormente, es un comportamiento esperado de una instancia del patrón *Decorator*, es por esto que se puede decir que esta instancia es una instancia válida del patrón *Decorator* de acuerdo a los criterios establecidos.

4.2.2 Instancia DEC_LineDecoration (JHotDraw).

Diagrama de clases.

La *Figura 16* muestra el diagrama de clases de la instancia.

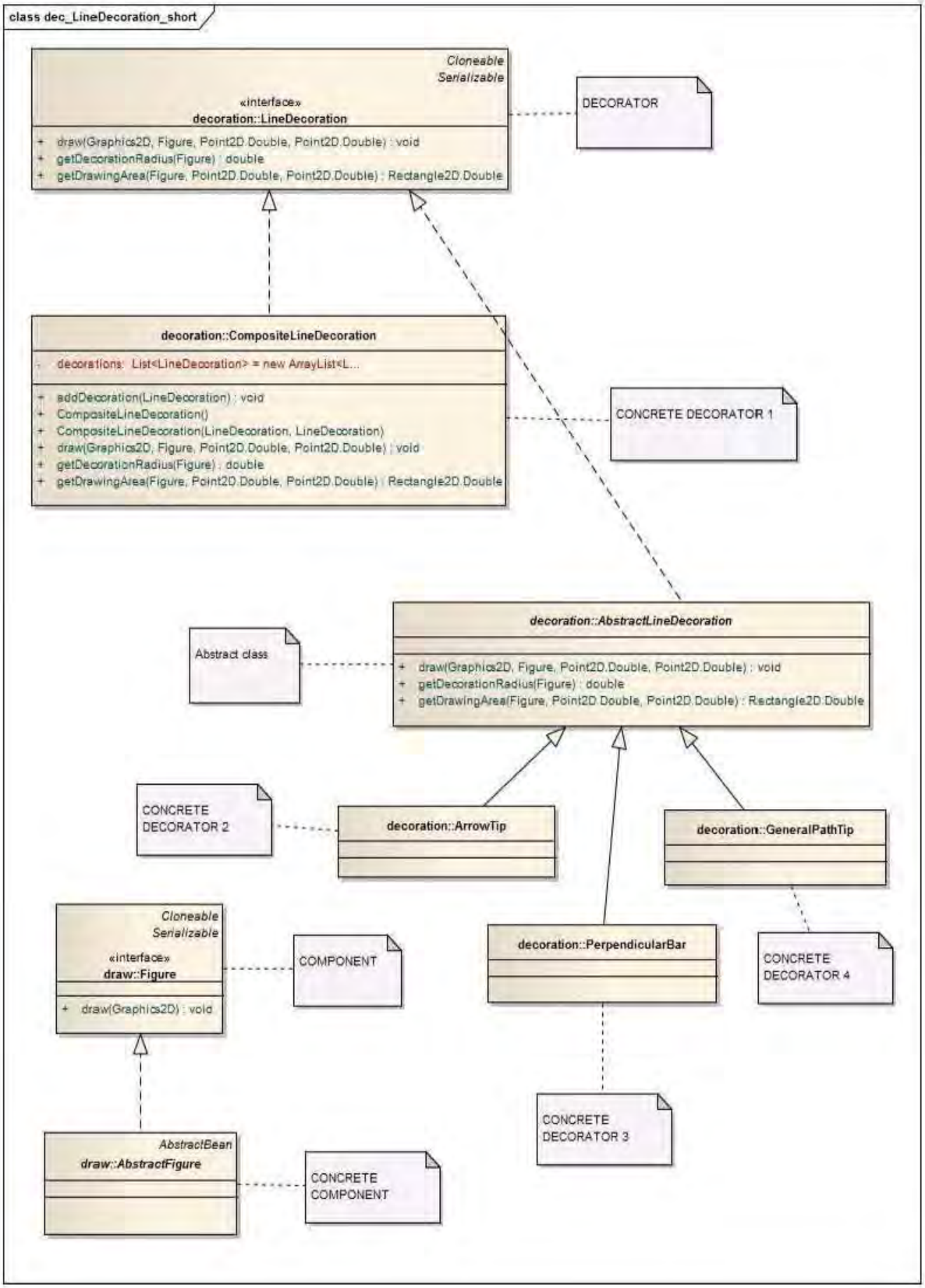

Figura 16, Diagrama de clases de la instancia DEC_LineDecoration.

Verificación de los criterios de estructura.

La Tabla 34 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 4 y la Figura 7, identificando cada uno de los elementos de la Tabla 4 en la *Figura 16*.

Elemento de la Figura 7	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Component	Figure
2	ConcreteComponent	AbstractFigure
3	Decorator	LineDecoration
4	ConcreteDecorator	CompositeLineDecoration ArrowTip, GeneralPathTip, PerpendicularBar
5	wrappedObj	*List<LineDecoration>*
6	newBehavior()	*CompositeLineDecoration(), addDecoration()*
7	newState	---

Tabla 34, Verificación de los criterios de estructura de la instancia *DEC_LineDecoration.*

Verificación de los criterios de interrelación.

La Tabla 35 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 5 y la Figura 7, identificando cada uno de los elementos en la *Figura 16*.

Elemento de la Figura 7	Descripción	Verificado
a	La clase *CompositeLineDecoration* implementa la interfaz *LineDecoration*.	*Sí*
b	La clase *LineDecoration* implementa la interfaz *Figure*.	*No*
c	La clase *AbstractFigure* implementa la interfaz *Figure*.	*Sí*

Tabla 35, Verificación de los criterios de interrelación de la instancia *DEC_LineDecoration.*

Otros criterios.

La Tabla 36 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 6 y la Figura 7, identificando cada uno de los elementos de la tabla en la *Figura 16* y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
DEC_1	Las clases *CompositeLineDecoration*, *ArrowTip, PerpendicularBar* y *GeneralPathTip* (*ConcreteDecorators*) agregan nuevo comportamiento a la clase abstracta *AbstractFigure* (*ConcreteComponent*). Por ejemplo el método *getDecorationRadius(Figure)* está destinado a obtener el radio del decorador, es decir, de un objeto *Figure* determinado.	*Sí*

DEC_2	En la clase *CompositeLineDecoration* el método *addDecoration()* debe envolver a un solo objeto del *LineDecoration*. En el *CompositeLineDecoration* se tiene el siguiente atributo: *private List<LineDecoration> decorations = new ArrayList<LineDecoration>();* el cual es un conjunto de objetos *LineDecoration*.	*No*
	Las clases *ArrowTip, GeneralPathTip, PerpendicularBar* envuelven sólo un objeto *Figure*. En la clase abstracta *AbstractLineDecoration* se tienen las siguientes líneas de código: *public void draw(Graphics2D g, Figure f, Point2D.Double p1, Point2D.Double p2) {}* Este método es empleado para dibujar el decorador a una sola figura "f".	*Sí*

Tabla 36, Verificación de otros criterios de la instancia *DEC_LineDecoration.*

Conclusiones.

Como se puede observar en el análisis anterior, uno de los criterios de interrelación no fue satisfecho (criterio b de de la Figura 7), esto indica que el decorador (en este caso es el *LineDecoration*) no es un objeto *Figure* por lo que el concepto de envolver un *Component* con un *Decorator*, como se describió a detalle en la descripción de este patrón, no fue satisfecho. Es decir, suponer que se tiene un *Component* ya decorado y que se desea decorarlo nuevamente con otro *Decorator* en esta instancia no será posible ya que este objeto ya decorado no es un *Component* por lo que no se puede introducir como argumento en ninguno de los métodos de los *ConcreteDecorator* para ser decorado.

Por otro lado, se tiene que el criterio DEC_2 tampoco fue satisfecho, como se describió anteriormente el *Decorator* sólo puede envolver a un solo *Component* y en esta instancia se puede decorar con más de un objeto como se puede notar en las siguientes líneas de código del método *addDecoration()* de la clase *CompositeLineDecoration*:

```
private List<LineDecoration> decorations = new ArrayList<LineDecoration>();
public void addDecoration(LineDecoration decoration) {
      if (decoration != null) {
            decorations.add(decoration);
      }
   }
```

Debido a que es una colección de objetos, se puede agregar cuantos objetos *LineDecoration* se quieran al mismo objeto de la clase *CompositeLineDecoration* llamando al método *addDecoration()*.

Se decidió tomar ésta como posible instancia por la estructura que tiene en general y por el nombre que se le dio a las clases e interfaces que forman parte de ella, ya que esto da una idea de decoración. Estrictamente hablando, se puede decir que la interfaz *LineDecoration* es un decorador pero no se puede decir que es una instancia del patrón *Decorator*. Como se vio anteriormente, la clase *CompositeLineDecoration* (que es uno de los *ConcreteObservers*) se puede decorar así misma y las clases *ArrowTip, GeneralPathTip* y *PerpendicularBar* son también objetos que decoran figuras pero no son figuras ya que no tienen ninguna relación con esta interfaz.

Finalmente se concluye que ésta no es una instancia válida de acuerdo a los criterios establecidos.

4.2.3 Instancia DEC_JComponent (Swing).

Diagrama de clases.

La Figura 17 muestra el diagrama de clases de la instancia.

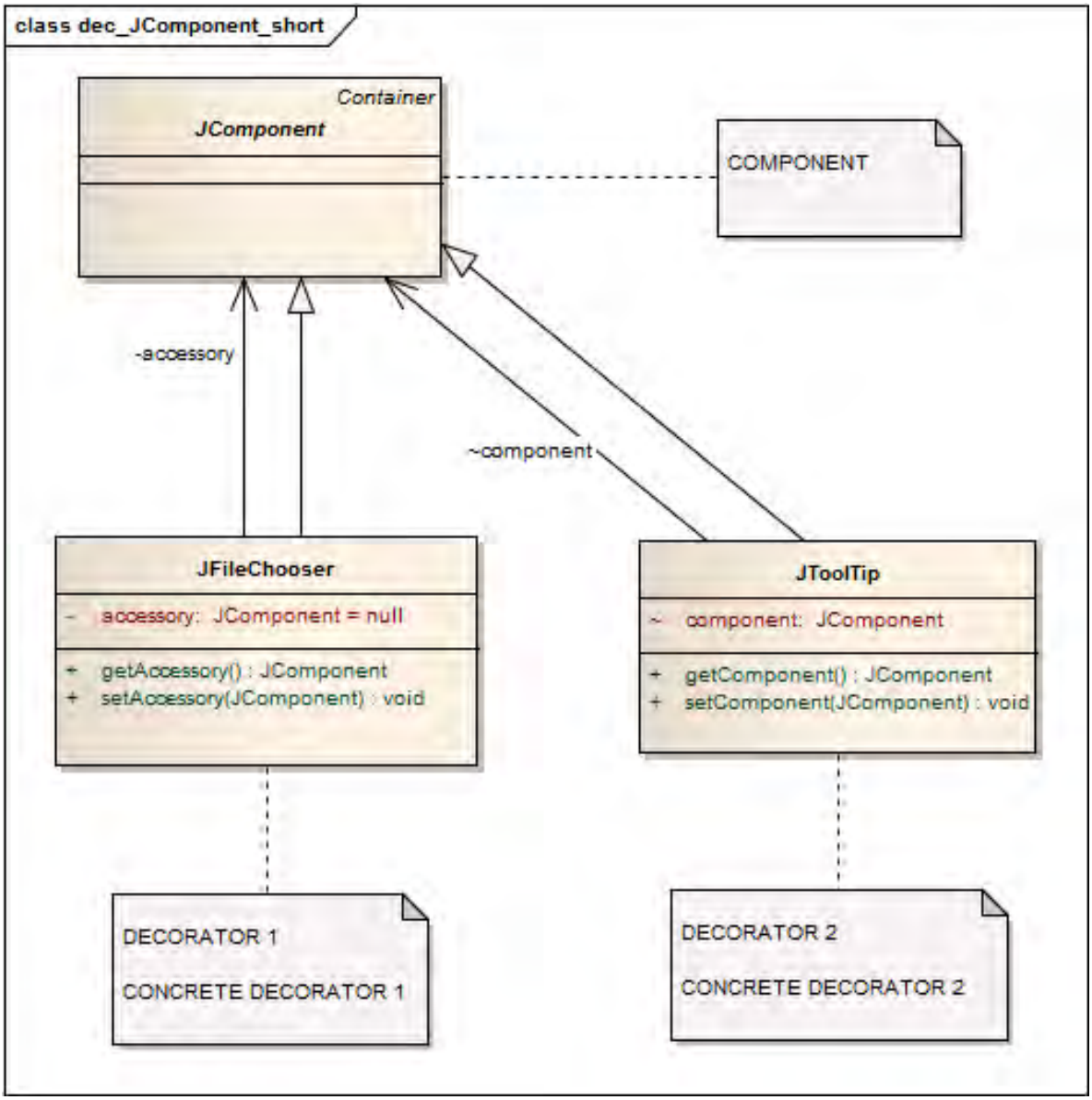

Figura 17, Diagrama de clases de la instancia *DEC_JComponent*.

Nota 1: Los *ConcreteComponent* no se muestran en el diagrama debido a que son demasiados.

Nota 2: Los métodos que proveen nuevo comportamiento tampoco se muestran en el diagrama de clases debido a la misma razón.

Verificación de los criterios de estructura.

La Tabla 37 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 4 y la Figura 7, identificando cada uno de los elementos de la Tabla 4 en la Figura 17.

Elemento de la Figura 7	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Component	JComponent
2	ConcreteComponent	AbstractButton, BasicInternalFrameTitlePane, Box, Box.Filler, JColorChooser, JComboBox, JFileChooser, JInternalFrame, JInternalFrame.JDesktopIcon, JLabel, JLayeredPane, JList, JMenuBar, JOptionPane, JPanel, JPopupMenu, JProgressBar, JRootPane, JScrollBar, JScrollPane, JSeparator, JSlider, JSpinner, JSplitPane, JTabbedPane, JTable, JTableHeader, JTextComponent, JToolBar, JToolTip, JTree, JViewport.
3	Decorator	JFileChooser, JToolTip.
4	ConcreteDecorator	JFileChooser, JToolTip.
5	wrappedObj	accessory (JFileChooser), component (JToolTip).
6	newBehavior()	El resto de los métodos de las clases Decorator.
7	newState	---

Tabla 37, Verificación de los criterios de estructura de la instancia *DEC_JComponent.*

Verificación de los criterios de interrelación.

La Tabla 38 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 5 y la Figura 7, identificando cada uno de los elementos en la Figura 17.

Elemento de la Figura 7	Descripción	Verificado
a	Todas las clases *ConcreteComponent* heredan de la clase *JComponent*.	*Sí*
b	Las clases *JFileChooser* y *JToolTip* heredan de la clase *JComponent*..	*Sí*
c	Las clases *ConcreteDecorator* heredan de las clases *Decorator*.	*NA*

Tabla 38, Verificación de los criterios de interrelación de la instancia *DEC_JComponent.*

NA: No aplica.

Otros criterios.

La Tabla 39 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 6 y la Figura 7, identificando cada uno de los elementos de la tabla en la Figura 17 y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
DEC_1	El o los métodos de las clases *JFileChooser* y *JToolTip* deberán proveer nuevo comportamiento a las clases *ConcreteComponent*, de forma que, al llamar estos métodos se puede hacer uso de este nuevo comportamiento.	*Sí*
DEC_2	Las clases *JFileChooser* y *JToolTip* deben envolver a un solo *Component*.	*Sí*

Tabla 39, Verificación de otros criterios de la instancia *DEC_JComponent.*

Conclusiones.

En esta instancia se puede notar que el *Component* es la clase principal de la biblioteca *Swing*, esta clase es la clase base de todos los componentes que se pueden utilizar con la biblioteca *Swing* de acuerdo a la arquitectura con la que fue construida.

Se puede notar también que todos los *ConcreteComponents* son todos los elementos que pueden ser utilizados al programar con la librería *Swing*, recuérdese que *Swing* es una biblioteca Java que tiene elementos gráficos que se utilizan en las interfaces gráficas de aplicaciones web, entre otras.

Además, en esta instancia se puede notar también que se cumplieron todos los criterios por lo que la estructura y las interrelaciones son correctas en este caso.

Se puede notar también que la clase *JFileChooser* tiene como función el proveer un mecanismo simple para escoger un archivo. Esta clase tiene una instancia de *JComponent* (*accesory*) la cual es utilizada para asignar un componente (cualquier *JComponent*) a algún objeto *JFileChooser*, esto con el objetivo de que este componente funcione como un atributo de forma que pueda ser utilizado por el programador como mejor le convenga.

Por otra parte, la clase *JToolTip* fue creada para asociar determinados atributos a un objeto de la clase *JComponent*, como por ejemplo, el atributo "*tipText*" es un atributo que sirve para asignar texto a un objeto *JToolTip* de modo que pueda mostrarse cuando el componente *JToolTip* sea mostrado. Esta clase también tiene una instancia de la clase *JComponent* y puede ser utilizado de igual forma que en la clase anterior.

Finalmente se puede concluir que ésta es una instancia válida del patrón *Decorator* de acuerdo a los criterios y el análisis descritos anteriormente.

4.3 Adapter.

4.3.1 Instancia ADP_TextFigure (JHotDraw).

Diagrama de clases

La Figura 18 muestra el diagrama de clases de la instancia.

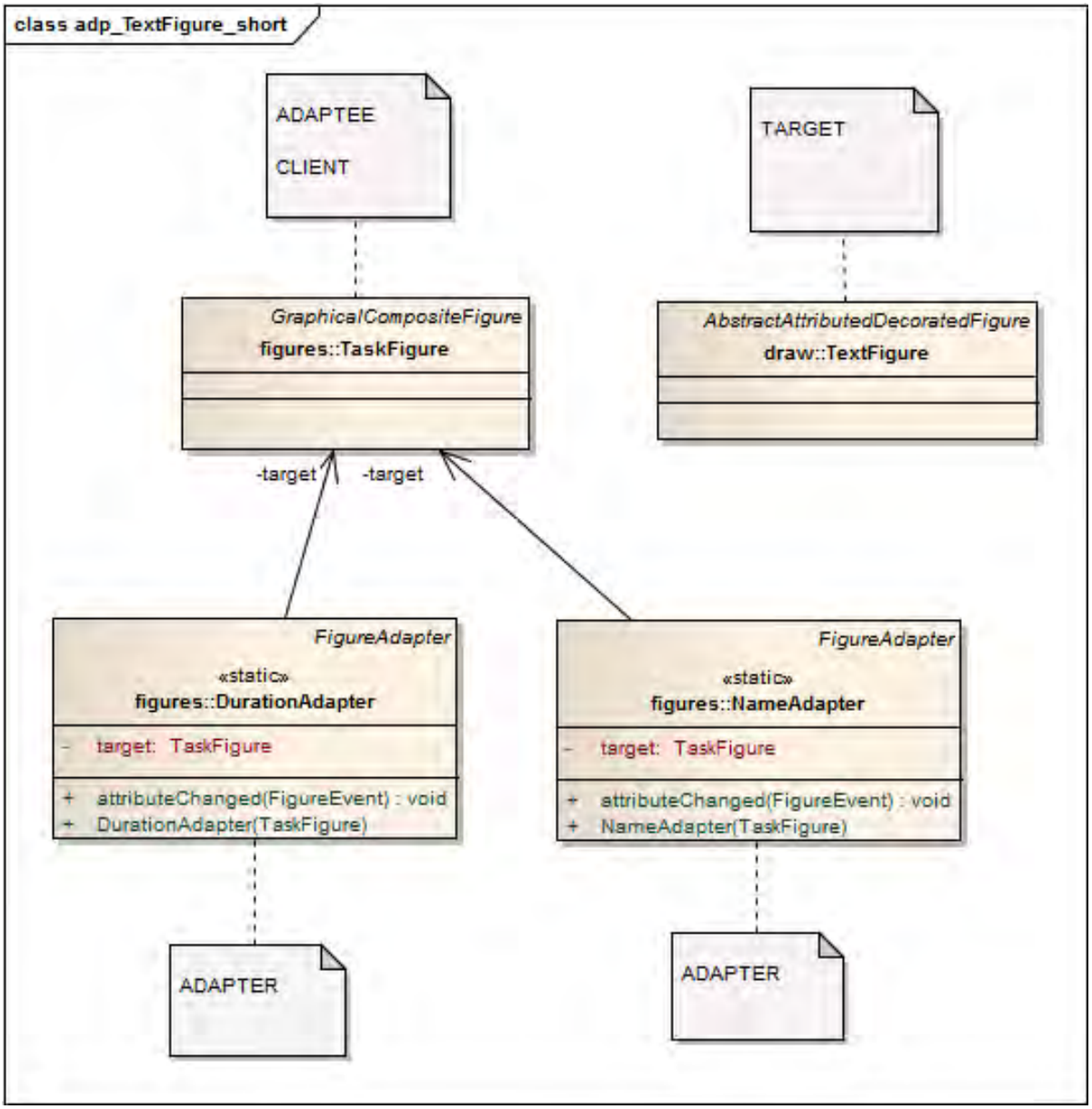

Figura 18, Diagrama de clases de la instancia *ADP_TextFigure*.

Verificación de los criterios de estructura.

La Tabla 40 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 7 y la **¡Error! No se encuentra el origen de la referencia.**, identificando cada uno de los elementos de la Tabla 7 en la Figura 18.

Elemento de la ¡Error! No se encuentra el origen de la referencia.	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Client	TaskFigure
2	Target	TextFigure
3	Adapter	NameAdapter, DurationAdapter
4	Adaptee	TaskFigure
5	*request()*	---
6	*request()*	*NameAdapter(TaskFigure), DurationAdapter(TaskFigure)*
7	*specificRequest()*	*nameFigure.addFigureListener(new NameAdapter(this)), durationFigure.addFigureListener(new DurationAdapter(this));*

Tabla 40, Verificación de los criterios de estructura de la instancia *ADP_TextFigure*.

Verificación de los criterios de interrelación.

La Tabla 41 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 8 y la **¡Error! No se encuentra el origen de la referencia.**, identificando cada uno de los elementos en la Figura 18.

Elemento de la ¡Error! No se encuentra el origen de la referencia.	Descripción	Verificado
a	En la clase *TextFigure* se crea un objeto de la clase *NameAdapter* (*new NameAdapter(this)*) para obtener el nombre del modelo *TaskFigure*.	*Sí*
b	El *NameAdapter* debe heredar de la clase *TextFigure* o implementarla en caso de que sea una interfaz.	*No*
c	The *NameAdapter* tiene una instancia de la clase *TaskFigure*, como se puede ver en la siguiente línea: *private TaskFigure target;*	*Sí*

Tabla 41, Verificación de los criterios de interrelación de la instancia *ADP_TextFigure*.

Otros criterios.

La Tabla 42 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 9 y la **¡Error! No se encuentra el origen de la referencia.**, identificando cada uno de los elementos de la tabla en la Figura 18 y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
ADP_1	La clase *TaskFigure* llama al *Adaptee* en el método *addFigureListener()* como se puede notar a continuación: *nameFigure.addFigureListener(**new** NameAdapter(**this**))* *durationFigure.addFigureListener(new DurationAdapter(this));* Como se puede observar, el llamado queda delegado a la misma clase *TaskFigure*.	*Sí*
ADP_2	*TaskFigure* y *TextFigure* deben tener una relación conceptual, es decir, las tareas de *TaskFigure* y *TextFigure* deben tener objetivos similares pero estos pueden ser llevados a cabo de formas diferentes.	*No*

Tabla 42, Verificación de otros criterios de la instancia *ADP_TextFigure*.

Conclusiones.

Estos *Adapter* fueron creados para obtener el nombre del modelo de un objeto *TaskFigure*, como se pudo notar anteriormente, *TaskFigure* juega el rol de *Client* y al mismo tiempo el de *Adaptee*, no se puede decir que este hecho indique que esta no es una instancia válida. Sin embargo, se puede notar que esta estructura da indicios de que muy probable no sea válida. Como se sabe, el principal objetivo del patrón *Adapter* es hacer posible la comunicación entre dos entes que no son compatibles y en este caso no es lógico que los *Adapter* permitan que *TaskFigure* (el *Client*) utilice el *TextFigure* para comunicarse consigo mismo.

Por otro lado se tiene que uno de los criterios de interrelación no fue satisfecho, *NameAdapter* no tiene relación alguna con *TextFigure*; esta relación es importante porque con esa relación se puede tener más de un *Adapter* de manera que se pueden "conectar" más de un *Adaptee* a un mismo *Target*, lo que en este caso no es posible.

Se tomó en cuenta ésta como instancia potencial debido a que los autores utilizaron la palabra "*Adapter*" para darle nombre a algunas clases, es por esta razón que es importante considerar el nombre que se le va a dar a las clases, sin embargo, en este caso se tiene un adaptador como lo indican los autores en la descripción de la clase estática *NameAdapter* como se muestra a continuación:

```
/**
* This adapter is used, to connect a TextFigure with the name of the TaskFigure
model.
*/
```

Sin embargo, crear un adaptador para obtener solamente el nombre del modelo del *TaskFigure* es un tanto excesivo, una mejor forma pudo haber sido, por ejemplo, crear un método que realizara esta tarea, sin embargo los autores decidieron crear clases estáticas como adaptadores para obtener este dato.

Como conclusión final se puede decir que las clases *NameAdapter* y *DurationAdapter* son adaptadores, sin embargo, no son instancias válidas del patrón *Adapter* de acuerdo a los criterios establecidos.

4.3.2 Instancia ADP_FigureSelectionListener (JHotDraw).

Diagrama de clases.

La Figura 19 muestra el diagrama de clases de la instancia.

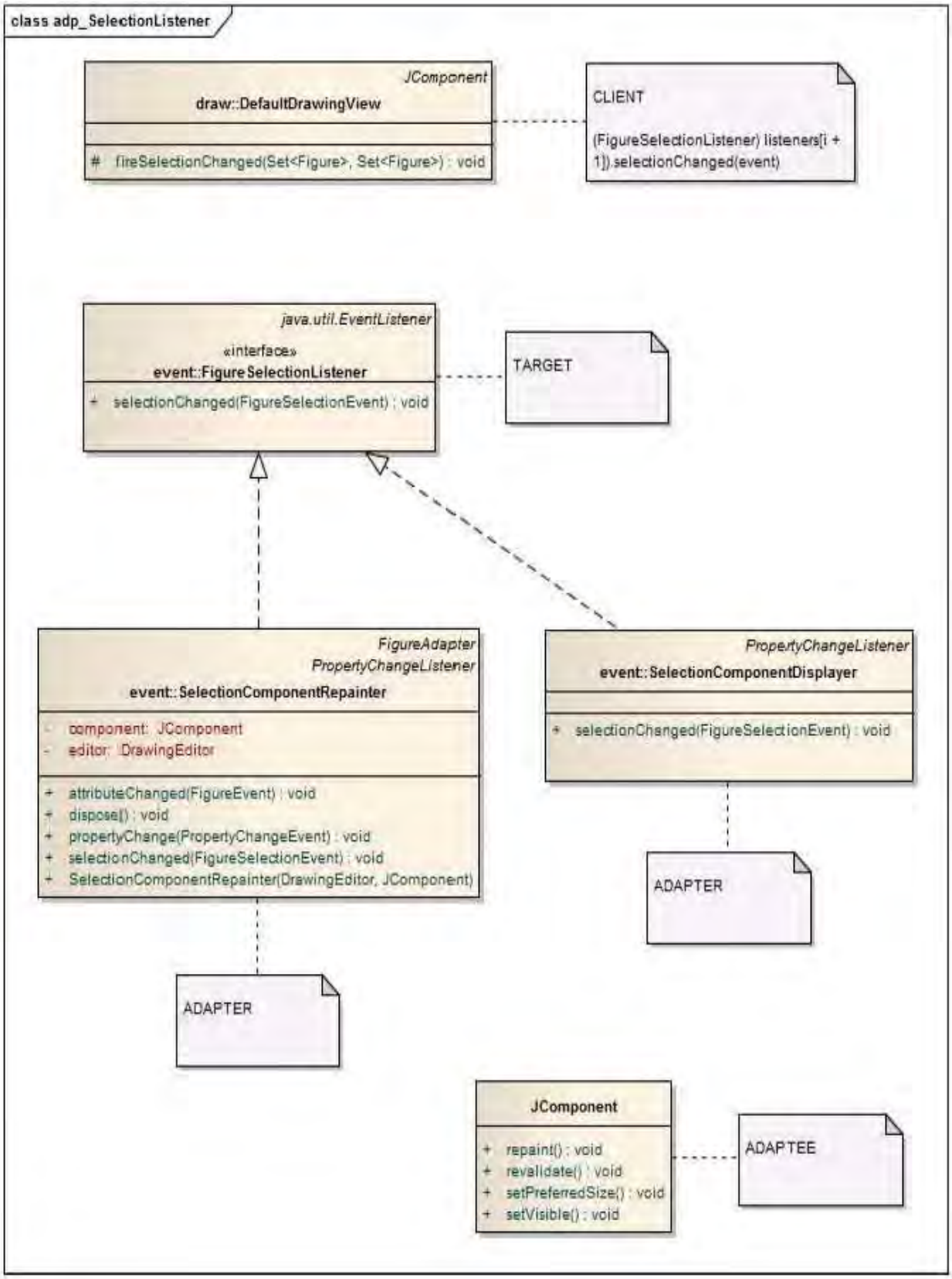

Figura 19, Diagrama de clases de la instancia *ADP_FigureSelectionListener*.

Verificación de los criterios de estructura.

La Tabla 43 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 7 y la **¡Error! No se encuentra el origen de la referencia.**, identificando cada uno de los elementos de la Tabla 7 en la Figura 19.

Elemento de la ¡Error! No se encuentra el origen de la referencia.	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Client	DefaultDrawingView
2	Target	FigureSelectionListener
3	Adapter	SelectionComponentRepainter, SelectionComponentDisplayer
4	Adaptee	JComponent
5	request()	*selectionChanged(FigureSelectionEvent)*
6	request()	*selectionChanged(FigureSlectionEvent)*
7	specificRequest()	SelectionComponentRepainter: *repaint()* SelectionComponentDisplayer: *revalidate()* *setPreferedSize()* *setVisible()*

Tabla 43, Verificación de los criterios de estructura de la instancia *ADP_FifureSelectionListener.*

Verificación de los criterios de interrelación.

La Tabla 44 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 8 y la **¡Error! No se encuentra el origen de la referencia.**, identificando cada uno de los elementos en la Figura 19.

Elemento de la ¡Error! No se encuentra el origen de la referencia.	Descripción	Verificado
a	*DefaultDrawingView* hace un llamado al *SelectionComponentRepainter* por medio del método *fireSelectionChanged()*, como se muestra en la siguiente línea: *(FigureSelectionListener) listeners[i + 1]).selectionChanged(event)*	*Sí*
b	La clase *SelectionComponentRepainter* implementa la interfaz *FigureSelectionListener*.	*Sí*

	La clase *SelectionComponentDisplayer* implementa la interfaz *FigureSelectionListener*.	*Sí*
c	*SelectionComponentRepainter* mantiene una instancia del *JComponent*, como se puede notar en la siguiente línea: *private JComponent component;*	*Sí*
	SelectionComponentDisplayer mantiene una instancia del *JComponent*, en el método *updateVisibility()* como se puede observar en la siguiente línea: *JComponent component* *component.setVisible(newValue);*	*Sí*

Tabla 44, Verificación de los criterios de interrelación de la instancia *ADP_FigureSelectionListener.*

Otros criterios.

La Tabla 45 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 9 y la **¡Error! No se encuentra el origen de la referencia.**, identificando cada uno de los elementos de la tabla en la Figura 19 y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
ADP_1	**SelectionComponentRepainter** La llamada del *Client* se realiza por medio del método *selectionChanged()*, dentro de éste existe otra llamada al método *repaint()* como se puede notar en las siguientes líneas de código: *public void selectionChanged(FigureSelectionEvent evt) {* *component.repaint();* *}*	*Sí*
	SelectionComponentDisplayer La llamada del *Client* se realiza por medio del método *selectionChanged()*, dentro de éste existe otra llamada al método *updateVisibility()* y, al mismo tiempo, en este mismo método existe una instancia de *JComponent* como se puede notar en las siguientes líneas de código: *JComponent component = weakRef.get();* *component.setVisible(newValue);* *component.setPreferredSize(null);* *component.revalidate();*	*Sí*
ADP_2	*FigureSelectionListener* y *JComponent* deben tener una relación conceptual, es decir, las tareas de *FigureSelectionListener* y *JComponent* deben tener objetivos similares pero estos pueden ser llevados a cabo de formas diferentes.	*No*

Tabla 45, Verificación de otros criterios de la instancia *ADP_FigureSelectionListener.*

Conclusiones.

Esta instancia en particular es especial ya que como se pudo observar en el análisis anterior, cumple con toda la estructura, en el caso de los criterios dinámicos (otros criterios) se tiene que el ADP_1 lo cumplen ambos adaptadores, es decir todas las llamadas que el *Client* realiza siempre quedan delegadas al *Adaptee*.

Sin embargo, observando esta instancia desde una perspectiva lógica y práctica se puede notar que no tiene mucho sentido crear una instancia del patrón *Adapter* para poder comunicar a las clases *FigureSelectionListener* y *JComponent*, además se puede notar también que estas dos clases tienen objetivos diferentes. La interfaz *FigureSelectionListener* es un *Observer* del *Subject FigureSelection* el cual fue creado para "escuchar" cuando la selección de algún objeto (entiéndase como objeto de un dibujo) cambia en un *DrawingView* y, por otro lado, se tiene que *JComponent* es una clase que fue diseñada para crear componentes de la biblioteca *Swing*. De lo anterior se puede notar que estas dos clases tienen objetivos completamente diferentes, de aquí que el criterio ADP_1 no fue satisfecho, y el objetivo de un *Adapter* es precisamente el permitir la comunicación entre dos entes que tienen objetivos similares pero que la llevan a cabo de diferentes maneras y en este caso no es así.

De acuerdo al análisis anterior se puede concluir que ésta no es una instancia válida del patrón *Adapter* de acuerdo a los criterios establecidos.

4.3.3 Instancia ADP_FigureListener (JHotDraw).

Diagrama de clases.

La Figura 20 muestra el diagrama de clases de la instancia.

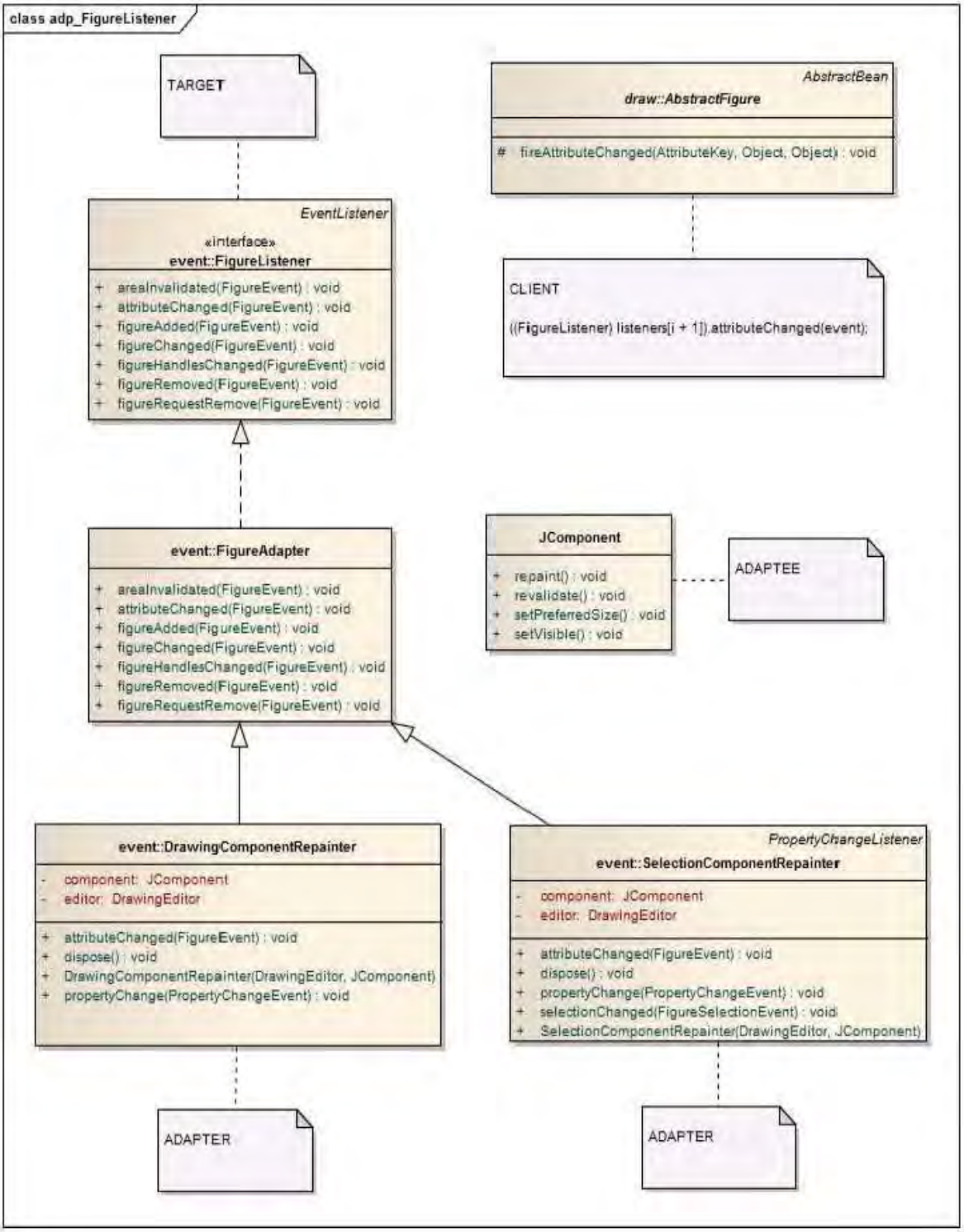

Figura 20, Diagrama de clases de la instancia *ADP_FigureListener*.

Verificación de los criterios de estructura.

La Tabla 46 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 7 y la **¡Error! No se encuentra el origen de la referencia.**, identificando cada uno de los elementos de la Tabla 7 en la Figura 20.

Elemento de la ¡Error! No se encuentra el origen de la referencia.	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Client	AbstractFigure
2	Target	FigureListener
3	Adapter	Adapter: FigureAdapter ConcreteAdapters: DrawingComponentRepainter SelectionComponentRepainter
4	Adaptee	JComponent
5	*request()*	*attributeChanged()*
6	*request()*	*attributeChanged()*
7	*specificRequest()*	*repaint()*

Tabla 46, Verificación de los criterios de estructura de la instancia *ADP_FigureListener.*

Verificación de los criterios de interrelación.

La Tabla 47 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 8 y la **¡Error! No se encuentra el origen de la referencia.**, identificando cada uno de los elementos en la Figura 20.

Elemento de la ¡Error! No se encuentra el origen de la referencia.	Descripción	Verificado
a	*AbstractFigure* hace una petición a *DrawingComponentRepainter* y *SelectionComponentRepainter* por medio de *FigureListener* y *FigureAdapter*, como se puede notar en la siguiente línea: *((FigureListener) listeners[i + 1]).attributeChanged(event);*	*Sí*
b	*DrawingComponentRepainter* y *SelectionComponentRepainter* heredan de la clase *FigureAdapter* y, a su vez, *FigureAdapter* implementa la interfaz *FigureListener*.	*Sí*
c	*DrawingComponentRepainter* y *SelectionComponentRepainter* poseen una instancia de *JComponent*.	*Sí*

Tabla 47, Verificación de los criterios de interrelación de la instancia *ADP_FigureListener.*

Otros criterios.

La Tabla 48 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 9 y la **¡Error! No se encuentra el origen de la referencia.**, identificando cada uno de los elementos de la tabla en la Figura 20 y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
ADP_1	El método *attributedChanged()* del objeto *FigureListener* es llamado en *AbstractFigure*, por medio de este método la llamada del cliente queda delegada a *JComponent* como se puede notar a continuación: *JComponent component;* *public void attributeChanged(FigureEvent evt) {* *component.repaint();* *}*	*Sí*
ADP_2	*FigureListener* y *JComponent* deben tener una relación conceptual, es decir, las tareas de *FigureListener* y *JComponent* deben tener objetivos similares pero estos pueden ser llevados a cabo de formas diferentes.	*No*

Tabla 48, Verificación de otros criterios de la instancia *ADP_FigureListener.*

Conclusiones.

Como se pudo notar en el análisis anterior los criterios de estructura e interrelación fueron satisfechos en esta instancia, se puede observar también que esta instancia es parecida a la instancia anterior ya que la estructura es muy similar y tampoco cumple con el criterio ADP_2.

Como se vio anteriormente *FigureListener* fue diseñado para crear *Observers* para los *Subjects*, en cambio se tiene que el *Adaptee* nuevamente es *JComponent* en el cual su objetivo principal es crear componentes de la biblioteca Swing. Por lo que se tiene una instancia muy parecida a la instancia anterior y, de igual forma se puede decir que ésta no es una instancia válida del patrón *Adapter* de acuerdo a los criterios establecidos, aunque la estructura sea la de una instancia válida, el sentido y las tareas que realizan las clases que intervienen en ésta no están relacionadas con las tareas y objetivos que corresponden a las clases que forman parte de una instancia del patrón *Adapter*.

4.3.4 Instancia ADP_ComponentUI (Swing).

Diagrama de clases.

La Figura **21** muestra el diagrama de clases de la instancia.

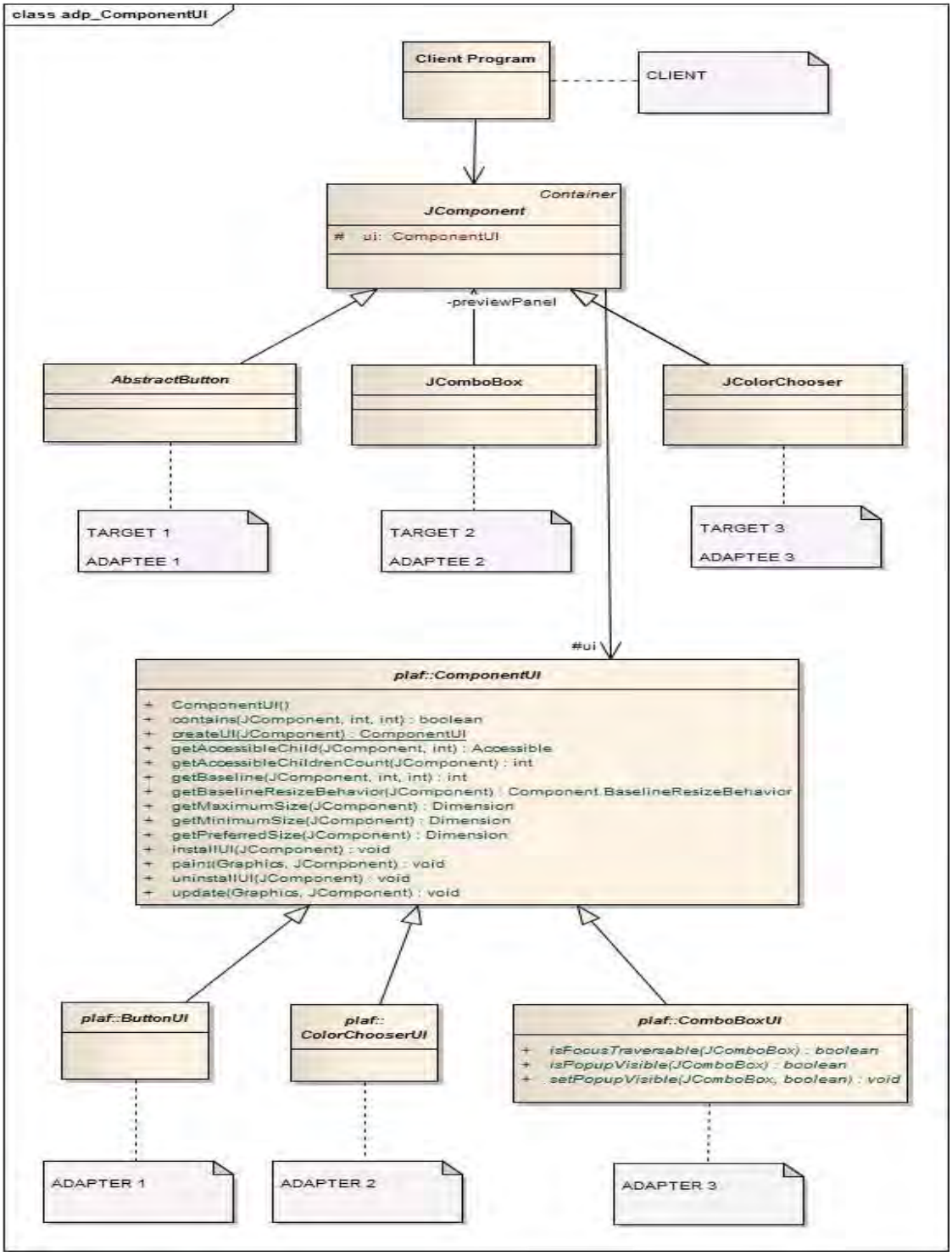

Figura 21, Diagrama de clases de la instancia *ADP_ComponentUI*.

Nota: Únicamente se muestran tres clases *Target* y *Adaptee* debido a que son demasiadas, sin embargo, en la siguiente tabla se mencionan todas.

Verificación de los criterios de estructura.

La Tabla 49 muestra el análisis y la verificación de los criterios de estructura de acuerdo a la Tabla 7 y la **¡Error! No se encuentra el origen de la referencia.**, identificando cada uno de los elementos de la Tabla 7 en la Figura **21**.

Elemento de la ¡Error! No se encuentra el origen de la referencia.	Nombre del rol en el patrón de diseño	Nombre en la instancia
1	Client	Client Program
2	Target	Todas las subclases de JComponent.
3	Adapter	*Todas las subclases de ComponentUI:* ButtonUI, ColorChooserUI, ComboBoxUI, DesktopIconUI, DesktopPaneUI, FileChooserUI, InternalFrameUI, LabelUI, ListUI, MenuBarUI, OptionPaneUI, PanelUI, PopupMenuUI, ProgressBarUI, RootPaneUI, ScrollBarUI, ScrollPaneUI, SeparatorUI, SliderUI, SpinnerUI, SplitPaneUI, TabbedPaneUI, TableHeaderUI, TableUI, TextUI, ToolBarUI, ToolTipUI, TreeUI, ViewportUI.
4	Adaptee	*Todas las subclases de JComponent:* AbstractButton, BasicInternalFrameTitlePane, Box, Box.Filler, JColorChooser, JComboBox, JFileChooser, JInternalFrame, JInternalFrame.JDesktopIcon, JLabel, JLayeredPane, JList, JMenuBar, JOptionPane, JPanel, JPopupMenu, JProgressBar, JRootPane, JScrollBar, JScrollPane, JSeparator, JSlider, JSpinner, JSplitPane, JTabbedPane, JTable, JTableHeader, JTextComponent, JToolBar, JToolTip, JTree, JViewport.
5	*request()*	…
6	*request()*	*Todos los métodos de las clases Adapter.*
7	*specificRequest()*	*Todos los métodos de las clases Adaptee.*

Tabla 49, Verificación de los criterios de estructura de la instancia *ADP_ComponentUI.*

Verificación de los criterios de interrelación.

La Tabla 50 muestra el análisis y la verificación de los criterios de interrelación de acuerdo a la Tabla 8 y la **¡Error! No se encuentra el origen de la referencia.**, identificando cada uno de los elementos en la Figura **21**.

Elemento de la ¡Error! No se encuentra el origen de la referencia.	Descripción	Verificado
a	El *Client Program* debe hacer una llamada al *Adapter* llamando a un método en él a través de la interfaz *Target*.	*Sí*
b	El *Adapter* debe implementar la interfaz *Target* o, en su defecto, heredar de ésta.	*No*
c	El *Adapter* debe tener una instancia del *Adaptee* o, implementar o heredar del *Adaptee*.	*No*

Tabla 50, Verificación de los criterios de interrelación de la instancia *ADP_ComponentUI.*

Otros criterios.

La Tabla 51 muestra el análisis y la verificación de otros criterios de acuerdo a la Tabla 9 y la **¡Error! No se encuentra el origen de la referencia.**, identificando cada uno de los elementos de la tabla en la Figura **21** y analizando que la descripción de cada uno de los criterios coincida con la descripción del criterio en la instancia.

Criterio	Descripción	Verificado
ADP_1	Todas las llamadas del *Client* quedan delegadas al *Adaptee*.	*No*
ADP_2	El *Target* y el *Adaptee* deben estar conceptualmente relacionados, es decir, deben tener las mismas tareas pero cada uno debe de llevarlas a cabo de forma diferente.	*Sí*

Tabla 51, Verificación de otros criterios de la instancia *ADP_ComponentUI.*

Conclusiones.

La clase *ComponentUI* y todas las subclases de ésta fueron diseñadas para dar soporte a la arquitectura *"look and feel"* de *Swing*, que básicamente es una arquitectura en la que se puede personalizar la interfaz gráfica de los componentes de *Swing*. Un objeto UI delegado por alguno de los componentes de *Swing* es responsable de implementar los aspectos de un componente que dependen de su apariencia, es decir, de la forma en que se muestran en la interfaz gráfica de la aplicación.

Esta instancia en particular se puede notar que no cumple con tres de los criterios, por lo que se puede decir que esta instancia no cumple con la estructura del patrón *Adapter*. La clase "*Client Program*" no es parte de la biblioteca Swing, esta clase fue creada con la finalidad de ilustrar cómo un programa diseñado por algún usuario de *Swing* puede hacer uso de este adaptador.

Las subclases de *JComponent*, que son las clases *Target*, poseen una instancia del *Adapter* y además estas clases son, a su vez, los *Adaptees*. Esto puede parecer una inconsistencia de esta instancia, sin embargo, este adaptador fue diseñado para que se pudiera utilizar dinámicamente cualquier componente de la biblioteca *Swing*, de forma que, en caso de que se requiera, se pueda utilizar cualquier componente con la interfaz gráfica que se desee.

Debido a lo anterior se puede decir que esta instancia funciona como un adaptador, sin embargo, esta no es una instancia del patrón *Adapter* de acuerdo a los criterios establecidos.

4.4 Resultado del análisis.

Después de analizar cada una de las instancias encontradas en ambos proyectos de los patrones seleccionados, se presentan los resultados generales del análisis anterior.

4.4.1 Observer.

En el caso del patrón *Observer*, de acuerdo a los resultados obtenidos se puede concluir que es un patrón fácil de encontrar ya que es comúnmente usado en los proyectos. Es fácil de implementar y su estructura es sencilla ya que las interrelaciones que posee son pocas. En este proyecto se encontraron aproximadamente 20 instancias de este patrón pero no se analizaron todas debido a que la mayoría de ellas son similares por lo que únicamente se seleccionaron las más representativas.

Otro aspecto que es importante destacar es que el patrón *Observer* es un patrón de gran utilidad para enviar mensajes a los componentes de software que estén interesados en dichos mensajes. Esta necesidad surge constantemente en ambos proyectos y es muy probable que esta necesidad surja también en la mayoría de los proyectos de software debido a que es común que los arquitectos o diseñadores de software se encuentren con la necesidad de enviar mensajes a determinados componentes de acuerdo a determinados eventos para que estos componentes realicen determinadas tareas.

Como se puede notar en el análisis anterior, todas las instancias encontradas fueron instancias válidas, lo cual indica que, primero, el patrón *Observer* es fácil de identificar, segundo, que los *Observers* son comúnmente llamados "*Listeners*" por lo que sólo basta con buscar las clases con esta palabra en su nombre para encontrar instancias de este patrón, en caso de que los *Observers* no sean llamados así deberán utilizarse otras técnicas que salen del alcance de este trabajo.

Se observó también que en muchos casos no es necesario tener *Subject* y *ConcreteSubject* sino únicamente *Subject*, ya que en ocasiones sólo se tiene un *Subject* en el cual se implementó todos los métodos necesarios para su funcionamiento.

Se observó también que el patrón *Observer* normalmente es útil cuando se requiere de comunicar un mismo mensaje u ocurrencia de un evento a ciertos componentes interesados en el, como se sabe, de acuerdo a la experiencia y de acuerdo a los expertos en el campo, este requerimiento es muy frecuente en los proyectos de desarrollo de software.

Otro aspecto que es importante destacar es el hecho de que se encontraron peculiaridades en algunas instancias, como por ejemplo, en la instancia *OBS_ListModel* existen cuatro clases que tienen el rol de *Subject* y *Observer* al mismo tiempo. Al principio de este proyecto se pensaba que una instancia que tuviera alguna clase en esta situación no podría ser una instancia válida pero, después del análisis de esta instancia, se llegó a la conclusión de que esto sí es válido debido a que el objetivo del patrón Observer y su funcionalidad se consideran adecuadas, además de que las instancias en esta situación (como la instancia *OBS_ListModel*) cumplieron con los criterios establecidos.

Otra instancia que es importante mencionar es la instancia *OBS_VetoableChange* en la cual, existen tres clases *Subject* en una misma instancia, esta situación no se esperaba al principio del proyecto por lo que este hecho se considera como una novedad en el patrón *Observer*.

4.4.2 Decorator.

A diferencia del patrón *Observer* (en el cual se encontró que todas las instancias fueron válidas), no todas las instancias encontradas del patrón *Decorator* resultaron instancias válidas de acuerdo a los criterios establecidos en el capítulo 4, lo cual indica que existen instancias que pueden parecer instancias del patrón *Decorator* pero no lo son. Como por ejemplo, existen clases que tienen la palabra "*decorator*" en su nombre; existen instancias que tienen una estructura similar a la del patrón pero el propósito de ésta es diferente del propósito del patrón.

En el caso de la instancia *DEC_decoratedFigure* se concluyó que es una instancia válida ya que cumple con todos los criterios establecidos para este patrón. Sin embargo, se llegó también a la conclusión de que la clase *DecoratedFigure* fue erróneamente nombrada, recuérdese que el rol de esta clase es el de *Decorator* por lo que la palabra "*decorated*" no es la mejor opción para esta clase. A pesar de esto, la estructura y el comportamiento de esta instancia fueron considerados como adecuados por lo que esta instancia se tomó como válida.

Otra instancia que resultó ser una instancia válida es la instancia *DEC_JComponent*, como se puede notar en los diagramas de clases de estas instancias la estructura es similar, en todos los decoradores (*DecoratedFigure* en la instancia *DEC_decoratedFigure* y *JFileChooser* y *JToolTip* en la instancia *DEC_JComponent*) se tiene un objeto del respectivo *Component*, así como sus métodos *set* y *get*. Además las interrelaciones de cada uno son similares también.

En cambio la instancia *DEC_LineDecoration* no resultó ser válida a pesar de que a primera vista aparentaba ser una instancia válida, ya que en un principio se tomó como instancia candidata del patrón *Decorator*. Inicialmente se escogió como candidata debido al nombre de la clase "*LineDecoration*" por lo que se pensó que esta clase bien podría ser un *Decorator*. Uno de los criterios más significativos es el criterio "b" (Tabla 5, Criterios de interrelación del patrón *Decorator*.) el cual no fue satisfecho en este caso ya que la clase *LineDecoration* no tiene ninguna relación con la interfaz *Figure*. Además es importante hacer notar que la estructura no es similar a la estructura del resto de las instancias, esta instancia podría ser más bien una instancia del patrón *Composite*, debido a la estructura, al nombre de la clase *CompositeLineDecoration* y a que se puede envolver más de un objeto con el método *addDecoration()* y el atributo *decorations* que es un objeto *ArrayList<LineDecoration>.*

El patrón *Decorator* no es un patrón que los autores de ambos proyectos hayan implementado constantemente como sucedió con el patrón *Observer*, únicamente se encontraron tres instancias de las cuales únicamente dos resultaron instancias válidas. Sin embargo, es un patrón que puede funcionar muy bien en los casos en los que se requiera que un mismo componente de software tenga diferente comportamiento dependiendo de necesidades específicas.

Además es importante notar que en el caso de la biblioteca *Swing*, su estructura y diseño permite que los usuarios puedan implementar fácil y prácticamente una instancia del patrón *Decorator*. Recuérdese que la clase principal de *Swing* es *JComponent* la cual es la base de todos los componentes que se pueden utilizar en *Swing*.

Todas las subclases pueden utilizarse como *ConcreteComponents*, únicamente se necesitaría crear una clase *Decorator* que herede de *JComponent*, que tenga un atributo *JComponent* y que tenga nuevo comportamiento según determinados requerimiento. Por ejemplo, supongáse que la clase *AbstractBorder* de *Swing* es una clase que hereda de *JComponent* y que posee un atributo de tipo *JComponent*, de esta forma, se puede tomar como una instancia del patrón *Decorator* válida, se puede valorar esta instancia más a detalle en la Figura 22, en el diagrama de clases:

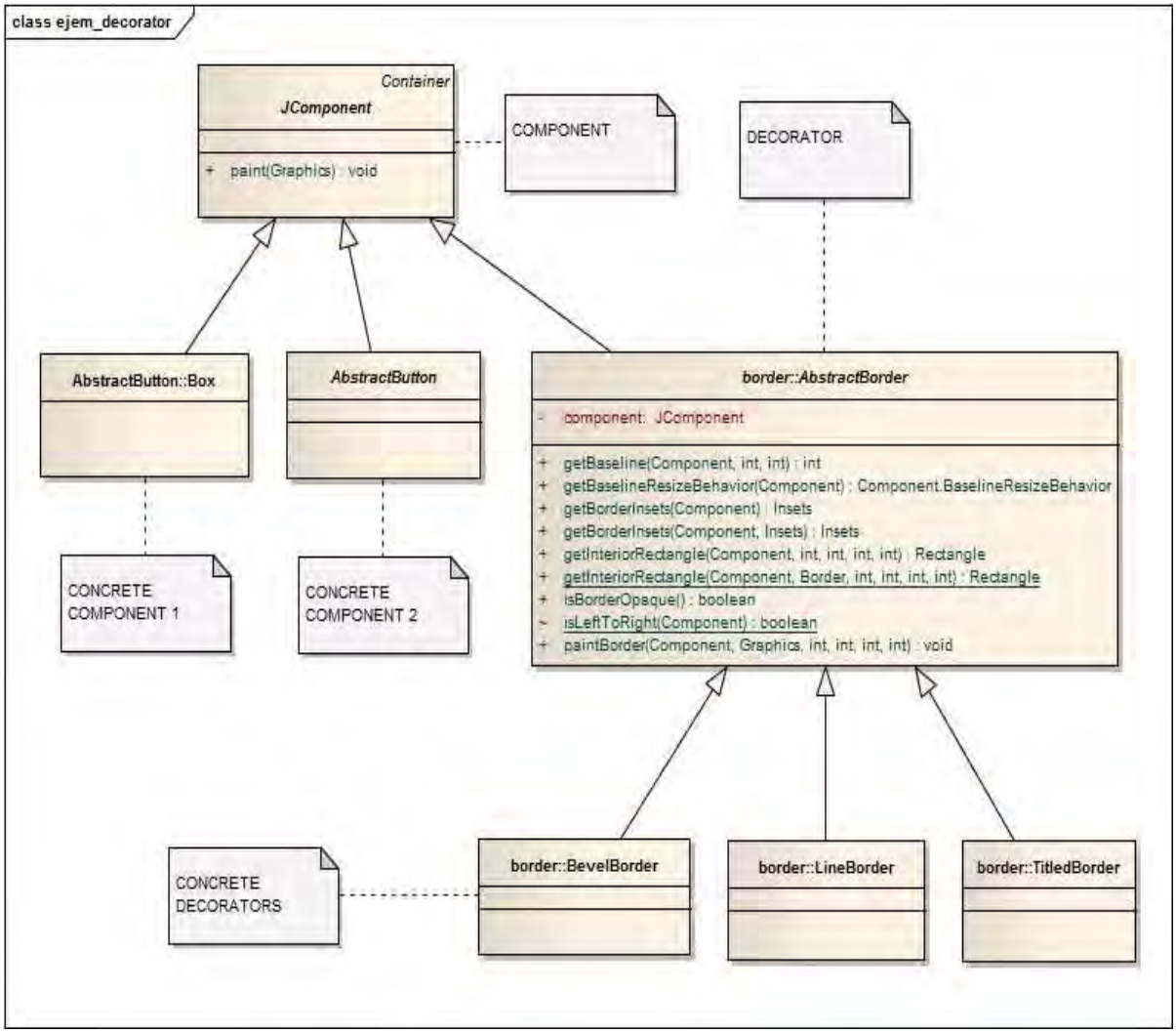

Figura 22, Diagrama de clases de una instancia ejemplo del patrón *Decorator*.

Como se puede observar en el ejemplo anterior, es fácil crear instancias del patrón *Decorator*, sin embargo, en la biblioteca *Swing* y *JHotDraw* no fue implementado con frecuencia este patrón.

4.4.3 Adapter.

En el caso del patrón *Adapter* se concluye que ninguna de las instancias son válidas en ninguno de los proyectos analizados. Esto indica que el patrón *Adapter* no es un patrón comúnmente usado en estos proyectos debido a que los autores consideraron innecesario su aplicación.

Recuérdese que el patrón *Adapter* es un patrón que permite la comunicación entre dos componentes de software que son incompatibles. Durante la búsqueda de instancias de este patrón se encontraron con clases que funcionan como adaptadores pero no son instancias del patrón *Adapter*; para tener un concepto claro de lo que es un adaptador se muestra a continuación la definición de "adaptar":

1. "*Acomodar, ajustar algo a otra cosa.*

2. *Hacer que un objeto o mecanismo desempeñe funciones distintas de aquellas para las que fue construido."* (Real Academia Española).

De acuerdo a las definiciones anteriores se puede decir que se encontraron instancias que funcionan como "adaptadores", es decir, instancias que permiten "ajustar" unas clases de forma que puedan comunicarse con otras clases que son incompatibles entre sí pero la estructura no es la estructura del patrón *Adapter*, como es el caso de la instancia *ADP_TextFigure* en la que las clases *NameAdapter* y *DurationAdapter* son adaptadores cuya función es la de obtener el nombre del modelo de un objeto *TaskFigure*, finalmente se concluyó que esta no es una instancia del patrón *Adapter* debido a que la estructura no es la propia de una instancia del patrón *Adapter*, además de que los objetivos son diferentes también. Otra instancia cuya estructura no es la de una instancia del patrón *Adapter* es la instancia *ADP_ComponentUI*, la cual es una instancia que fue creada para dar soporte a la arquitectura "*look and feel*" de Swing en la que cada uno de los componentes de esta biblioteca deberá de mostrarse con la interfaz gráfica adecuada independientemente del sistema operativo o plataforma en la que es utilizado, por lo que se necesita un "adaptador dinámico" para que cada componente se muestre y funcione adecuadamente.

A diferencia de las instancias mencionadas anteriormente, la estructura de las instancias *ADP_FigureSelectionListener* y *ADP_FigureListener* es la propia de la de una instancia del patrón *Adapter*, sin embargo, estas instancias no cumplieron con el último criterio, el cual establece que las tareas del *FigureSelectionListener, FigureListener* y *JComponent* deben tener objetivos similares, pero estas tareas se llevan a cabo de forma diferente en cada una de las clases, esto indica claramente que estas instancias no son adaptadores de ningún tipo y mucho menos una instancia del patrón *Adapter* a pesar de que tenga la estructura del patrón *Adapter*.

Es importante hacer notar también que el patrón *Adapter* sí fue implementado en *JHotDraw* pero en un nivel más alto, es decir, los autores diseñaron clases cuya función es la de permitir que estos proyectos puedan ejecutarse en diferentes plataformas. El rol cliente puede ser cualquier programa de usuario, a continuación se ilustrará el diagrama de clases del ejemplo anterior:

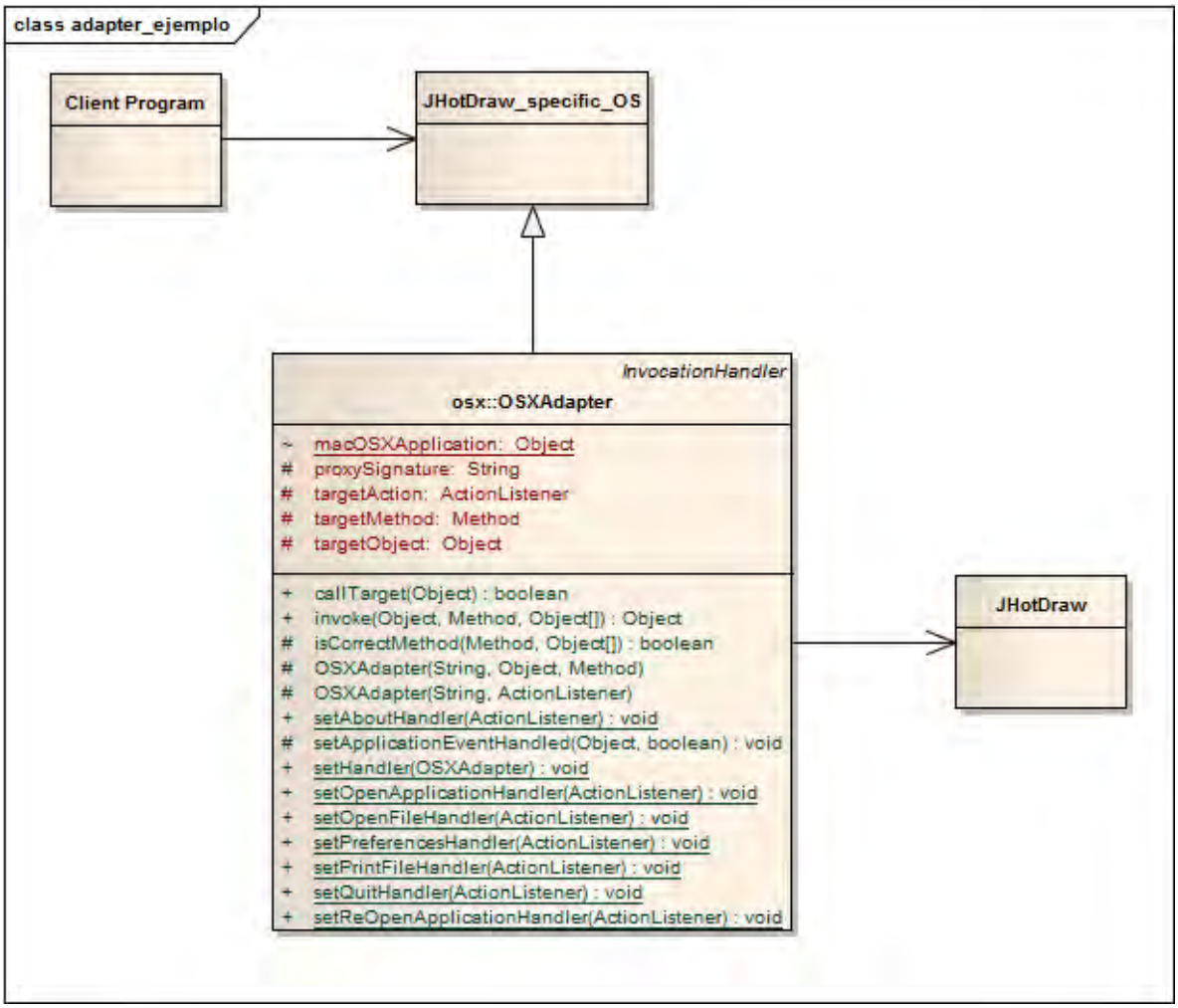

Figura 23, Diagrama de una instancia del patrón *Adapter* con la clase *OSXAdapter*.

En donde la clase *OSXAdapter* funciona como un adaptador que permite que cualquier aplicación de usuario pueda hacer uso de cualquier componente de *JHotDraw* en cualquier plataforma. A pesar de que aquí existe una clase que funciona como un adaptador, este diagrama de clases no es una instancia del patrón *Adapter* debido a que *JHotDraw* es una aplicación no una clase como tal, al igual que *Client Program* y *JHotDraw_specific_OS*.

Finalmente se concluye que este patrón no fue implementado en los proyectos analizados y es muy probable que en proyectos similares tampoco haya sido implementado, sin embargo se encontraron "adaptadores", es decir, estructuras de código que sirven para hacer posible la comunicación entre dos componentes de software incluso entre dos programas pero que no son instancias del patrón *Adapter*.

CONCLUSIONES GENERALES

En el presente trabajo se cumplió con todos los objetivos establecidos, se determinaron los criterios para cada patrón de diseño de acuerdo al análisis realizado y se determinó de acuerdo a estos si cada una de las instancias analizadas son válidas o no, y porque.

Se establecieron los criterios generales para detectar instancias los patrones de diseño: *Observer*, *Adapter* y *Decorator* en proyectos de código abierto donde se apliquen. Se comprobó que los patrones mencionados anteriormente están o no aplicados adecuadamente en los proyectos de código abierto de *JHotDraw* y *Swing* de acuerdo a los criterios establecidos. Se analizó la aplicación de los patrones de diseño seleccionados en los mismos proyectos, se verificó que estos patrones hayan sido aplicados adecuadamente de acuerdo a los criterios que se definieron en la misma investigación.

Finalmente se establecieron criterios sólidos (Capítulo 4) en los que se puedan basar otras personas para aplicar adecuadamente los patrones de diseño mencionados anteriormente y al mismo tiempo validar si los patrones son adecuadamente aplicados en proyectos actuales.

En el presente proyecto se realizó también una breve descripción de los patrones de diseño en general, se describieron algunos patrones de diseño: *Observer, Decorator y Adapter*. Además se describió brevemente los proyectos analizados, sus características y el por qué se escogieron, así como su estructura. Se realizó además, una breve descripción de las técnicas y herramientas empleadas para hacer posible la realización del análisis, se llegó a la conclusión de que la búsqueda de palabras clave fue una de las técnicas más efectivas ya que gracias a ésta se pudieron encontrar muchas instancias rápidamente, especialmente del patrón *Observer*. Se menciona también en qué forma se utilizó *Enterprise Architect* y como contribuyó éste en el análisis y desarrollo del proyecto.

Se presentaron los criterios para la validación de instancias de los patrones de diseño *Observer, Decorator* y *Adapter*. Asimismo se presentaron las instancias identificadas en los proyectos *JHotDraw* y *Swing*, en ellas se mostró el diagrama de clases de cada instancia, el análisis de cada una de ellas validando cada uno de los criterios contra las similitudes encontradas en las instancias.

Se pudo notar también, después del análisis realizado, que la aplicación, implementación y validación de los patrones de diseño no es una tarea fácil. Los patrones de diseño son una abstracción de una idea o concepto y, como se sabe, es casi imposible que dos personas puedan percibir de igual forma un mismo concepto, es por lo anterior que validar si un patrón de diseño fue implementado correctamente o no es una tarea difícil. En este proyecto no se pretende determinar la correctez de las instancias sino más bien se pretende validarlas de acuerdo a los criterios establecidos en el capítulo 4, es decir, se pretende determinar si cada uno de los criterios se cumplió en cada una de las instancias y a partir de esto se llegó a una conclusión: si es una instancia o si no es una instancia.

Además se pudo notar también que algunas instancias de los patrones de diseño analizados no cumplen con todos los criterios, esto indica que los patrones de diseño no fueron implementados adecuadamente o simplemente no son instancias como se pensaba en un principio. Por ejemplo,

la instancia *ADP_FigureListener*, la cual es una instancia del patrón *Adapter*, a pesar de que al principio parecía una instancia del patrón *Adapter*, finalmente se llegó a la conclusión de que no es una instancia válida. Por otro lado se encontraron instancias, sobre todo en el patrón *Observer*, con muy buenas implementaciones en ambos proyectos, instancias fáciles de identificar, analizar y comprender.

Es importante hacer notar también que, particularmente en el patrón *Decorator*, su estructura es estática, es decir, una vez que se diseña y crea una instancia del patrón *Decorator*, a partir de sus elementos estructurados se pueden crear diversos objetos con diversos comportamientos de acuerdo a determinados requerimientos.

A diferencia del patrón *Observer* y *Decorator*, el patrón *Adapter* puede apreciarse en un nivel más alto es decir, como se observó en el "Resultado del análisis" se puede notar adaptadores en la arquitectura de *JHotDraw* como un adaptador que permite que la aplicación pueda ejecutarse y programarse en diversas plataformas, sin embargo, no se encontró ninguna instancia válida de este patrón de diseño.

Se encontraron también algunas características en todas las instancias. Una de estas características es que se encontraron instancias que fueron diseñadas como tal, instancias de patrones de diseño, y por el contrario se encontraron instancias que no fueron diseñadas a propósito sino que fueron diseñadas como instancias de patrones de diseño por accidente o coincidencia.

En la introducción del presente trabajo se describió la metodología a seguir para el análisis de las instancias. Esta metodología fue llevada a cabo de manera satisfactoria ya que, como era de esperarse, el análisis de la segunda iteración se llevó a cabo en base a la primera. Como se describió anteriormente, se buscaron instancias del patrón *Observer*, se analizaron dichas instancias para así obtener la primera versión de los criterios para evaluar dichas instancias, a partir de este análisis se continuaron analizando las instancias del patrón *Observer* y se buscaron instancias del patrón *Decorator* y *Adapter*, los criterios se actualizaron conforme a las iteraciones que se fueron realizando a lo largo del proyecto. El Dr. Fuhrman (profesor de la *École de Thecnologie Superie*, Montreal, Canada; cotutor del presente trabajo), sugirió que los criterios podían clasificarse en criterios de estructura, de interrelación y otros, debido a que, de esta manera, el análisis y la comprensión de la misma podrían llevarse a cabo de manera más práctica y rápida por lo que los criterios finalmente quedaron divididos de esta manera. Por otra parte la M. en C. Guadalupe Ibargüengoitia sugirió que estos patrones de diseño eran adecuados para el proyecto debido a su reputación y utilidad.

La mayoría de los métodos fueron nombrados adecuadamente como en el caso del patrón *Observer* en el que el método "*registerObserver()"* normalmente se nombró como *add_Nombre_de_la_instancia_Listener()*; o el caso de los métodos con la palabra *"fire"* que dan la idea de disparar algo de acuerdo a la ocurrencia de un evento, estos métodos proporcionan una idea clara del comportamiento del método por lo que esto se considera una buena práctica en el desarrollo de software.

Como se sabe, los criterios se establecieron únicamente para tres patrones de diseño por lo que aun falta trabajo futuro en los patrones de diseño como por ejemplo evaluar otros patrones de diseño e incluso identificar instancias híbridas en las que se identifique comportamiento de dos o más patrones de diseño en una misma instancia.

Finalmente se puede concluir que los patrones de diseño analizados en el presente trabajo son muy útiles en el desarrollo de software ya que existen muchas probabilidades de que estos patrones sean de gran utilidad en proyectos de software. Sin embargo, es importante hacer notar que, si bien, los patrones de diseño aportan calidad, mantenibilidad y rapidez al desarrollo de software, existen instancias en las que los patrones de diseño no se implementaron con la estructura exacta, por lo que se consideraron como instancias no válidas, sin embargo, estas instancias fueron implementadas así porque los autores consideraron que la implementación modificada de la instancia era mejor que implementar exactamente el patrón de diseño.

Debido a que el desarrollo de software es un campo donde los expertos se pueden encontrar con infinidad de requerimientos de acuerdo a diversas situaciones, considero que la implementación de un patrón de diseño o no depende de la situación o requerimiento, por lo que el diseñador deberá tomar la decisión de cuál es la mejor opción en cada caso. La implementación de una instancia modificada podría ser una buena práctica de desarrollo en los casos en los que se requiera, todo depende de la situación que se tenga en cada caso por lo que no se puede afirmar que la implementación de un patrón de diseño sea la mejor opción o no todo depende de los requerimientos del proyecto.

Bibliografía

- Brown, W. J., Malveau, R. C., "Skips" McCormick, H. W., & Mowbray, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures and Projects in Crisis.* United States of America: John Wiley and Sons Inc.
- Buschmann, F. M. (1996). *Pattern-Oriented Software Architecture, A System of Patterns.* Inglaterra: Wiley.
- Fowler, M. (2000). The XP 2000 Conference. *Is design dead?* Sarnidia.
- Freeman, E., Freeman, E., & Bates, B. (2004). *Head First Design Patterns.* O'Reilly Media.
- Gamma, E. H. (1996). *Design Patterns Elements of Reusable Object-Oriented Software.* Addison Wesley.
- Horstmann, C. S. (2006). *Object-Oriented Design and Patterns.* John Willey and Sons.
- Infoworld Inc. (2011). *Become a programming Picasso with JHotDraw*. Retrieved mayo 22, 2011, from http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-jhotdraw.html
- Oracle Corporation. (2010). *A Swing Architecture Oveview*. Retrieved junio 02, 2011, from http://java.sun.com/products/jfc/tsc/articles/architecture/
- Oracle Corporation. (2010). *Package javax.swing*. Retrieved May 31, 2011, from http://download.oracle.com/javase/1.5.0/docs/api/javax/swing/package-summary.html
- Oracle Corporation. (2011). *Java SE JDK 6u25 Download*. Retrieved octubre 05, 2011, from http://www.oracle.com/technetwork/java/javase/downloads/jdk-6u25-download-346242.html
- Real Academia Española. (n.d.). *rae*. Retrieved septiembre 30, 2011, from http://buscon.rae.es
- Sparx Systems Pty Ltd. (2007). *Características de la herramienta de diseño UML Enterprise Architect*. Retrieved septiembre 20, 2011, from http://www.sparxsystems.com.ar/products/ea_features.html
- Steling, S., & Maassen, O. (2001). *Applied java patterns.* Prentice Hall.
- Thomas Eggenschwiler, E. G. (2007, 07 11). *JHotDraw as Open-Source Project*. Retrieved 01 04, 2011, from http://www.jhotdraw.org/
- Tutorials, F. J. (2009). *Java Swing Free Java tutorials*. Retrieved May 31, 2011, from http://www.freejavaguide.com/java_swing.html