UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

APLICACIÓN DE PATRONES DE DISEÑO DE SOFTWARE EN EL DISEÑO DE UNA APLICACIÓN DE BASE DE DATOS ORIENTADA A OBJETOS.

TRABAJO DE GRADUACIÓN

PRESENTADO A JUNTA DIRECTIVA DE LA

FACULTAD DE INGENIERÍA

POR

LUIS FERNANDO ALONZO JERONIMO

ASESORADO POR ING. JOSE RICARDO MORALES PRADO

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, AGOSTO DE 2005

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Murphy Olympo Paiz Recinos
VOCAL I	
VOCAL II	Lic. Amahán Sánchez Álvarez
VOCAL III	Ing. Julio David Galicia Celada
VOCAL IV	Br. Kenneth Issur Estrada Ruiz
VOCAL V	Br. Elisa Yazminda Vides Leiva
SECRETARIA	Inga. Marcia Ivonne Véliz Vargas

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Sydney Alexander Samuels Milson
EXAMINADOR	Inga. Virginia Victoria Tala Ayerdi
EXAMINADOR	Inga. Ligia Maria Pimentel Castañeda
EXAMINADOR	Ing. Edgar Estuardo Santos Sutuj
SECRETARIO	Ing. Pedro Antonio Aguilar Polanco

HONORABLE TRIBUNAL EXAMINADOR

Cumpliendo con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

APLICACIÓN DE PATRONES DE DISEÑO DE SOFTWARE EN EL DISEÑO DE UNA APLICACIÓN DE BASE DE DATOS ORIENTADA A OBJETOS

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas en febrero de 2004.

Luis Fernando Alonzo Jeronimo

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

GLOSARIO

Clase abstracta Clase cuyo propósito principal es definir una interfaz, delegando la mayoría o toda su implementación a sus subclases. No puede ser instanciada.

Clase cliente Clase que no forma parte de un patrón pero que interactúa con las clases que este define.

Clase concreta Clase que implementa todos los métodos definidos y no posee métodos abstractos y si puede ser instanciada.

Composición de objetos Método alternativo a la herencia para obtener nueva funcionalidad ensamblando objetos existentes, conocido como *black box reuse* o reutilización de caja negra.

Delegación Forma para implementar la composición de objetos, en la cual un objeto delega responsabilidades a otro objeto y se pasa como parámetro a sí mismo para que el objeto que recibe las responsabilidades pueda hacer referencia al objeto delegador.

Diagrama de clases Representación gráfica de las clases, sus atributos métodos y relaciones estáticas entre ellas.

Diagrama de interacción	Representación gráfica del orden de ejecución de las peticiones o paso de mensajes entre objetos.
Diagrama de objetos	Representación gráfica de las instancias de las clases en un momento específico en tiempo de corrida.
Diagrama de secuencias	Tipo de diagrama de interacción en el cual el tiempo se mide de arriba hacia abajo y una línea vertical indica el tiempo de vida del objeto.
Firma de la operación	También conocida como firma del método es la definición del método, su nombre, atributos y valor de retorno.
Framework	También conocido como armazón es un conjunto de clases relacionadas que describen una estructura de *software* ampliable y reutilizable.
Herencia	Forma de obtener o extender la funcionalidad de una clase existente en una nueva clase. También conocida como *white box reuse* o reutilización de caja blanca.
Método abstracto	Método u operación declarada por una clase pero no es implementado.

Método concreto	Método u operación declarada e implementada por una clase.
Toolkit	Conjunto de clases que proveen una funcionalidad general, y no definen el diseño de una aplicación.

RESUMEN

Actualmente, muchos desarrolladores de *software* se enfrentan con problemas frecuentes que, por no haber un documento que logre unificar soluciones existentes, deben ingeniarse cada uno de ellos(as) una solución, que posiblemente no sea la óptima, y se incurran en costos innecesarios como tiempo y recursos económicos.

Con el presente trabajo de graduación se pretende unificar los patrones de diseño de *software* existentes al desarrollo de una aplicación real, para que los diseñadores tengan una visión general de la aplicación de estos patrones. Por lo tanto, los beneficiados del presente trabajo son las personas que están relacionadas en el proceso de diseño y desarrollo de una nueva aplicación (desarrolladores, DBA's, analistas, entre otras), que utilizan la metodología de objetos, ya que se obtendrán mayores beneficios al aplicar soluciones existentes, que son resultado de la experiencia de otros desarrolladores y expertos en el área, que han sido probadas y realmente funcionan. Por lo tanto, los patrones son soluciones resultado de la experiencia y que nos ayudan a no cometer los mismos errores.

En el presente trabajo se asumirá que el lector tiene conocimientos mínimos de la programación orientada a objetos, ya que no se pretende profundizar en este aspecto. El conocimiento de los diagramas UML también será necesario, ya que se dará una breve explicación de los diagramas utilizados; pero es necesario, para mayor comprensión del lector el conocimiento de este lenguaje.

El manejador de base de datos a utilizar es Oracle 9i, por su capacidad de manejar algunos conceptos de metodología orientada a objetos, pero debe quedar claro que no es una base de datos totalmente orientada a objetos. Este trabajo está enfocado, principalmente, en el proceso de desarrollo de la aplicación, empleando los patrones de diseño de *software* aplicables, que existen actualmente. El lenguaje por utilizar en la presentación y ejemplificación de los patrones es Java.

OBJETIVOS

General

Dar a conocer el concepto de patrones de diseño de *software* y los distintos patrones existentes en la actualidad, debido a que no se le ha dado suficiente divulgación, para su aprovechamiento y utilización. Además se pretende aplicar dichos patrones en el diseño de una aplicación para alcanzar la integración de los conceptos de patrones de diseño de *software* y el proceso de análisis y diseño de una aplicación real.

Específicos

1. Definir los conceptos principales referentes a los patrones de diseño de *software*, reseña histórica, clasificación, objetivos, ventajas y desventajas.
2. Identificar los patrones de diseño disponibles actualmente según su clasificación, describir aspectos teóricos, prácticos y su forma de uso.
3. Definir la aplicación a realizar para ejemplificar el uso de los patrones de diseño de *software*, mediante la metodología de proceso unificado *Rational*.
4. Documentar las fases del proceso de análisis y diseño de la aplicación, para identificar los puntos exactos en donde se utilizarán los patrones de diseño de *software*; para identificar el beneficio derivado de la utilización de los patrones de diseño de *software* en el proceso de diseño y desarrollo de una aplicación real, como también los aspectos en que se presentaron deficiencias o se debe mejorar.

INTRODUCCIÓN

El presente trabajo de graduación pretende dar a conocer los aspectos teóricos y prácticos de los patrones de diseño de *software*, para aplicarlos en el proceso de análisis y diseño de una aplicación de base de datos orientada a objetos. Los patrones de diseño de *software* describen soluciones simples y elegantes para problemas específicos en el diseño de *software* orientado a objetos.

Un patrón de diseño describe una solución a un problema recurrente o frecuente durante el proceso de análisis y diseño de *software* orientado a objetos. Este nos indica la forma de crear *software*, basados en la metodología de desarrollo del mismo orientado a objetos, de forma conocida, manejando correctamente las clases y objetos definidos. Los patrones de diseño tienen una serie de características, entre las cuales se pueden mencionar: son soluciones técnicas, concretas; se utilizan en situaciones frecuentes, favorecen la reutilización de código, entre otras.

El diseño de *software* orientado a objetos no es una tarea fácil, y realizar el diseño de un *software* orientado a objetos que además sea reutilizable es aún menos fácil, ya que el diseño debe ser específico al problema que tratamos de resolver actualmente, también debe ser general para reutilizarse en futuros situaciones y problemas. Realizar este tipo de diseño puede tomar un poco más de trabajo y esfuerzo, pero este esfuerzo extra indudablemente será recompensado en el incremento de la flexibilidad y reusabilidad de nuestros diseños y los patrones de diseño son los encargados de ayudarnos a lograrlos.

Esta trabajo se encuentra dividida en tres capítulos, de los cuales se puede omitir la lectura del capítulo 1, si el lector ya conoce lo que son los patrones de diseño de *software*. Cabe mencionar que en el presente trabajo se asume que el lector tiene previo conocimiento de la metodología de objetos. En el primer capítulo se introduce al lector en el concepto de los patrones de diseño de *software*, es decir, su historia, definición, forma de utilizarlos y organizarlos, los problemas que pretenden resolver, entre otros. El capítulo dos presenta una lista de patrones de diseño de *software*, los cuales se encuentran organizados de acuerdo con su propósito. Se eligió esta forma de presentarlos en el presente trabajo ya que es una de las formas de organización más comunes y fáciles de entender, pero existen distintas formas de organizar los patrones de diseño además de ésta, por ejemplo, se pueden organizar de acuerdo con su alcance (a nivel de clases o a nivel de objetos).

En el capítulo tres se aplican algunos los patrones de diseño de *software*. En este capítulo es conveniente que el lector tenga conocimientos sobre diagramas de UML, pero no es estrictamente necesario, ya que se explicarán de forma breve los diagramas utilizados. Se define la aplicación que se diseñará, y la toma de requerimientos. Seguidamente, se desarrolla la etapa de análisis y diseño del sistema.

Actualmente, en nuestro país no se ha difundido lo suficiente el concepto de patrones de diseño, por lo que este trabajo pretende dar a conocer a los desarrolladores y analistas de sistemas que utilizan la metodología de objetos las soluciones que ofrecen los patrones de diseño, y lograr el desarrollo de *software* flexible y reutilizable.

1. INTRODUCCIÓN A LOS PATRONES DE DISEÑO DE *SOFTWARE*

1.1 Introducción

Los patrones de diseño indican la forma de construir *software* flexible y reutilizable, de cómo utilizar de manera eficiente las clases y objetos. La idea de patrones de *software* tuvo su origen del campo de la arquitectura. Christopher Alexander, un arquitecto, escribió dos libros revolucionarios que describían patrones en arquitectura de construcción y planificación urbana: *A Pattern Language: Towns, Buildings, Construction* (Oxford University Press, 1977) y *The Timeless Way of Building* (Oxford University Press, 1979). Las ideas presentadas en estos libros son aplicables a varios campos además de la arquitectura, incluyendo el *software*.

En 1987, Ward Cunningham y Kent Beck usaron algunas de las ideas de Alexander y desarrollaron cinco patrones para el diseño de interfaces de usuario (UI), que publicaron en un artículo de OOPSLA'87 llamado *Using Pattern Languages for Object-Oriented Programs*. Posteriormente en 1991 Jim Coplien publica el libro *Advanced C++ Programming Styles and Idioms* donde realiza un catálogo de una especie de patrones. En 1994 se publicó uno de los libros más influyentes de la década del 90: *Design patterns*[1], también llamado *Gang of Four* (o GoF), el cual popularizo la idea de los patrones.

1.2 Definición de patrón

"Cada patrón describe un problema que ocurre una y otra vez en nuestro ambiente, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser utilizada más de un millón de veces sin hacerlo siquiera dos veces de la misma manera"[2].

Cuando Christopher Alexander dio esta definición, estaba hablando de patrones en la construcción de edificios y torres, pero esta definición es igualmente valida para los patrones de diseño orientado a objetos, ya que nuestras soluciones están expresadas en términos de objetos e interfaces, en vez de paredes y puertas.

1.3 Tipos de patrones

Existen distintos tipos de patrones, como por ejemplo, patrones de programación concurrente, de interfaz gráfica, de organización de código, de optimización de código, etc. A continuación se detallaran algunos tipos de patrones para mostrar que existen más patrones además de los patrones de diseño.

Patrones de arquitectura

Estos patrones indican formas de descomponer, conectar y relacionar sistemas. Estos se ubican en un nivel de abstracción mayor que el de los patrones de diseño. Entre ellos se puede mencionar *pipes-filters*, abstracción de datos y organización orientada a objetos, invocación implícita basada en eventos, sistemas estratificados o en capas (*layered systems*), entre otros[3].

Patrones de programación

Son patrones de bajo nivel sobre un lenguaje de programación específico, los cuales describen como implementar cuestiones concretas; por ejemplo, al utilizar el patrón *Prototype*[4], se necesita implementar el método u operación clonar() en la clase prototipo. La implementación de este método cambiara dependiendo del lenguaje que estemos utilizando, debido a las características propias de cada lenguaje de programación, por ejemplo, Smalltalk provee la implementación de un método c*opy()*, que es heredado a todas las subclases del tipo *Object;* mientras que en Java, provee la implementación de un método *clone()* heredado de todas las subclases de *Object* similar que en SmallTalk, pero este método es declarado protegido dentro de la clase *Object*, por lo que puede ser llamado únicamente desde un método definido por una clase que implemente la interfaz *Cloneable* o bien ser explícitamente sobrescrito por esa clase para que sea público[5]. Como se puede observar, la implementación de una operación, método o cualquier requerimiento específico puede variar drásticamente de un lenguaje de programación a otro.

Patrones de análisis

Estos patrones definen reglas que permiten modelar un sistema satisfactoriamente y ayudan a elegir las clases adecuadas y la forma como estas deben interactuar. Estos patrones también son conocidos como GRASP (*General responsibility assignment software patterns*). Los patrones de análisis son menos concretos que los patrones de diseño. Entre estos podemos mencionar: bajo acoplamiento, alta cohesión, experto, indirección, etc.

Patrones organizacionales

Son patrones que describen la forma de organizar grupos humanos, generalmente relacionados con el *software*. El dominio de los patrones de diseño abarca el diseño de *software*, mientras que el dominio de los patrones organizacionales es la administración del *software* y sus procesos. Un cambio introducido con los patrones de diseño puede proporcionar una retroalimentación en cuestión de días, mientras que para los patrones organizacionales puede tomar meses o años. Entre los patrones organizacionales podemos mencionar: *scrifice one person*, *daycare*, *firewalls*, y *gatekeeper*[6].

Patrones de proceso

Son una colección de técnicas generales, acciones, tareas o actividades para el desarrollo de *software* orientado a objetos. Se identifican 3 tipos de patrones de proceso principales: *task process*, *stage process* y *phase process*[7].

1.4 Patrones de diseño de *software*

1.4.1 Definición

Los desarrolladores expertos en la metodología orientada a objetos están de acuerdo en que lograr que un diseño sea flexible y reutilizable a la primera vez es casi imposible. Antes de terminar el diseño, este ha de ser probado y modificado varias veces. Una técnica que los expertos utilizan es, no tratar de resolver cada problema sobre la base de los mismos principios, por el contrario, reutilizar soluciones en las que se ha trabajado anteriormente.

Aprovechar la experiencia adquirida directa o indirectamente es de lo que tratan los patrones. Los patrones resuelven problemas de diseño específicos y hacen diseños orientados a objetos más flexibles, elegantes y reutilizables. Un diseñador que se encuentra familiarizado con los distintos patrones, puede aplicarlos de inmediato a algún problema de diseño, en vez del mismo descubrir la solución. Un patrón está compuesto principalmente por los siguientes cuatro elementos:

- El nombre del patrón describe el problema de diseño, su solución y consecuencias en una o dos palabras. Nombrar los patrones incrementa nuestro vocabulario de patrones. Un vocabulario de patrones nos permite mejorar la comunicación con colegas o en la documentación.
- El problema; este describe cuando aplicar el patrón. Explica el problema y su contexto. Algunas veces se incluye una lista de condiciones que deben cumplirse antes de que tenga sentido aplicar el patrón.
- La solución. Se describen los elementos que conforman el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño específico o su implementación, ya que un patrón es como una plantilla que puede aplicarse en distintas situaciones. El patrón provee una descripción abstracta del problema de diseño y como un arreglo general de clases y objetos lo resuelven.
- Las consecuencias son el resultado e implicaciones del uso del patrón. Entre estas podemos mencionar el lenguaje y su implementación, su impacto en la flexibilidad, extensibilidad y portabilidad del sistema.

Un patrón nombra, abstrae e identifica los aspectos clave de la estructura de un diseño común, el cual es útil para crear un diseño orientado a objetos reutilizable. Un patrón identifica las clases participantes y sus instancias, los roles y colaboraciones, y la distribución de responsabilidades. Cada patrón de diseño se enfoca en un problema o situación específica. A continuación se presentan las características de los patrones de diseño.

1.4.2 Características

- Los patrones de diseño son soluciones concretas, es decir, proponen soluciones a problemas específicos, no son teorías genéricas.
- Son soluciones técnicas, ya que indican soluciones basados en la programación orientada a objetos. En algunas situaciones tienen más utilidad con algunos lenguajes de programación que con otros, debido a las características propias de cada lenguaje[8].
- Se utilizan en situaciones frecuentes y favorecen la reutilización de código. Ayudan a construir clases reutilizables.
- El uso de un patrón no se refleja en el código. Al aplicar un patrón, el código resultante no debe reflejar el patrón o patrones que lo inspiro.
- Es difícil reutilizar la implementación de un patrón. Al aplicar un patrón, aparecen clases específicas que dan solución a un problema específico, y que no serán aplicables a otros problemas que requieran el mismo patrón.

1.4.3 Clasificación

Debido al gran número de patrones de diseño existentes y a su diversidad en granularidad y nivel de abstracción, se necesita una forma de organizarlos. Una correcta clasificación y organización de los patrones

permitirá al lector aprenderlos y comprenderlos de una mejor manera. La clasificación que propone el libro GoF[9] se basa en uno de dos criterios.

El primero se basa en el propósito del patrón y refleja lo que el patrón hace. El segundo criterio se basa en el alcance del patrón, es decir, en que se aplica principalmente el patrón, en las clases o en los objetos directamente. Los patrones pueden tener cualquiera de los siguientes propósitos:

- **Creación**. Los patrones de creación se enfocan en el proceso de creación o instanciación de los objetos.
- **Estructura**. Los patrones estructurales manejan la composición de clases u objetos.
- **Comportamiento**. Estos patrones indican las formas en que las clases y objetos interactúan y distribuyen responsabilidades.

Según el segundo criterio mencionado, los patrones de clase manejan las relaciones entre clases y subclases. Estas relaciones son establecidas a través de herencia, es decir de forma estática, resueltas en tiempo de compilación. Los patrones de objeto manejan las relaciones entre objetos, las cuales pueden cambiar en tiempo de corrida, y son más dinámicas que las relaciones de clases.

Los patrones estructurales de clase utilizan herencia para componer otras clases, mientras que los patrones estructurales de objeto describen formas de ensamblar objetos. Los patrones de comportamiento de clase utilizan herencia para describir algoritmos y flujo de control, mientras que los patrones de comportamiento de objeto describen como un grupo de objetos cooperan para realizar una tarea que un objeto no puede desempeñar solo.

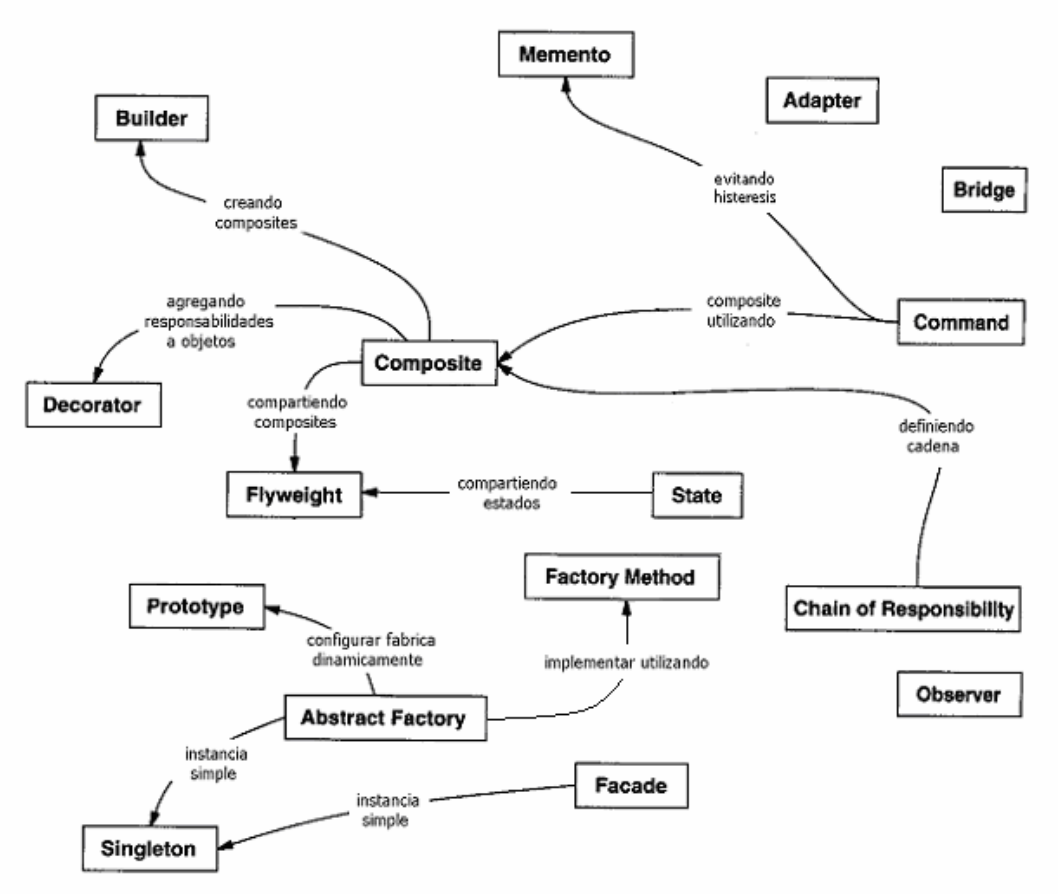
A continuación se presenta una tabla que representa la forma como se han clasificado los patrones de diseño que se detallarán más adelante:

Figura 1. Clasificación de patrones de diseño

		PROPOSITO		
		Creación	Estructura	Comportamiento
ALCANCE	Clase	*Factory Method*	*Adapter*(clase)	
	Objeto	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Adapter*(objeto) *Bridge* *Composite* *Decorator* Façade *Flyweight*	*Chain of responsibility* *Command* *Memento* *Observer* *State*

Existen otras formas de clasificar los patrones de diseño, como por ejemplo, organizarlos de acuerdo a como se hacen referencia entre ellos (ver la figura 2), su forma de uso o cuando se aplican.

Figura 2. Relaciones de los patrones de diseño



1.5 Formato de un patrón de diseño de *software*

Para describir un patrón, no basta con su representación gráfica. Debe existir un documento en el cual se capture el producto final del proceso de desarrollo y las relaciones entre clases y objetos. El libro GoF propone una plantilla donde se presenta la información acerca del patrón en una forma estructurada y clara, para hacer de los patrones de diseño algo fácil de aprender, recordar, compara y reutilizar. La plantilla está compuesta de la siguiente manera:

a) Nombre del patrón. El nombre del patrón debe comunicar la esencia del patrón de una forma clara y en pocas palabras. Un buen nombre es vital porque ese nombre formara parte del vocabulario de patrones.

b) Intención. En esta parte se indica que hace el patrón, sus razones de ser, que pretende, y a que problema en particular esta enfocado.

c) Conocido con, otros nombres con los cuales se conoce el patrón, si los hubiera.

d) Motivación. Indica los motivos por los cuales se considera como un patrón. Incluye un escenario que ilustre el problema de diseño y como las clases y estructuras de objetos en el patrón resuelven el problema. El escenario sirve para ayudar al lector a entender la abstracción utilizada para definir el patrón.

e) Aplicabilidad. Indica las situaciones en las que el patrón puede ser aplicado. Debe incluir algunos ejemplos de diseños pobres en los cuales el patrón puede ser aplicado y la forma de reconocer las situaciones para aplicar el patrón.

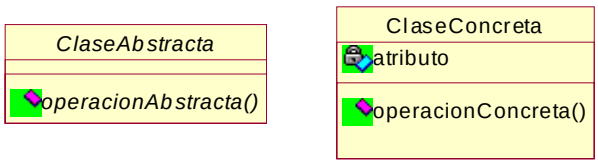
f) Estructura. Es una representación gráfica de las clases incluidas en el patrón a través de un diagrama de clases. También se incluye diagramas de interacción para ilustrar la secuencia de peticiones y colaboraciones entre objetos. Debido a la importancia de la comprensión de los diagramas presentados específicamente en esta sección de la plantilla de los patrones y a lo largo del presente trabajo, a continuación se describe de cada uno de los diagramas utilizados.

Diagrama de clases

En este diagrama se presentan las clases de una forma gráfica con su estructura y las relaciones estáticas entre ellas. Una clase es representada por una caja con el nombre de la clase en el borde superior del cuadrado. Los atributos de la clase se escriben después del nombre de la clase y los métodos u operaciones de la clase después de los atributos.

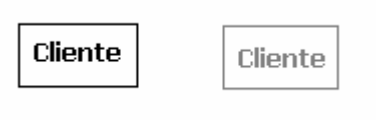
El tipo de la operación (valor de retorno) o de un atributo es opcional, y se coloca antes del nombre de la operación o del atributo de la clase. Las clases o las operaciones abstractas se representan con cursiva, como se muestra en la siguiente figura:

Figura 3. Representación de clases abstractas y ordinarias



En algunos patrones de diseño es de utilidad saber donde las clases cliente hacen referencia a las clases participantes del patrón. Cuando un patrón incluye una clase cliente como uno de sus participantes, es decir, que el cliente tiene responsabilidades en el patrón, el cliente aparece como una clase ordinaria. Cuando el patrón no incluye un cliente como participante (el cliente no tiene responsabilidades en el patrón), pero el incluirlo ayuda a dejar más claro la forma como las clases participantes interactúan con las clases clientes; entonces, la clase cliente debe representarse en color gris. A continuación se presenta un ejemplo:

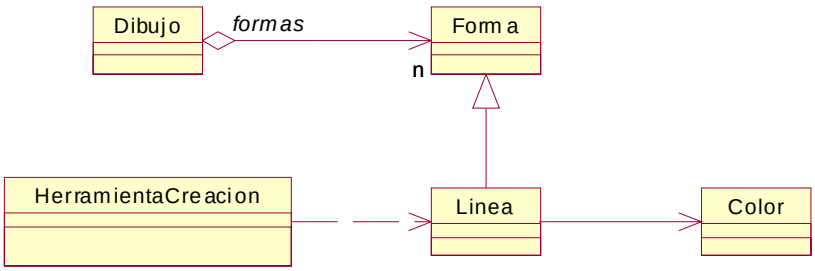
Figura 4. Clase cliente participante y clase cliente no participante



Para representar la herencia se utiliza un triangulo conectando la subclase hacia la clase padre. En las referencias de objeto, las relaciones parte de o de agregación se representan con una flecha con un diamante en la base. La flecha apunta hacia la clase que esta siendo agregada o formando parte de la clase. Una flecha sin el diamante en la base denota que el objeto mantiene una relación con el objeto agregado que otros objetos comparten. (Como en la figura 5, la línea tiene una referencia hacia el objeto color, y otros objetos también tendrán una referencia hacia ese mismo objeto). Se puede incluir un nombre para la referencia cerca de la base para diferenciar las distintas relaciones.

Algo también importante de describir gráficamente es que clases son instanciadas con otras lo cual, se representara con una flecha punteada apuntando hacia la clase que es instanciada o creada. Esta relación también es llamada de instanciación o dependencia, pues la dependencia entre ambas clases, es decir, un cambio en una clase afectara a la otra clase relacionada. Como se puede ver en la figura 5, la clase HerramientaCreacion crea una instancia de la clase línea. La multiplicidad en las relaciones se representara de la siguiente manera: n muchos, 1 exactamente uno, o un rango específico. La multiplicidad nos sirve para determinar si una relación es obligatoria u opcional y para conocer el número máximo y mínimo de las instancias a crear. La figura 5 muestra que múltiples objetos forma serán parte de dibujo.

Figura 5. Tipos de relaciones



Y por último, definimos anotaciones en pseudo código para definir formas de implementar las operaciones, como se muestra en la figura 6.

Figura 6. Notas de implementación

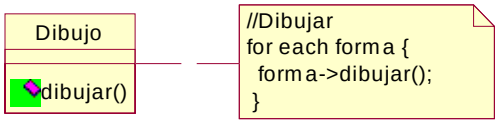


Diagrama de objetos

Los diagramas de objetos representan instancias de las clases, exclusivamente. Proveen un *snapshot* o instantánea de los objetos en un patrón de diseño. Los objetos son nombrados como *aClase*, donde *Clase* es la clase del objeto. El símbolo a utilizar para representar los objetos es una caja redondeada en los bordes, con una línea separando el nombre del objeto de las referencias a otros objetos.

Figura 7. Diagrama de Objetos

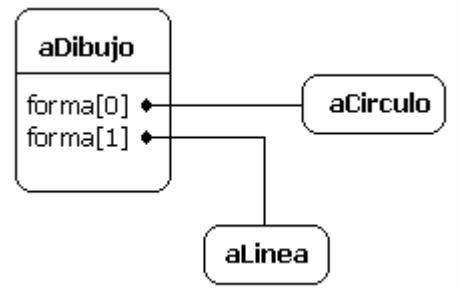
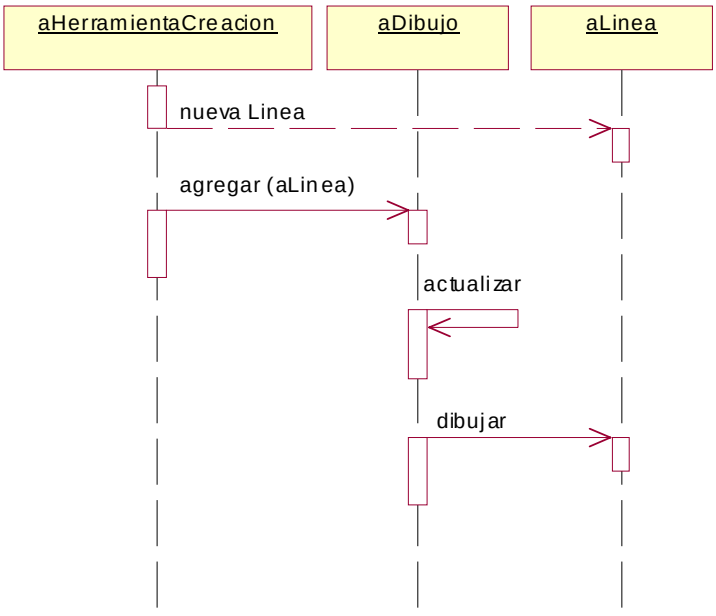


Diagrama de interacción

Estos diagramas muestran el orden de ejecución de las peticiones entre objetos. Se utilizará el diagrama de secuencias para representar las interacciones de los objetos. El tiempo se mide de arriba hacia abajo. Una línea vertical indica el tiempo de vida de un objeto. Para nombrar los objetos se utiliza la misma notación que en el diagrama de objetos (*aClase*). Una línea vertical punteada indica que el objeto aun no ha sido creado. Un rectángulo vertical indica que el objeto esta activo, es decir, que se encuentra manejando una petición de otro objeto. La operación puede enviar peticiones o mensajes a otros objetos, estos son representados con una línea horizontal apuntado al objeto receptor del mensaje. El nombre de la petición o mensaje enviado debe ser escrito sobre la flecha. Una petición para crear un objeto es representado con una línea horizontal punteada apuntando hacia el objeto a crear.

Figura 8. Diagrama de interacción



g) Participantes. Las clases y objetos que participan en el patrón de diseño, con sus responsabilidades.

h) Colaboraciones. Describe gráficamente mediante un diagrama de interacción o simplemente de manera textual la forma como los participantes colaboran para resolver sus tareas o responsabilidades.

i) Consecuencias. Indica como el patrón logra sus objetivos, las consecuencias y los resultados de utilizarlo. Los aspectos que la estructura del sistema que se puede cambiar independientemente. Al final, lo que se busca es indicar los beneficios y riesgos que puede ocasionar el uso del patrón.

j) Implementación. Información detallada sobre la forma de implementar el patrón, las decisiones de diseño y las formas de codificarlo sin tomar en cuenta la plataforma o lenguaje específico en el que se implementará. Se debe incluir riesgos, ideas, y técnicas que se debe tomar en cuenta al implementar el patrón.

k) Muestra de código y utilización. Un fragmento de código que ilustre la forma de implementar el patrón.

l) Usos conocidos. Ejemplos en los que el patrón se haya aplicado en productos y sistemas conocidos para facilitar la comprensión del lector.

m) Patrones relacionados. Indica los patrones que se encuentran relacionados de manera cercana. También se indica los patrones que deben funcionar como alternativa o de forma complementaria.

1.6 Antipatrones

Si los patrones nos ofrecen una forma de resolver un problema típico, los antipatrones nos hablan de actitudes o formas de enfrentarse a los problemas con consecuencias negativas conocidas, de experiencias que han arruinado otros proyectos. La fuerza de los antipatrones se basa en que puede ser más fácil de detectar a primeramente las malas líneas en el desarrollo de *software* que elegir el camino correcto, o dicho de otra forma, descartar las alternativas incorrectas nos puede ayudar en la elección de la alternativa mejor. Se clasifican en antipatrones de desarrollo, de arquitectura del *software* y de gestión de proyectos. A continuación se presentan ejemplos en cada una de las categorías.

1.6.1 Desarrollo de *software*

- *Lava flow* (código muerto). Cuando se continua incluyendo código en la aplicación que ya no se utiliza, como continuar incluyendo código para Win16 cuando la aplicación ya solo funciona para 32 bits. Se mantiene código que se supone importante aunque se desconoce si se seguirá utilizando. Un exceso de rotación de personal unida a una documentación deficiente puede provocar esta situación. La solución pasa por mantener actualizada la documentación y por depurar el código obsoleto. Java facilita este mantenimiento al disponerse de la cláusula *deprecated* para mantener el código obsoleto solo por un tiempo.
- *Functional Decomposition*. Diseño no orientado a objetos.
- *Poltergeists*. No se sabe lo que hacen algunas clases.
- *Spaghetti Code*. Muchos *if* o *switch*.
- *Cut-and-paste programming*. Cortar y pegar código.

1.6.2 Arquitectura de *software*

- ***Stovepipe enterprise*:** aislamiento en la empresa, islas de automatización. Se produce cuando en la empresa se desarrollan sistemas de manera independiente. Esto dificulta la interconexión de sistemas y la reutilización, lo cual incrementa los costes de desarrollo, se crean islas de automatización, y provoca que existan tecnologías incompatibles, arquitecturas monolíticas y falta de documentación. Se dificulta la ampliación de los sistemas, suelen faltar estándares. La causa suele estar en una falta de estrategia tecnológica en la empresa, falta de cooperación y comunicación entre departamentos y niveles, deficiencias en el conocimiento de la tecnología, en definitiva ausencia de un objetivo claro referente a el uso de tecnología en la empresa.

- ***Stovepipe system*:** sistema heredado, aislamiento entre sistemas. De manera semejante al aislamiento en la empresa, pero aplicado a uno de sus sistemas, se produce por un desarrollo mal planificado o que carece de planificación, en el que sus subsistemas parecen haber sido desarrollados por equipos distintos sin comunicación y coordinación, los diferentes subsistemas se conectan unos a otros directamente, por lo que aparecen fuertes dependencias (alto acoplamiento entre clases). Al ampliarse el sistema aumenta la interdependencia y la complejidad con lo que el mantenimiento llega a ser insostenible y se plantea la sustitución completa. Suele producirse un distanciamiento entre "documentación y desarrollo". Los diseñadores ignoran los detalles de integración. Se suele producir al superarse el presupuesto para el desarrollo y se posponen las fechas de entrega, para "llegar a tiempo" se desarrolla de manera apresurada sin actualizar la documentación. Los cambios

requieren fuertes esfuerzos y se recurre a parches provisionales. La causa puede ser la utilización de múltiples formas de integración de los subsistemas (por ejemplo módulos enlazados estáticamente en unos casos y de forma dinámica en otros) con lo que el sistema es difícil de documentar y mantener, falta de abstracción ofreciéndose interfaces únicos y rígidos, ausencia de metadatos ofreciéndose pocas alternativas a la ampliación y configuración del sistema. Una solución es la arquitectura basada en componentes ya que permite una fácil sustitución de los módulos del sistema.

- ***Vendor Lock-In*:** arquitectura dependiente del producto. Cuando los proyectos de desarrollo de *software* se basan en soluciones tecnológicas propietarias de un proveedor particular, al tomar la tecnología de un producto se depende del vendedor. Nuevas versiones del producto comercial pueden ser incompatibles, alejadas de los estándares y las arquitecturas abiertas, todo ello dificulta el mantenimiento del sistema. La solución a este problema se basa en no adquirir paquetes en base a la publicidad del vendedor sin un examen técnico detallado y pruebas de validación. Se puede minimizar su impacto utilizando el patrón adaptador, de forma que encapsulamos el acceso a la clase y futuros cambios del producto solo afectaran al adaptador y no a toda la aplicación.

- ***Architecture by implication*:** arquitectura implícita. Sucede cuando no se específica la arquitectura del sistema o ignora alguno de sus apartados: arquitectura del *software* (especificación del lenguaje, bibliotecas, nomenclaturas), arquitectura *hardware* (configuración de clientes y servidores), arquitectura de comunicación (protocolos y dispositivos de red), arquitectura de

datos (archivos y bases de datos), arquitectura de seguridad, administración de sistemas, etc. Se suele dar cuando se considera que la documentación no es necesaria y la arquitectura a utilizar se da por supuesta, puede ser un exceso de confianza de desarrolladores con experiencias exitosas en otros proyectos, ausencia de soporte técnico, etc.

- ***Design by committee***: diseño por comité, navaja suiza, chapa de oro, enfermedad de estandarización. Se da cuando el proyecto se diseña a través de las reuniones de un comité demasiado numeroso o inexperto. Las reuniones se hacen largas e improductivas, se pretende incorporar las ideas de todos al proyecto. Falta un responsable único del proyecto, se incorporan chapas de oro, elementos que aumentan la dependencia de nosotros por el cliente, la documentación se hace voluminosa e ilegible, aumenta la complejidad de las abstracciones pretendiendo que sirvan para todo, las decisiones solo se pueden realizar por el comité con lo que se retrasa el avance del proyecto. La solución implica que: los equipos que elaboran las abstracciones deben ser reducidos y de expertos, asignar responsabilidades a cada miembro del equipo o comité, las reuniones deben ser concisas y con unos objetivos concretos, asignar prioridades a los requerimientos y ceñir las abstracciones a los mismos.

- ***Reinvent the wheel***: se supone que se debe desarrollar desde cero, falta información y tecnología reutilizable entre proyectos. Falta de visión empresarial. Arquitectura cerrada y específica para un proyecto. La solución es la búsqueda de tecnología reutilizable

(la reusabilidad empieza por reutilizar lo existente antes que por diseñar sistemas reutilizables), identificar interfaces estándar.

1.6.3 Gestión de proyectos de *software*

Esta la llamada parálisis de análisis, que se da cuando un equipo de inteligentes analistas comienzan una fase de análisis que solo acaba cuando se cancela el proyecto. Es el síndrome del 'gran proyecto' se refleja en la pretensión de que el sistema lo haga todo, se realice con las últimas herramientas, se utilicen los últimos paradigmas, etc., todo esto requiere nuevos equipos de desarrollo, con lo que el proyecto se reinicia a cada nueva crisis de grandiosidad.

También influye el emplear analistas con reducidos conocimientos, de forma que las simultáneas curvas de aprendizaje cuestionan lo ya analizado y obligan a una revisión continua. Pueden existir conflictos entre objetivos en ocasiones por causas políticas.

La solución consiste en realizar desarrollos pequeños (Debe tomarse en cuenta la velocidad de evolución de la tecnología, un proyecto a más de 6 meses se topará con cambios tecnológicos). Utilizar desarrolladores antes que analistas, los desarrolladores pueden aprender a analizar un requerimiento, mientras que la mayoría de los analistas no son capaces de desarrollar una solución. Comenzar el proyecto con un prototipo de arquitectura y un conjunto reducido de requerimientos para desarrollar en no más de un mes. Entre otros antipatrones de gestión de proyectos tenemos: *corncob*: personas problemáticas, administración irracional y la des administración de proyectos.

1.7 Herramientas para la utilización de patrones de diseño

En la actualidad, no se dispone de ninguna herramienta para la utilización directa de los patrones de diseño en el proceso de desarrollo de una manera completa. Algunos intentos se han basado en el diseño de clases que representan al patrón para poderse utilizar directamente, en otros casos se han utilizado plantillas (*templates*). El problema reside en el propio concepto de patrón, ya que un mismo patrón puede tener muchas implementaciones diferentes (tomar una codificación concreta puede ser útil en unos casos y molesta en otros) ya que el patrón es solo la idea y no una codificación concreta. De hecho al codificar un patrón éste deja de serlo, ya no es una solución a un problema reiterativo, sino un ejemplo de una solución a un problema concreto. Por otro lado, no disponer de una herramienta capaz de gestionar los patrones dificulta el seguimiento de los mismos a partir del código.

La solución más frecuente pasa por incluir en los nombres de las clases el nombre del patrón o del colaborador concreto, no obstante no es un método lo suficientemente convincente.

2. CATÁLOGO DE PATRONES DE DISEÑO DE *SOFTWARE*

Hasta ahora hemos visto los aspectos teóricos, orígenes y especificaciones formales de los patrones de diseño, pero no un patrón de diseño específico. En este capítulo examinaremos a detalle distintos patrones de diseño, pero antes de ello, necesitamos saber tres puntos muy importantes para su correcta utilización:

- Los problemas de diseño que los patrones nos ayudan a resolver.
- La forma de seleccionar un patrón de diseño, ya que debido a la gran cantidad de patrones existentes, puede resultar un poco difícil encontrar un patrón que se adecue a un problema específico al que nos enfrentamos.
- La forma de utilizar un patrón de diseño. Ya que se ha seleccionado un patrón en particular, y sabemos que problema puede resolver, debemos saber como utilizarlo de la mejor manera.

2.1 Problemas de diseño que los patrones ayudan a resolver

Un objeto esta compuesto de datos y procedimientos que operan sobre dichos datos. Los procedimientos son llamados métodos u operaciones, los cuales son ejecutadas cuando el objeto recibe una petición o mensaje de un cliente. Los mensajes son la única forma de lograr que un objeto ejecute una operación, y debido a esta restricción, se dice que el estado interno del objeto se encuentra encapsulado. A continuación se presentan varios de los problemas que los diseñadores enfrentan, y como los patrones ayudan a resolverlos.

2.1.1 Encontrar los objetos apropiados

Una de las partes más difíciles del diseño orientado a objetos es descomponer un sistema en objetos, debido a que una gran cantidad de factores entran en juego, como la encapsulación, granularidad, dependencia, flexibilidad, desempeño, reutilización, entre otras. Los patrones de diseño ayudan a identificar las abstracciones menos obvias y los objetos que las representan, por ejemplo, objetos que representan un proceso o algoritmo, por lo regular no tienen una representación física en un sistema real, pero son una parte crucial para un diseño flexible. El patrón *State* representa cada estado de una instancia como un objeto, pero estos objetos son difíciles de visualizar si no se tiene el conocimiento del patrón que los origina.

2.1.2 Determinar la granularidad de un objeto

Otro problema que se enfrentan los diseñadores es como saber que debe ser un objeto y que no. El patrón fachada (*Facade*) describe como representar subsistemas como objetos, y el patrón *Flyweight* como manejar un gran número de objetos. Otros patrones describen como descomponer un objeto en objetos más pequeños, como los patrones *Abstract Factory* y *Builder*, los cuales se encargan de producir objetos cuya única responsabilidad es crear otros objetos, mientras que el patrón *command* produce objetos cuya responsabilidad es implementar peticiones en otros objetos o grupos de objetos.

2.1.3 Especificar la interfaz de un objeto

La definición de la interfaz de un objeto es otra situación en la que los patrones son de mucha utilidad. Cada operación declarada por un objeto específica el nombre de la operación, los objetos que toma como parámetros y el valor de retorno de la operación, lo cual es llamado la Firma de la operación. El conjunto de firmas de operaciones de un objeto es lo que llamamos la interfaz del objeto. Cualquier petición que coincida con una operación declarada por el objeto puede ser enviada hacia el objeto. Un tipo es un nombre utilizado para describir una interfaz particular. Un objeto puede tener diferentes tipos y diferentes objetos pueden compartir un tipo. Dos objetos del mismo tipo necesitan compartir solamente parte de su interfaz.

Las interfaces son fundamentales en sistemas orientados a objetos, ya que los objetos son conocidos únicamente a través de sus interfaces. No hay forma de conocer algo acerca de un objeto sin pasar por su interfaz. Los patrones de diseño ayudan a definir interfaces identificando elementos clave y los datos que son enviados a través de las interfaces. El patrón *memento* describe como encapsular y guardar el estado interno de un objeto, para que el objeto pueda ser restaurado a ese estado más tarde. Algunos patrones también especifican relaciones entre interfaces, como el *Decorator*.

2.1.4 Especificar la implementación de un objeto

La implementación de un objeto esta definida por su clase, y los patrones, ayudan a especificar la implementación de un objeto. La clase define los datos internos del objeto y su representación, y define las operaciones que el objeto puede realizar. Una clase abstracta tiene como objetivo definir una interfaz común para sus subclases. Una clase abstracta no puede ser instanciada. Las operaciones que una clase abstracta define pero no implementa son llamadas operaciones abstractas. Las subclases pueden redefinir el comportamiento de sus clases padre, es decir, pueden rescribir una operación definida por su clase padre. La herencia de clases permite definir clases simplemente extendiendo otras clases, facilitando la creación de familias de objetos.

Es importante entender la diferencia entre la clase de un objeto y su tipo. La clase define el estado interno del objeto y la implementación de sus operaciones, mientras que el tipo del objeto se refiere a su interfaz. Un objeto puede tener diferentes tipos, y objetos de diferentes clases pueden tener el mismo tipo. Existe una estrecha relación entre clase y tipo, ya que la clase define las operaciones que un objeto puede realizar, también define el tipo del objeto.

También es importante comprender la diferencia entre la herencia de clases y la herencia de interfaces (subtipos). La herencia de clases define la implementación de un objeto en términos de la implementación de otro, mientras que la herencia de interfaces define cuando un objeto puede ser usado en lugar de otro, es decir, polimorfismo. Muchos de los patrones de diseño se basan en esta distinción, como el patrón *Chain of responsibility*, en el cual los objetos deben compartir un tipo común, pero usualmente no comparten una implementación común.

La herencia de clases es básicamente un mecanismo para extender la funcionalidad de una aplicación reutilizando esa funcionalidad de las clases padre, lo cual permite definir una nueva clase de objeto rápidamente en términos de un objeto ya definido, pero la implementación de la reutilización no lo es todo, ya que se debe tener habilidad para definir herencia, con el objetivo de formar familias de objetos con interfaces idénticas, ya que el polimorfismo depende de ello. Cuando la herencia es utilizada de la manera apropiada, todas las clases derivadas de una clase abstracta compartirán su interfaz. Existen dos beneficios al manipular objetos solamente en términos de la interfaz definida por una clase abstracta:

- Los clientes no necesitan conocer los tipos específicos de los objetos que utilizan, si los objetos se adecuan a la interfaz que el cliente espera.
- Los clientes no necesitan conocer las clases que implementan estos objetos. Los clientes únicamente conocen las clases abstractas que definen la interfaz.

Con base en esto, se define el primer principio del diseño orientado a objetos: "Se debe programar hacia una interfaz, no a una implementación". No debe declararse variables para ser instancias de una clase concreta particular, por el contrario, debe hacerse para una interfaz definida por una clase abstracta.

Claro está que es necesario instanciar clases concretas (es decir, especificar una implementación particular) en algún punto del sistema, y los patrones de creación a*bstract factory*, *builder*, *factory method*, *prototype* y *singleton* permiten realizar esa tarea; abstrayendo el proceso de creación de los objetos, estos patrones ofrecen diferentes formas de asociar una interfaz con su implementación de forma transparente en el momento de la creación

o instanciación del objeto. Los patrones de creación ayudan a que el sistema escrito esté desarrollado en términos de interfaces y no de implementaciones específicas (clases concretas).

2.1.5 Utilizando los mecanismos de reutilización

Dos de las formas más comunes de reutilizar funcionalidad en sistemas orientados a objetos es a través de la herencia de clases y la composición de objetos. La reutilización con herencia de clases es comúnmente llamado *white box reuse*. Este término se refiere a que con herencia, los datos internos de la clase padre son a veces visibles a sus clases hijas. La composición de objetos es una alternativa a la herencia de clases, ya que se puede obtener nueva funcionalidad ensamblando objetos, para obtener una funcionalidad mayor y más compleja. La composición de objetos requiere que los objetos a ser ensamblados tengan correctamente definidas sus interfaces. Este estilo de reutilización es comúnmente llamado *black box reuse*, ya que los detalles internos de los objetos no son visibles, es decir, los objetos se manejan únicamente como cajas negras.

La herencia de clases es definida de forma estática, en tiempo de compilación. Esta forma de reutilización hace más fácil modificar la implementación a ser reutilizada, ya que cuando una subclase sobrescribe algunas operaciones, puede afectar las operaciones que hereda, asumiendo que sean llamados las operaciones que han sido reescritos. La herencia de clases tiene algunas desventajas, como por ejemplo: No se puede cambiar la implementación heredada de una clase padre en tiempo de corrida, ya que la herencia es definida en tiempo de compilación. Otra desventaja es que la herencia por lo regular expone detalles de la implementación de la clase padre a sus clases hijas, por lo que se dice que "la herencia rompe la encapsulación". La implementación de subclases se vuelve muy dependiente

de la implementación de la clase padre, por lo que cualquier cambio en la clase padre forzará a realizar cambios en las clases hijas. Estas dependencias de implementación pueden causar problemas cuando se intenta reutilizar una subclase, por lo que limita la flexibilidad y reusabilidad del diseño. Una solución a este problema es usar herencia solo de clases abstractas, ya que estas clases proveen poca o ninguna implementación.

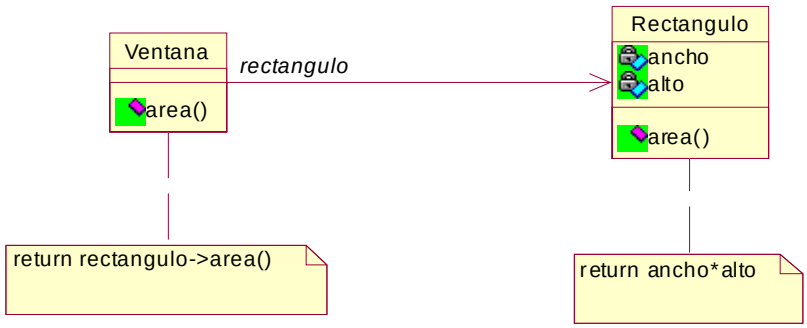
La composición de objetos es definida en forma dinámica, adquiriendo referencias a otros objetos en tiempo de corrida. La composición requiere que los objetos respeten las interfaces de otros objetos, lo cual requiere un cuidadoso diseño de las interfaces, para permitir la utilización de un objeto con muchos otros. Como los objetos son manejados únicamente por su interfaz, “no se rompe la encapsulación”. Cualquier objeto puede ser reemplazado por otro en tiempo de corrida si comparten el mismo tipo, y como la implementación de los objetos está escrita en términos de interfaces, existirán menos dependencias de implementación. La utilización de la composición de objetos en mayor grado que la herencia de clases ayuda a mantener la propiedad de encapsulación en las clases, y a mantener estas clases enfocadas en una tarea específica. Además, las clases y jerarquías de clases serán pequeñas y es probable se mantengan de esa manera. Pero, un diseño basado en la composición de objetos tendrá un gran número de objetos, y el comportamiento del sistema dependerá de las relaciones entre ellos, en lugar de ser definido por una sola clase.

En base a lo anterior, se define el segundo principio del diseño orientado a objetos: Favorecer la composición de objetos en lugar de la herencia de clases.

Una forma de implementar la composición de objetos de una forma tan poderosa como la herencia es a través de la delegación. En la delegación, dos objetos se encargan de manejar una petición: un objeto receptor delega operaciones a su delegado. Pero con la herencia, una operación heredada puede hacer referencia al objeto recibido a través de la variable *this*. Para lograr el mismo efecto con la delegación, el receptor se pasa a sí mismo al objeto delegado para dejar que la operación delegada haga referencia al objeto receptor.

Por ejemplo, en lugar de hacer la clase ventana una subclase de rectángulo, la clase ventana puede reutilizar el comportamiento de la clase rectángulo manteniendo una instancia de la clase rectángulo, delegándole el comportamiento específico de esa clase. En otras palabras, en lugar que una ventana sea un rectángulo, la ventana poseerá un rectángulo. Ahora la ventana podrá enviar peticiones a la instancia de rectángulo explícitamente, a diferencia de la herencia de clases, en la que la clase ventana hereda estas operaciones.

Figura 9. Clase ventana delega su operación area() hacia la instancia rectángulo



La principal ventaja de la delegación, es que facilita la composición de comportamientos en tiempo de corrida y la realización de cambios en la forma en que están compuestos los objetos. Por ejemplo, la ventana puede

cambiar a forma circular en tiempo de corrida simplemente reemplazando la instancia de rectángulo por una instancia de una clase circulo, asumiendo que las clases rectángulo y circulo son del mismo tipo.

La delegación tiene una desventaja común a las técnicas que hacen *software* más flexible a través de la descomposición de objetos: el *software* altamente parametrizado es más difícil de entender que un *software* definido en forma estática[10]. La delegación es una buena opción de diseño únicamente cuando simplifica un diseño en lugar de complicarlo más. No existe una regla que diga cuándo usar la delegación y cuándo no, porque la efectividad de esta técnica dependerá del contexto y de la experiencia en su manejo. Hay patrones que utilizan delegación, como por ejemplo *State*.

Otra técnica para reutilizar funcionalidad es a través de tipos parametrizados, también conocidos como plantillas (*templates*). Esta técnica permite definir un tipo sin específicar los otros tipos que utiliza. Los tipos no específicados son proporcionados como parámetros a la hora de utilizarlos. Por ejemplo, la clase Lista puede estar parametrizada por el tipo de elementos que contiene. Para declarar una lista de enteros, se debe proporcionar el tipo *integer* como parámetro a la lista parametrizada. Los tipos parametrizados nos dan una tercera forma (adicional a herencia de clases y composición de objetos) para definir o componer comportamientos en sistemas orientados a objetos. Muchos diseños pueden ser implementados al utilizar cualquiera de estas tres técnicas.

Existen diferencias entre estas tres técnicas que deben ser mencionadas. La composición de objetos permite cambiar el comportamiento definido en tiempo de corrida, pero esto puede producir un descenso en la eficiencia del sistema. La herencia permite proveer implementaciones de operaciones predeterminadas y deja a las subclases

sobrescribirlas, y los tipos parametrizados permiten cambiar los tipos que una clase puede utilizar, pero ni la herencia ni los tipos parametrizados, pueden cambiar en tiempo de corrida. La mejor forma o técnica dependerá del diseño y las restricciones de la implementación.

A continuación se presenta un resumen con las principales características a evaluar al elegir un método de reutilización, con una ponderación para cada método, donde ✓✓✓ representa la ponderación máxima y ✓ la ponderación mínima. En los métodos que no se colocó ponderación, dicha característica no es proporcionada. (**N/A** No Aplica).

	Herencia	Composición de objetos	Tipos parametrizados
Flexibilidad en tiempo de corrida	.	✓✓✓	.
Performance	✓✓✓	✓✓	✓✓
Complejidad de implementación	✓	✓✓✓	✓✓
Legibilidad del diseño	✓✓✓	✓	✓✓
Encapsulación	✓	✓✓✓	**N/A**
Flexibilidad y resusabilidad de diseño	✓	✓✓✓	**N/A**

Con los tipos parametrizados, la encapsulación no aplica ya que este método no es una técnica estrictamente orientada a objetos. Para flexibilidad y reusabilidad del diseño no se posee suficiente información para asignar una ponderación.

La representación en UML para la herencia es a través de diagramas de clases y las relaciones estáticas definidas entre ellas (relaciones de herencia, agregación y asociación); para representar la composición de objetos se utiliza la relación de dependencia o instanciacion, diagramas de objetos y diagramas de interacción.

2.1.6 Relación de estructuras dinámicas (*run time)* y estáticas (*compile time*)

La estructura dinámica (*run time*) de un programa orientado a objetos, por lo regular, presenta poca semejanza con su estructura codificada. El código de la estructura (*compile time*) consiste en clases relacionadas mediante herencia, y la estructura dinámica (*run time*) consiste en redes de comunicación de objetos cambiantes rápidamente.

Para empezar, es necesario conocer la diferencia entre la agregación y asociación de objetos, y cómo se presentan en tiempo de ejecución y de compilación. La agregación implica que un objeto pertenece o es responsable de otro objeto, generalmente se habla de un objeto teniendo o siendo parte de otro objeto. También implica que el objeto agregado y su dueño tienen tiempo de vida iguales. La asociación implica que un objeto raramente conoce a otro objeto. Los objetos asociados pueden realizar peticiones entre ellos, pero no son responsables uno del otro. La relación de asociación es más débil que la de agregación. Es muy fácil confundir la agregación con la asociación, ya que por lo regular son implementadas de la misma manera. En los lenguajes de programación no existe diferencia entre la agregación y la asociación. Por ejemplo, en C++, la agregación puede ser implementada al definir variables miembro que son instancias reales, pero es más común definirlas como apuntadores o referencias a instancias. La asociación es implementada de la misma forma que la agregación.

Debido a estas diferencias, sabemos que el código no revela nada acerca de cómo un sistema debe trabajar. La estructura de tiempo de ejecución debe ser impuesta por el diseñador, no por lenguaje. Las relaciones entre objetos y sus tipos deben ser diseñados de la mejor manera, ya que de ello depende qué tan buena o mala es la estructura de tiempo de ejecución de un programa.

Muchos patrones de diseño capturan la diferencia entre las estructuras en tiempo de compilación y en tiempo de ejecución explícitamente. Por ejemplo, los patrones *composite* y *decorator* son útiles para construir complejas estructuras dinámicas. El patrón *observer* involucra estructuras dinámicas que son difíciles de entender si no se conoce el patrón. Por lo general, las estructuras de tiempo de ejecución no son claras al verlas desde el código, a menos que se conozcan los patrones necesarios.

2.1.7 Diseñar anticipándose a los cambios

La clave para maximizar la reutilización está en anticiparse a los nuevos requerimientos y cambios en los requerimientos actuales. El diseño del sistema debe ser capaz de evolucionar de manera adecuada. Es necesario considerar la forma en que el sistema evolucionará durante su período de vida. Un diseño que no tome en cuenta la evolución del sistema aumenta el riesgo de que exista la necesidad de realizar modificaciones al diseño en el futuro. Estos cambios pueden incluir modificaciones en la definición de las clases y su implementación, modificaciones en los clientes, realización de nuevas pruebas, entre otros. Un rediseño afecta muchas partes de un sistema de *software*, y el costo es elevado.

Los patrones de diseño ayudan a evitar un rediseño, asegurándose que un sistema pueda cambiar o evolucionar en formas específicas. Cada patrón de diseño permite que varios aspectos de la estructura del sistema puedan cambiar, independientemente de otros aspectos. A continuación se describen algunas causas comunes que conducen a la realización de un rediseño, y los patrones que están enfocados a evitar esas causas:

2.1.7.1 Creación de un objeto especificando su clase explícitamente

Al especificar el nombre de una clase cuando se crea un objeto se está asociando directamente el objeto a una implementación particular, en lugar de a una interfaz particular. Esta asociación directa puede complicar cambios futuros, por lo que se deben crear los objetos de forma indirecta. Los patrones que se enfocan a esta situación son: *abstract factory*, *factory method* y *prototype*.

2.1.7.2 Dépendencia sobre operaciones específicas

Cuando se específica una operación particular, se asocia directamente una forma de satisfacer una petición o mensaje. Evitar este tipo de codificación hace más fácil realizar cambios a la forma en que una petición o mensaje es manejado en tiempo de compilación y de ejecución. Los patrones enfocados a esta situación son: *chain of responsibility* y *command*.

2.1.7.3 Dependencia de plataforma de *hardware* y *software*

Un *software* dependiente de alguna plataforma específica es muy difícil portarlo a otras plataformas. Por lo tanto, es importante diseñar sistemas que límiten la dependencia del *software* de la plataforma. Los patrones enfocados a esto son *abstract factory* y *bridge*.

2.1.7.4 Dependencia de representaciones de objetos o implementaciones

Los clientes que conocen la forma en que un objeto es representado, almacenado, su ubicación o su implementación pueden necesitar ser modificados cuando estos objetos sufran cambios. Ocultar esta información a los clientes mantiene los cambios de los clientes y los otros objetos de forma separada. Patrones: *abstract factory*, *bridge*, *memento*.

2.1.7.5 Dependencia de algoritmos

Los algoritmos por lo regular son optimizados o reemplazados durante el desarrollo. Los objetos que dependen de un algoritmo deberán ser modificados cuando el algoritmo cambie, por lo cual, los algoritmos que tienen mayor probabilidad de cambio deben ser aislados. El patrón *builder* se enfoca en este aspecto.

2.1.7.6 Clases fuertemente acopladas

Cuando existen clases fuertemente acopladas es difícil reutilizar la funcionalidad de cada clase por separado, ya que unas dependen de otras. Esto provoca que en un sistema no sea posible cambiar o remover alguna clase sin entender o cambiar muchas otras clases. El sistema se convierte en

una masa densa que es difícil de entender, portar y mantener. Patrones: *abstract factory*, *bridge*, *chain of responsibility*, *command*, *facade* y *observer*.

2.1.7.7 Extensión de funcionalidad a través de subclases

Definir un objeto a través de subclases no es una tarea fácil. Definir una nueva subclase requiere un conocimiento profundo de la clase padre, por ejemplo, redefinir una operación puede requerir que otra operación también sea redefinida. Las subclases pueden generar una explosión de clases, es decir, puede ser necesario introducir gran cantidad de subclases para una extensión simple. La composición de objetos y la delegación en particular proveen alternativas flexibles para la herencia. Se puede agregar nueva funcionalidad a una aplicación mediante composición de objetos existentes en formas nuevas en lugar de definir nuevas subclases de clases existentes. Por otra parte, el uso excesivo de la composición de objetos puede hacer los diseños menos legibles. Muchos patrones de diseño producen diseños en los cuales se puede introducir nueva funcionalidad, simplemente definiendo una subclase y realizando composición de sus instancias, con otras existentes. Patrones: *bridge*, *chain of responsibility*, *composite*, *decorator* y *observer*.

2.1.7.8 Imposibilidad de alterar clases

Algunas veces es necesario modificar una clase que no puede ser modificada de manera convencional, o talvez se necesite el código fuente de una clase y no esté disponible, como sucede con librerías de clases comerciales. Algunos patrones ofrecen formas de modificar las clases en estas circunstancias, como por ejemplo, *adapter* y *decorator*.

Estas situaciones reflejan la flexibilidad que los patrones de diseño ayudan a introducir en el *software*. El punto crítico de esa flexibilidad dependerá del tipo de *software* que se desea construir.

A continuación se presenta el rol que juegan los patrones de diseño en tres principales tipos de *software* que se han identificado: programas de aplicación, paquetes de herramientas (*toolkits*) y las armazones (*frameworks*).

En un programa de aplicación, la reutilización interna, mantenibilidad y escalabilidad son las principales prioridades. La reutilización interna se encarga de verificar que el diseño y la implementación sean correctos y que no se diseñe o se implemente más de lo necesario. Los patrones de diseño que reducen dependencias producen un incremento en la reutilización interna, ya que al eliminar dependencias sobre operaciones específicas aislando y encapsulando cada operación, se hace más fácil la reutilización de una operación en diferentes contextos.

Un *toolkit* es un conjunto de clases relacionadas y reutilizables que proveen alguna utilidad de propósito general. Por ejemplo, una colección de clases para manejar listas, pilas, tablas, etc. Un *toolkit* no impone un diseño particular en una aplicación, sólo proveen funcionalidad que puede ayudar a la nueva aplicación a realizar su trabajo. Los *toolkits* ayudan a los desarrolladores a rescribir programas de funcionalidad general o común, y se enfocan en la reutilización de código.

Un *framework* o armazón es un conjunto de clases relacionadas que describen una estructura de *software* fácilmente ampliable y reutilizable. Por ejemplo, una armazón puede servir para construir compiladores para diferentes lenguajes de programación, o construir aplicaciones de modelos

financieros. Una armazón se puede adecuar para una aplicación particular, creando subclases para la aplicación específica de las clases abstractas de la armazón.

La armazón define la arquitectura de la aplicación, para que los diseñadores y desarrolladores puedan enfocarse en detalles específicos de la aplicación. La armazón captura las decisiones de diseño comunes al dominio de la aplicación. Las armazones se enfocan en la reutilización de diseño en lugar de la reutilización de código. Cuando se utiliza un *toolkit*, se escribe el cuerpo principal de la aplicación y se llama o ejecuta código reutilizable definido por el *toolkit*. Cuando se utiliza una armazón, lo que se reutiliza es el cuerpo principal de la aplicación, y se escribe el código que es llamado por la armazón. Esto implica que se debe escribir las operaciones con nombres particulares y convenciones de ejecución definidas, pero se reducen las decisiones de diseño que se deben tomar, ayuda a construir aplicaciones rápidamente, y las aplicaciones tienen estructuras similares, lo cual hace más fácil su mantenimiento y reflejan mayor consistencia para los usuarios.

Diseñar una aplicación no es fácil, diseñar un *toolkit* es aún menos fácil y diseñar una armazón es la menos fácil de todas. Un diseñador de armazones debe confiar en que una arquitectura definida trabajará correctamente para todas las aplicaciones de un dominio. Cualquier cambio al diseño de la armazón puede reducir sus beneficios de forma considerable, ya que la principal ayuda que brinda una armazón es la arquitectura que define.

Los patrones ayudan a crear la arquitectura de una armazón que se adecue a distintas aplicaciones sin necesidad de rediseñar la arquitectura. Debido a algunas características comunes entre patrones y armazones, se tiende a confundir los términos, pero existen tres diferencias principales entre ellos:

1. Los patrones de diseño poseen mayor abstracción que las armazones.
2. Los patrones de diseño son elementos de una arquitectura más pequeños que los de una armazón. Una armazón típica contiene varios patrones de diseño, pero un patrón de diseño nunca contiene varias armazones.
3. Los patrones de diseño son menos especializados que una armazón. Por ejemplo, una armazón siempre tiene un dominio particular de aplicaciones, pero un patrón de diseño puede ser aplicado en casi la mayoría de aplicaciones.

2.2 Seleccionando un patrón de diseño

En la actualidad el número de patrones de diseño es muy grande, y la tendencia es que siga aumentando, por lo que escoger un patrón que se adecúe al problema que tratamos de resolver puede ser un poco difícil, especialmente si no se está familiarizado con los distintos patrones existentes.

Una forma para encontrar el patrón adecuado para nuestro problema es estudiando y analizando comó los patrones de diseño resuelven los problemas de diseño, cómo identifican y determinan la granularidad de los objetos y sus interfaces. Otra forma para hallar el patrón correcto es leer la intención de cada patrón, ya que esta sección del patrón proporciona una idea de que pretende el patrón. También se puede analizar la relación entre

los patrones, ya que se puede identificar el patrón o grupo de patrones más adecuado al problema actual.

También se debe tomar en cuenta las causas por las que ocurre un rediseño, para identificar si nos encontramos dentro de una de esas causas e identificar los patrones que ayudan a evitarlas. O bien, se debe considerar qué partes del diseño pueden variar. En lugar de considerar qué causas producen cambios en un diseño, debemos considerar qué deseamos dejar de forma variable o en disponibilidad de cambiar, sin realizar un rediseño. En este enfoque se concentran varios patrones de diseño.

2.3 Forma de utilizar un patrón de diseño

Ya que se ha elegido el o los patrones adecuados al problema que tratamos de resolver, debemos aplicarlo de la manera adecuada. No existe una guía o receta para aplicar los patrones, sólo la experiencia nos indicará la forma correcta de aplicarlos, pero se propone una serie de pasos para iniciar a los nuevos diseñadores en el uso de los patrones de diseño:

- Leer el documento que describe el patrón, para tener una idea de lo que indica el patrón, poniendo especial atención en las secciones de aplicabilidad y consecuencias.

- Leer nuevamente el patrón, ahora enfocándose en las secciones de estructura, participantes y colaboración, para comprender de manera correcta las clases y objetos que forman el patrón y cómo se relacionan.

- Analizar el código de ejemplo, para conocer un ejemplo concreto de la implementación del patrón.

- Definir nombres para los participantes principales en la aplicación. Los nombres definidos en los patrones de diseño son muy abstractos para aparecer directamente en la implementación de la aplicación.

- Definir las clases, declarar sus interfaces, establecer las relaciones de herencia y definir instancias para representar datos y referencias a objetos. Se deben identificar las clases de la aplicación que serán afectadas por el patrón, y modificarlas de acuerdo a él.

- Definir nombres específicos a las operaciones en el patrón. Los nombres dependerán de la aplicación, y se debe ser consistente en la convención para nombrar las operaciones.

- Implementar las operaciones para definir las responsabilidades y colaboraciones en el patrón. La sección de implementación ofrece *tips* o técnicas que sirven de guía en la implementación.

Los patrones de diseño no deben aplicarse de forma indiscriminada. Con los patrones de diseño se alcanza flexibilidad y escalabilidad en el diseño, pero también pueden complicarlo y representar costos económicos y de desempeño. Los patrones de diseño deben aplicarse solamente si la flexibilidad que representa su utilización es realmente necesaria.

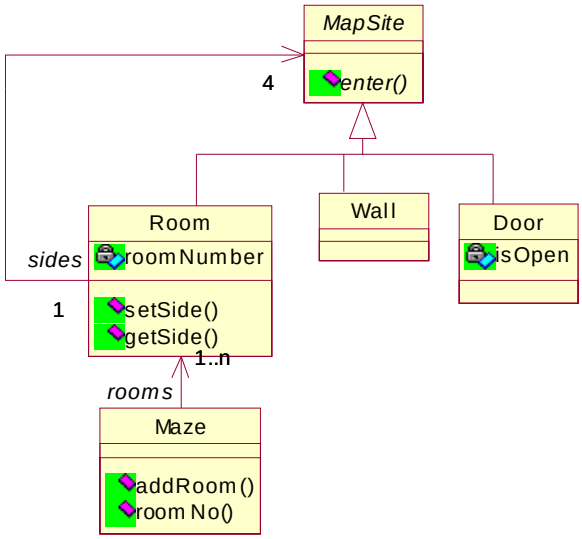
2.4 Patrones de creación

Los patrones de creación proveen una forma independiente de creación, composición y representación de los objetos instanciados. Un patrón de creación de clase[11] utiliza la herencia para variar la clase que es instanciada, mientras que un patrón de creación de objeto, delega la creación o instanciacion hacia otro objeto.

Los patrones de creación adquieren mayor importancia cuando un sistema depende mayormente de la composición de objetos que dé la herencia de clases. Estos patrones encapsulan la información referente a las clases concretas que el sistema utiliza y ocultan la forma en que las instancias de estas clases son creadas y organizadas, lo que proporciona gran flexibilidad en cuanto a qué, quién, cómo y cuándo se crea un objeto.

Según su alcance, tenemos entre los patrones de creación de clase al *factory method*, y entre los de creación de objeto: *abstract factory*, *builder*, *prototype* y *singleton*. Se utilizará un ejemplo en común para representar la implementación de los cinco patrones de creación. El ejemplo consiste en crear un laberinto para un juego, en el cual solo nos interesa cómo se crea el laberinto, por lo que no se tomara en cuenta detalles relacionados con el comportamiento y desarrollo del laberinto. El laberinto estará formado por un conjunto de cuartos, paredes, y puertas[12]. Cada cuarto tiene 4 lados: norte, sur, este y oeste. En la implementación de esta clase, se utiliza las iniciales en inglés de las direcciones mencionadas (*north, south, east, west*). La clase *MapSite* es la representación abstracta de todos los componentes del laberinto (*Maze*). En esta clase solo se define el método *enter()*. Este método se comportará dependiendo de qué objeto lo invoque, por ejemplo, si es una puerta, cambia de posición si esta abierta, y si no, no podrá cambiar de ubicación. r*oom, wall y door* son subclases concretas que definen relaciones estáticas entre los componentes del laberinto. La figura 10 muestra las relaciones definidas sobre estas clases.

Figura 10. Diagrama de clases ejemplo de laberinto



La clase *Maze* representa una colección de cuartos. Podemos agregar un cuarto a través del método *addRoom*(), y encontrar un cuarto en particular por su número, mediante el método *roomNo*(), que devuelve una referencia a un objeto *room*. En la implementación de estas clases, en Java, presentada a continuación, no se implementan algunos de los métodos de las clases, ya que no son de vital importancia para la comprensión de los patrones.

```
abstract class MapSite {
        public abstract void enter();
        }

class Room extends MapSite {
        private MapSite sides[];
        private int roomNo;

        public Room (int _roomNo) {}
        public MapSite getSide(char _side) {
                return null;
                }
        public void setSide (char _side, MapSite _mapSite) {}
        public void enter() {}
        }

class Wall extends MapSite {
        public Wall () {}
        public void enter() {}
        }

class Door extends MapSite {
```

```
        private Room room1;
        private Room room2;
        private boolean isOpen;

        public Door(Room _r1, Room _r2) {
                room1 =_r1;
                room2 =_r2;
                isOpen =true;
                }
        public void enter() {}
        }

class Maze {
        public Maze() {}
        public void addRoom(Room _room) {}
        public Room roomNo(int _roomNo) {
                //Implementacion Metodo de busqueda
                return null;
                }
        }

public class MazeGame {
        public MazeGame() {
                createMaze();
                }
        public Maze createMaze() {
                Maze aMaze =new Maze();
                Room r1 =new Room(1);
                Room r2 =new Room(2);
                Door theDoor =new Door(r1,r2);

                aMaze.addRoom(r1);
                aMaze.addRoom(r2);
                r1.setSide('N',new Wall());     //N ORTH
                r1.setSide('E',theDoor);        //E AST
                r1.setSide('S',new Wall());     //S OUTH
                r1.setSide('W',new Wall());     //W EST
                r2.setSide('N',new Wall());
                r2.setSide('E',new Wall());
                r2.setSide('S',new Wall());
                r2.setSide('W',theDoor);
                return aMaze;
                }
        }
```

La clase adicional *MazeGame* es la encargada de crear el juego, por lo tanto es responsable de la definición de las relaciones dinámicas entre los objetos creados. Una manera directa de crear los objetos, es mediante una serie de operaciones para agregar los componentes al laberinto e interconectarlos, lo cual está definido en el método *createMaze*() de la clase *MazeGame*, donde se crea un laberinto de dos cuartos con una puerta entre ellos. La implementación de este método es funcionalmente correcta, pero es completamente inflexible. Define una estructura o esquema del laberinto único. Cambiar la estructura del laberinto implica cambiar la implementación

de *createMaze*, ya sea mediante herencia, lo que implica sobrescribir nuevamente todo el método, o bien solo cambiar partes del método lo cual es más propenso a errores y no promueve la reutilización.

Los patrones de creación muestran formas de definir las relaciones dinámicas entre objetos de una manera más flexible, pero no necesariamente menos reducida y compleja. Hace más fácil cambiar las clases que definen los componentes del laberinto en forma dinámica. Por ejemplo, se necesita reutilizar el diseño de un laberinto para un nuevo juego que contiene laberintos encantados (*EnchantedMaze*). El juego de laberintos encantados tiene nuevos componentes, como puertas que pueden ser abiertas solo con un hechizo (*DoorNeedingSpell*), o cuartos con características mágicas(*EnchantedRoom*).

Realizar los cambios en el método *createMaze*() para poder manejar estos nuevos objetos no es una tarea fácil, ya que se encuentra definido dentro del código de forma explicita las clases concretas de los objetos que deseamos crear. Los patrones de creación muestran diferentes formas de remover estas referencias explicitas.

2.4.1 *Factory method*

Intención

Define una interfaz para la creación de un objeto al dejar a las subclases decidir cuál clase instanciar. Este patrón permite a las clases trasladar la creación de los objetos a las subclases.

Conocido como: constructor virtual.

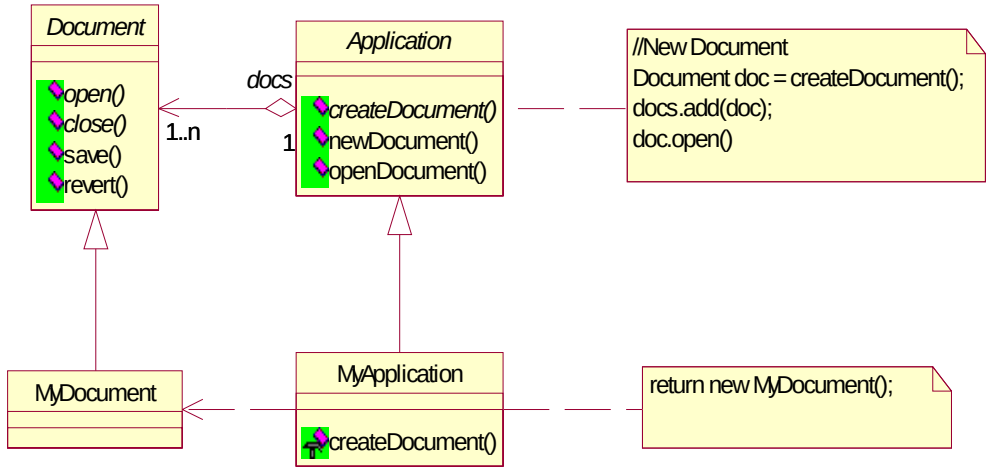
Motivación

Los *frameworks*[13] utilizan clases abstractas para definir las relaciones entre objetos, y también en algunas ocasiones son responsables de crear estos objetos. Por ejemplo, tenemos un *framework* para aplicaciones que pueden presentar múltiples documentos al usuario. Este *framework* posee dos clases abstractas: *application* y *document*, por lo que al implementar una aplicación específica se deben crear subclases de ellas. Para crear una aplicación que maneja archivos de texto, definimos las clases TextApplication y *TextDocument*, ambas subclases de *application* y *document* respectivamente. La clase *application* es responsable de manejar los objetos tipo *document*, y los creara cuando sea necesario, pero como las subclases de *Document* son específicas de la aplicación, no del *framework*, la clase *Application* no conocerá que subclase de *Document* debe crear un objeto, solo conoce el momento en que debe ser creado. El *framework* debe crear objetos de estas clases, pero él sólo maneja clases abstractas, de las cuales no puede crear objetos.

El patrón *factory method* presenta una solución: encapsular la información acerca de qué subclase debe instanciar y, mover esta información fuera del *framework*. Las subclases de *application* deberán implementar el método abstracto *createDocument* para retornar el objeto apropiado.

De esta manera, una vez se ha creado una instancia de una subclase de *application*, esta instancia puede crear instancias de documentos específicos sin conocer su clase. Se le llama al método *createDocument* un método fábrica (*factory method*) ya que es responsable de la fabricación de un objeto. El siguiente diagrama muestra las relaciones de las clases *application* y *document*.

Figura 11. *Framework* de manejo de múltiples documentos

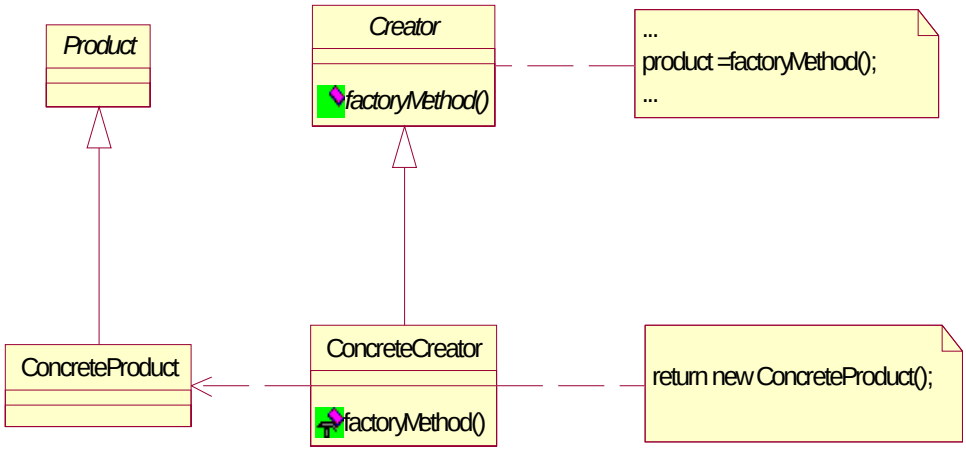


Aplicabilidad

El patrón *factory method* debe utilizarse cuando se presente alguna de las siguientes situaciones:

- La clase no conoce las clases de los objetos que debe crear
- La clase desea que sus clases hijas definan los objetos a crear
- Una clase delega responsabilidades sobre una de varias subclases, y se desea localizar la subclase delegada o sobre la que se delegó responsabilidad

Figura 12. Estructura patrón *factory method*



Participantes

- *Product*: define la interfaz de los objetos que el método *factoryMethod()* crea
- *ConcreteProduct*. Implementa la interfaz de *product*
- *Creator*. Declara el método *factoryMethod()* el cual retorna un objeto de la clase *Product*. Se puede definir una implementación default del método, que retorne un ConcreteProducto *default*
- *ConcreteCreator*. Sobrescribe el método *factoryMethod()* para retornar una instancia de ConcreteProduct

Colaboraciones

La clase creator permite a sus subclases definir el método *factoryMethod* para que este devuelva una instancia del objeto *ConcreteProduct*.

Consecuencias

Este patrón elimina la necesidad de definir explícitamente las clases de aplicaciones específicas dentro del código. En el código sólo se maneja la interfaz del objeto (*product*). Crear objetos dentro de una clase con un método *factoryMethod* es más flexible que crear el objeto directamente. Este patrón también permite conectar jerarquías de clases paralelas[14]. Una potencial desventaja de este patrón es que los clientes deban crear nuevas subclases de *Creator* solamente para crear un objeto en particular.

Implementación

Existen dos variantes principales: la primera, cuando creador es una clase abstracta y no provee una implementación para el método *factoryMethod* (las subclases deben implementar el método) y la segunda, cuando creador es una clase concreta y provee una implementación *default* para *factoryMethod* (en este caso se utiliza un método *factoryMethod* principalmente por flexibilidad).

Otra variante de la implementación de este patrón es parametrizar el *factoryMethod*, permitiendo crear objetos de distintas clases. El método *factoryMethod* toma un parámetro que identifica la clase de objeto que debe crear, y todos los objetos creados deben compartir la misma interfaz.

Una buena práctica es utilizar una forma estándar para nombrar los métodos y de esta forma identificar claramente que se está utilizando métodos fábrica o *factoryMethod*.

Muestra de código y utilización

El método *createMaze*() definido en la clase *MazeGame*[15] construye y devuelve un objeto *Maze* (laberinto). El problema con la implementación de este método es su inflexibilidad, ya que define explícitamente las clases de los objetos que crea. Para solucionar este problema introducimos varios *factoryMethod* en *MazeGame* para dejar que las subclases definan sus componentes. La clase *MazeGame* quedará de la siguiente forma:

```
class MazeGame {
        public MazeGame() {}
        public Maze createMaze() {
                Maze aMaze =makeMaze();
                Room r1 =makeRoom(1);
                Room r2 =makeRoom(2);
                Door theDoor =makeDoor(r1,r2);

                aMaze.addRoom(r1);
                aMaze.addRoom(r2);

                r1.setSide('N',makeWall());
                r1.setSide('E',theDoor);
                r1.setSide('S',makeWall());
                r1.setSide('W',makeWall());

                r2.setSide('N',makeWall());
                r2.setSide('E',makeWall());
                r2.setSide('S',makeWall());
                r2.setSide('W',theDoor);

                return aMaze;
                }

        //factory methods
        public Maze makeMaze() {
                return new Maze();
                }
        public Room makeRoom(int _n) {
                return new Room(_n);
                }
        public Wall makeWall() {
                return new Wall();
                }
        public Door makeDoor(Room _r1, Room _r2) {
                return new Door(_r1,_r2);
                }
        }
```

Se provee una implementación default que retorna objetos de las clases más simples *maze, room, wall* y *door*. Distintos juegos pueden heredar de la clase *MazeGame* para especializar partes del laberinto. Por ejemplo, veamos la siguiente clase:

```
class EnchantedMaze extends MazeGame {
        public EnchantedMaze() {
                super();
                }
        /* sobre escribe solo los metodos específicos para esta subclase */
        public final Room makeRoom(int _roomNo) {
                return new EnchantedRoom(_roomNo);
                }
        public final Door makeDoor(Room _r1, Room _r2) {
                return new DoorNeedingSpell(_r1,_r2);
                }
        }
```

La clase *EnchantedMaze* es una subclase de *MazeGame*, la cual sobrescribe solo los métodos que le interesa manejar de forma distinta al comportamiento definido por *default* en la clase padre. Las clases *EnchantedRoom* y *DoorNeedingSpell* son subclases de *room* y *door*, respectivamente. A continuación se muestra un fragmento de código donde se muestra la forma fácil y flexible para crear distintos juegos.

```
MazeGame game =new MazeGame();
game.createMaze();
MazeGame gameEnchanted =new EnchantedMaze();
gameEnchanted.createMaze();
```

Usos conocidos

Los métodos fabrica o *factory*Methods se encuentran en la mayoría de *toolkits* y *frameworks*.

Patrones relacionados

El patrón *abstract factory,* por lo regular, es implementado con *factory*Methods. También es utilizado con *prototype*.

2.4.2 *Abstract Factory*

Intención

Proveer una interfaz para la creación de familias de objetos relacionados o con dependencias entre ellos, sin especificar las clases concretas de los objetos.

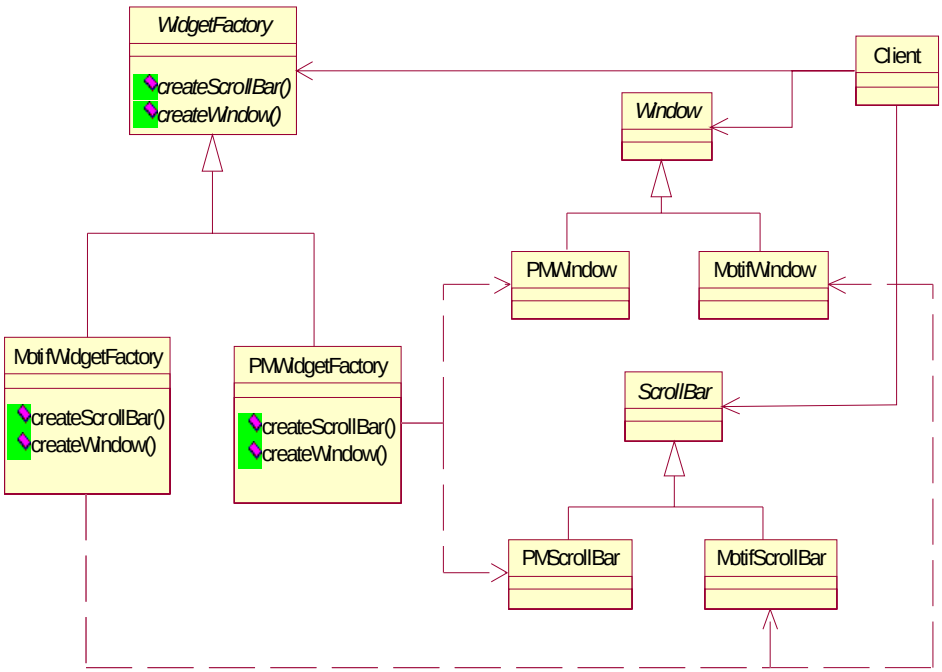
Conocido como: *Kit*.

Motivación

Si diseñamos un *toolkit*[16] para manejar la interfaz del usuario, que soporte diferentes *look-and-feels*[17] debemos considerar las diferencias en comportamiento y apariencia de los componentes específicos de cada interfaz del usuario, como barras de desplazamiento o *scroll bars*, ventanas y botones. Para ser portable sobre múltiples *look-and-feels*, una aplicación no debe especificar explícitamente en el código[18] los componentes particulares de un *look-and-feel*. Para lograr esto, se debe definir una clase abstracta que fabrique los objetos específicos de cada *look-and-feel* llamados *widgets* mediante una interfaz. También debe definirse una clase abstracta que maneje cada tipo de componente o *widget* y sus subclases concretas que los implementen, para cada *look-and-feel* específico. La interfaz de la clase abstracta que fabricará los objetos, *WidgetFactory*, define un método que retorna un nuevo *widget* por cada clase abstracta que define un tipo de componente específico.

Cuando un cliente llama, ejecuta estos métodos para obtener instancias de widgets particulares, permanece independiente del l*ook-and-feel* específico, ya que el cliente maneja los objetos específicos de un *look-and-feel* a través de una interfaz definida por una clase abstracta, no a través de clases concretas. La figura 13 presenta el diagrama de las clases mencionadas anteriormente.

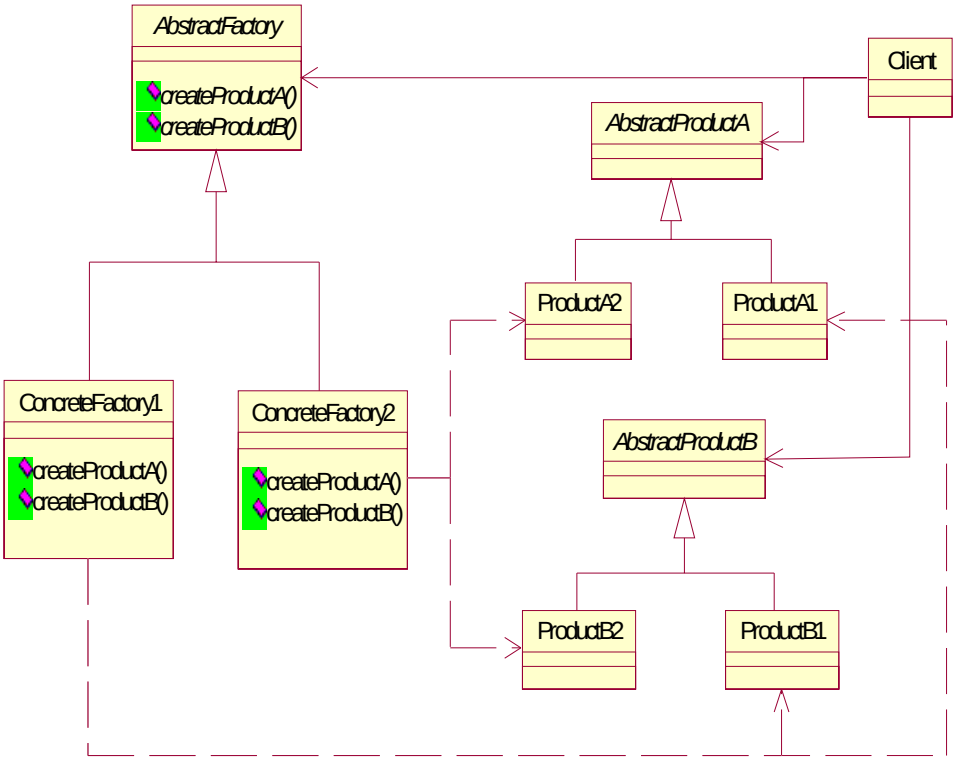
Figura 13. *Toolkit*: manejo de multiples *look-and-feel*



Aplicabilidad

- Cuando un sistema debe ser independiente de la forma en que sus objetos son creados, compuestos y representados
- Cuando un sistema debe ser configurado con uno de múltiples conjuntos o familias de productos específicos
- Cuando se desea implementar una librería de productos y no deseamos revelar la implementación particular de cada producto, solo las interfaces

Figura 14. Estructura patrón *abstract factory*



Participantes

- *AbstractFactory*. Declara la interfaz para crear objetos de una clase concreta mediante una clase abstracta
- *ConcreteFactory*. Implementa los métodos para crear los objetos específicos
- *AbstractProduct*. Declara una interfaz para un tipo de componente
- *ConcreteProduct*. Define el objeto a ser creado por su correspondiente fabrica concreta o *ConcreteFactory* e implementa la interfaz definida por *AbstractProduct*
- *Client*. Utiliza solamente las interfaces declaradas por las clases *AbstractFactory* y *AbstractProduct*

Colaboraciones

Por lo regular, sólo se crea una instancia de la clase *ConcreteFactory* de forma dinámica. Esta instancia se encarga de crear los objetos específicos de una implementación particular. Para crear objetos de otra implementación, se deberá utilizar una instancia diferente de la clase *ConcreteFactory*. La clase *AbstractFactory* traslada la creación de los objetos específicos hacia su clase hija *ConcreteFactory*.

Consecuencias

- Aisla o separa las clases concretas de los clientes. Los clientes manipulan las instancias a través de sus interfaces abstractas.

- Facilita el intercambio de familias de objetos solo cambiando la clase fábrica concreta o *ConcreteFactory* ya que esta crea una familia completa de productos. De acuerdo a la figura 13, podemos cambiar

de componentes *Motif* hacia componentes de *presentation manager,* solo cambiando la instancia *factory* correspondiente y recreando la interfaz.

- Promueve la consistencia entre los productos u objetos, ya que cuando los objetos en una familia son implementados para trabajar juntos, es necesario que la aplicación utilice objetos de una sola familia a la vez.

- El soporte para nuevas clases de componentes requiere extender la interfase definida por *AbstractFactory*, lo cual implica cambiar todas sus subclases previamente definidas.

Implementación

Como se menciono en la sección de colaboraciones, una aplicación solo necesita una instancia de una clase *ConcreteFactory* por cada familia de productos, por lo que es implementada como una instancia *Singleton*[19].

La clase *AbstractFactory* solo declara la interfaz para crear los productos, y es responsabilidad de la clase *ConcreteProduct* crearlos. La forma más común de crear los objetos es mediante la definición de un método fábrica o *FactoryMethod*[20] por cada producto.

Si se tienen muchas familias de productos posibles, la clase *ConcreteFactory* puede implementarse mediante el patrón *prototype* (Sección 2.4.4).

Muestra de código y utilización

Definimos la clase *MazeFactory* para la creación de los componentes de un laberinto o *Maze*. Los clientes que construyan los laberintos tomarán como parámetro una instancia de *MazeFactory* para que se pueda especificar de forma dinámica que clase de *Room, Wall*, o *Door* construir. La definición de esta clase se encuentra en el siguiente fragmento de código:

```
class MazeFactory {
        public MazeFactory() {}
        public Maze makeMaze() {
                return new Maze();
                }
        public Wall makeWall() {
                return new Wall();
                }
        public Room makeRoom(int _roomNo) {
                return new Room(_roomNo);
                }
        public Door makeDoor(Room _r1, Room _r2) {
                return new Door(_r1,_r2);
                }
        }
```

A continuación se encuentra la nueva implementación del método *CreateMaze* de la clase *MazeGame*, al utilizar como parámetro una instancia de *MazeFactory*.

```
public Maze createMaze(MazeFactory _factory) {
        Maze aMaze =_factory.makeMaze();
        Room r1 =_factory.makeRoom(1);
        Room r2 =_factory.makeRoom(2);
        Door theDoor =_factory.makeDoor(r1,r2);

        aMaze.addRoom(r1);
        aMaze.addRoom(r2);

        r1.setSide('N',_factory.makeWall());
        r1.setSide('E',theDoor);
        r1.setSide('S',_factory.makeWall());
        r1.setSide('W',_factory.makeWall());

        r2.setSide('N',_factory.makeWall());
        r2.setSide('E',_factory.makeWall());
        r2.setSide('S',_factory.makeWall());
        r2.setSide('W',theDoor);

        return aMaze;
        }
```

Como se puede observar, ya no se definen explícitamente los nombres de las clases de los objetos a crear, por lo que se pueden crear laberintos compuestos de distintos componentes. Por ejemplo, podemos crear la clase *EnchantedMazeFactory* al heredar de la clase MazeFactory. Esta nueva clase sobrescribirá los métodos de la clase padre y retornará objetos específicos de subclases de *Room* y *Door*.

```
class EnchantedMazeFactory extends MazeFactory {
        public EnchantedMazeFactory() {
                super();
                }
        /* sobre escribe solo los metodos específicos para esta subclase */
        public final Room makeRoom(int _roomNo) {
                return new EnchantedRoom(_roomNo);
                }
        public final Door makeDoor(Room _r1, Room _r2) {
                return new DoorNeedingSpell(_r1,_r2);
                }
        }
```

La clase *MazeFactory* es simplemente una colección de *FactoryMethods* con una implementación *default*, por lo que es muy fácil crear subclases de ella y sobrescribir los métodos necesarios, como lo hace la clase *EnchantedMazeFactory* en el bloque de código anterior. Además, como la clase *MazeFactory* no es abstracta, esta clase actúa como las clases *AbstractFactory* y *ConcreteFactory* al mismo tiempo.

```
MazeFactory factory =new MazeFactory();
cliente.createMaze(factory);

MazeFactory enchantedFactory =new EnchantedMazeFactory();
cliente2.createMaze(enchantedFactory);
```

Tomar como parámetro una instancia de *MazeFactory* o de una de sus subclases, nos permite crear laberintos de distintos tipos de una manera más flexible.

Usos conocidos

Este patrón es utilizado para definir fábricas abstractas para generar objetos de interfaces de usuario específicas. También es utilizado por algunos *frameworks* de GUI para lograr portabilidad a través de distintos sistemas de ventanas, como Xwindow y SunView.

Patrones Relacionados

Las clases *AbstractFactory* son implementadas con *FactoryMethods*, pero también pueden ser implementadas utilizando el patrón *Prototype*. Y por lo regular, una clase fabrica concreta (*ConcreteFactory*) es implementada mediante el patrón *Singleton*.

2.4.3 *Builder*

Intención

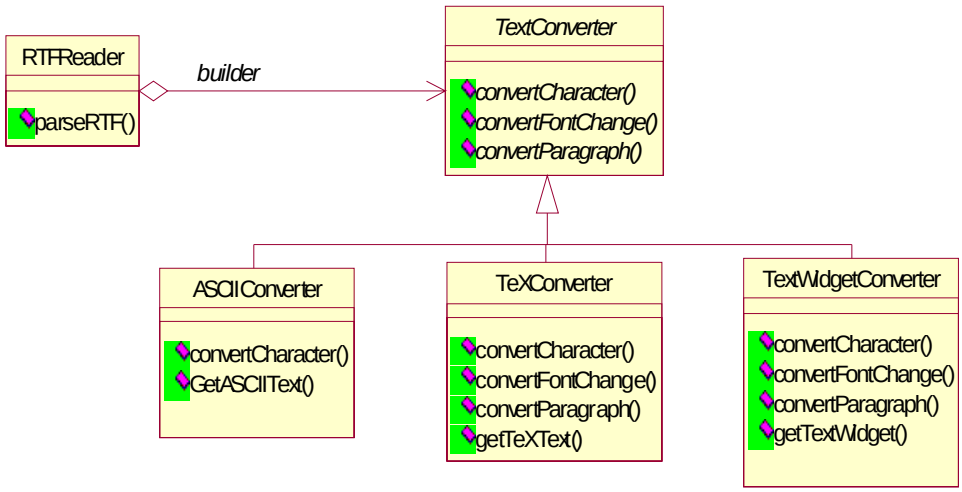
Separar la construcción de un objeto complejo de su representación, para que el mismo proceso sirva para crear diferentes representaciones.

Motivación

Una clase que lee documentos RTF[21] deberá permitir la conversión de estos documentos a distintos formatos de texto, como a un Ascii. Debido a que el número de conversiones posibles es ilimitado, la aplicación debe permitir agregar fácilmente un nuevo tipo de conversión sin modificar la clase que maneja los documentos, que en la figura 15 es representada como *RTFReader*. Esto se logra con un objeto encargado de realizar la conversión de RTF hacia cualquier otro formato de texto. La clase abstracta

TextConverter define la interfaz para manejar la conversión a distintos formatos específicos mediante objetos especializados en cada conversión.

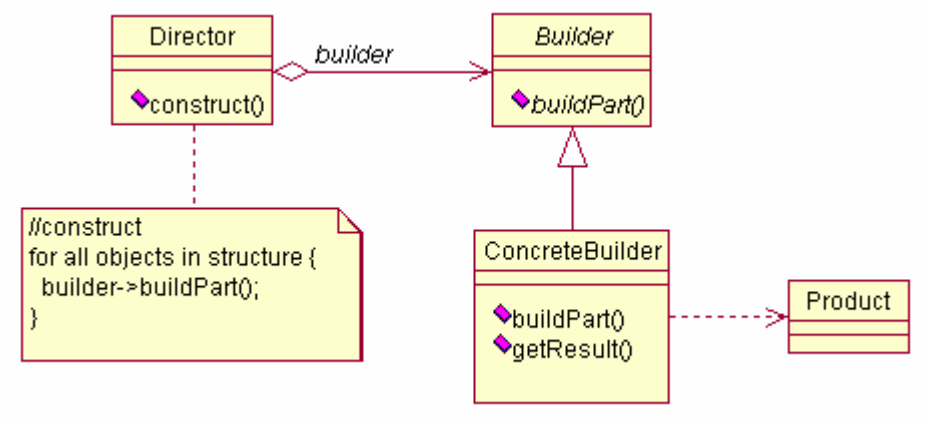
Figura 15. Ejemplo de patrón *builder*



Aplicabilidad

Se debe utilizar este patrón cuando el algoritmo para crear un objeto complejo debe ser independiente de las partes que componen el objeto y de la forma en que son ensamblados, y también cuando el proceso de construcción debe permitir diferentes representaciones del objeto que es construido.

Figura 16. Estructura de patrón *builder*



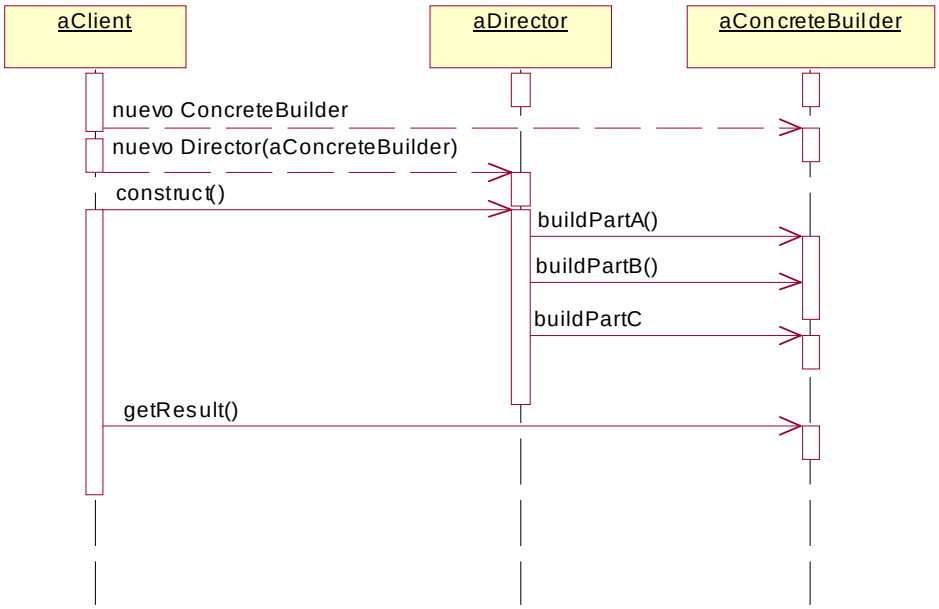
Participantes

- *Builder*. Específica la interfaz abstracta para crear partes de un objeto *Product*
- *ConcreteBuilder*. Construye la representación interna de *Product* y define el proceso por el cual es ensamblado, implementando la interfaz definida por *Builder*. También provee una interfaz para obtener el objeto que crea.
- *Director*. Construye un objeto utilizando la interfaz *Builder*.
- *Product*. Representa el objeto complejo en construcción.

Colaboraciones

El siguiente diagrama de secuencia muestra como *builder* y *director* interactúan con un cliente:

Figura 17. Colaboración entre objetos participantes de *builder*



Consecuencias

- Permite variar la representación interna de un objeto complejo fácilmente.
- Separa el código para la construcción y la representación del objeto complejo.
- Proporciona un mejor control sobre el proceso de construcción. A diferencia de otros patrones de creación que construyen productos en un solo paso, *Builder* proporciona un proceso para construir un objeto complejo paso a paso, bajo el control de una clase Directora. Una vez que el producto está ensamblado, la clase directora obtiene el objeto complejo o producto del constructor.

Implementación

Definir una interfaz de ensamblado y construcción. Las clases concretas de *Builder* construyen los productos paso a paso, por lo que la interfaz definida por la clase abstracta *Builder* debe ser muy general para permitir la creación de productos de cualquier clase de *Builders* concretos. Los métodos *build* en la clase *builder* deben ser abstractos, para forzar a los clientes a sobrescribir los métodos en los que están interesados.

Los productos ensamblados o producidos por los constructores concretos (*ConcreteBuilder*) difieren en gran manera en su representación, que no se ganaría mucho definiendo una clase abstracta para las distintas representaciones de los productos.

Muestra de código y utilización

Para ejemplificar este patrón, cambiaremos el método *createMaze* definido en la sección 2.4. En esta variante, el método *createMaze* recibirá como argumento una instancia de la clase *MazeBuilder*, la cual está definida de la siguiente manera:

```
class MazeBuilder {
        public MazeBuilder() {}
        public void buildMaze() {}
        public void buildRoom(int _room) {}
        public void buildDoor(int _roomFrom, int _roomTo) {}
        public Maze getMaze() {}
        }
```

Esta interfaz permite crear un tres tipos de objetos: *Maze, Rooms*, y *Doors*. El método *getMaze*() retorna un objeto *Maze*, el cual es el producto complejo ensamblado. El método *createMaze* quedará de la siguiente manera:

```
public Maze createMaze(MazeBuilder _builder) {
        _builder.buildMaze();
        _builder.buildRoom(1);
        _builder.buildRoom(2);
        _builder.buildDoor(1,2);
        return _builder.getMaze();
        }
```

MazeBuilder es la interfaz para crear los laberintos o *Mazes*, pero no es ella quien los crea. Estos deben ser creados por una subclase que implemente sus métodos, la cual definimos como *StandardMazeBuilder*.

```
class StandardMazeBuilder extends MazeBuilder{
        private Maze currentMaze;

        public StandardMazeBuilder() {
                currentMaze =null;
                }
        public void buildMaze() {
                currentMaze =new Maze();
                }
        public void buildRoom(int _room) {
                if (currentMaze.roomNo(_room) !=null) {
                        Room room =new Room(_room);
                        currentMaze.addRoom(room);
                        room.setSide('N',new Wall());
                        room.setSide('S',new Wall());
                        room.setSide('E',new Wall());
                        room.setSide('W',new Wall());
                        }
                }
        public void buildDoor(int _roomFrom, int _roomTo) {
                Door d =new Door(currentMaze.roomNo(_roomFrom),currentMaze.roomNo(_roomTo));

                currentMaze.roomNo(_roomFrom).setSide(commonWall(currentMaze.roomNo(_roomFrom),curr
                entMaze.roomNo(_roomTo)),d);

                currentMaze.roomNo(_roomTo).setSide(commonWall(currentMaze.roomNo(_roomTo),currentM
                aze.roomNo(_roomFrom)),d);
                }
        public Maze getMaze() {
                return currentMaze;
                }
        private char commonWall(Room _r1, Room _r2) {
                /*      metodo para determinar la direccion de la
                        pared en comun entre los dos cuartos _r1, _r2. ('N','S','E',"W')*/
                return ' ';
                }
        }
```

Ahora un cliente puede utilizar el método *createMaze* junto con una instancia de *StandardMazeBuilder* para crear un *Maze* de la siguiente manera:

```
Maze maze;
MazeGameBuilder game =new MazeGameBuilder();
StandardMazeBuilder builder =new StandardMazeBuilder();

game.createMaze(builder);
maze =builder.getMaze();
```

Usos conocidos

Utilizado por *parsers* de compiladores para crear el árbol sintáctico, nodo por nodo y luego retornar el resultado final al compilador, que es la estructura completa. También es un patrón muy utilizado en Smalltalk.

Patrones relacionados

Abstract factory es similar a *builder*, ya que ambos permiten la construcción de objetos complejos, pero *builder* se enfoca en la construcción de estos objetos paso a paso y retorna el producto en un paso final, mientras que *abstract factory* se enfoca en familias de objetos (simples o complejos) y el producto es retornado de manera inmediata.

2.4.4 *Prototype*

Intención

Especificar las clases de objetos a crear y utilizar una instancia prototipo, y crear nuevos objetos copiando o clonando este prototipo.

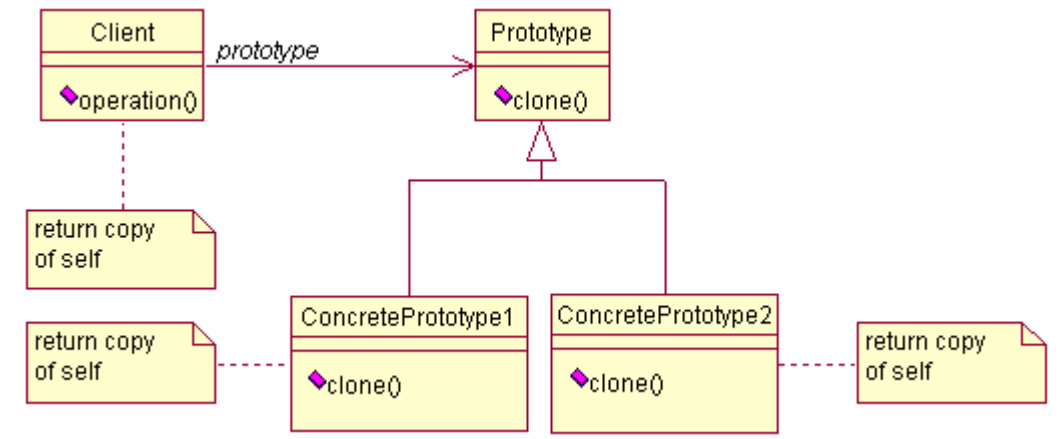
Motivación

Permitir la clonación de una instancia de una clase concreta. Esta instancia es llamada prototipo. Este patrón reduce el número de clases en el sistema y hace más fácil agregar nuevas clases en el futuro.

Aplicabilidad

Este patrón debe utilizarse cuando el sistema debe ser independiente de la forma en que sus productos son creados, ensamblados y representados, cuando las clases a instanciar son especificadas dinámicamente. En algunos casos es más conveniente y eficiente instalar un número reducido de prototipos y clonarlos, en lugar de instanciar cada clase manualmente cada vez que es necesario.

Figura 18. Estructura patrón *prototype*



Participantes

- *Prototype* declara la interfaz para permitir la clonación
- *ConcretePrototype* implementa la operación de clonación

- *Client* crea un nuevo objeto solicitando a *Prototype* que se clone a sí mismo

Colaboraciones

Un cliente solicita a un prototipo que se clone a sí mismo.

Consecuencias

Este patrón posee algunas consecuencias similares a las de *abstract factory* y *builder*: Mantiene separadas las clases concretas de los clientes, separa el código para la construcción y representación de un objeto y proporciona un mejor control sobre la construcción de un objeto. Además presenta los siguientes beneficios:

- Permite agregar y eliminar productos de forma dinámica.
- Definir nuevos objetos variando valores de variables. Un sistema altamente dinámico permite definir nuevos comportamientos a través de la composición de objetos, especificando nuevos valores para las variables de los objetos en lugar de definir nuevas clases.
- Permite clonar un prototipo en lugar de solicitar a un método la creación de un nuevo objeto.

Cuando se clona un objeto, no se llama al constructor del objeto clonado, por lo que puede ser necesario inicializar el clon.

La principal responsabilidad de este patrón es que cada subclase de *prototype* implemente el método *clone*.

Implementación

Cuando el número de prototipos en un sistema no es fijo, se debe utilizar un administrador de prototipos, el cual debe llevar un registro de los prototipos disponibles en el sistema.

La implementación del método clone no es una tarea fácil y es una acción potencialmente peligrosa, ya que puede ocasionar efectos colaterales no deseados, por ejemplo, si un objeto abre un flujo de e/s y es clonado, habrá dos objetos capaces de operar sobre el mismo flujo, y si uno de ellos cierra el flujo, el otro podría intentar escribir en él causando un error, por lo que es muy importante definir de manera correcta los prototipos a utilizar en el sistema. En Java, el método clone definido por *Object* como protegido, genera una copia bit a bit del objeto. Este método puede ser llamado desde un método definido por una clase que implemente la interfaz *Cloneable*, o bien ser sobrescrito por esa clase para que sea publico.

Muestra de código y utilización

En este ejemplo, definimos *MazePrototypeFactory*, una subclase de *MazeFactory* definida en el patrón *Abstract Factory* (Sección 2.4.2). Esta clase será inicializada con prototipos de los objetos que creará, los cuales serán pasados como argumentos en el constructor.

```
class MazePrototypeFactory extends MazeFactory{
        private Maze prototypeMaze;
        private Wall prototypeWall;
        private Room prototypeRoom;
        private Door prototypeDoor;

        public MazePrototypeFactory(Maze _m, Wall _w, Room _r, Door _d) {
                prototypeMaze =_m;
                prototypeWall =_w;
                prototypeRoom =_r;
                prototypeDoor =_d;
                }
        public Maze makeMaze() {
                return (Maze) prototypeMaze.clone();
```

```
		}
	public Wall makeWall() {
		return (Wall) prototypeWall.clone();
		}
	public Room makeRoom(int _roomNo) {
		Room aRoom = (Room) prototypeRoom.clone();
		aRoom.initialize(_roomNo);
		return aRoom;
		}
	public Door makeDoor(Room _r1, Room _r2) {
		Door aDoor = (Door) prototypeDoor.clone();
		aDoor.initialize(_r1,_r2);
		return aDoor;
		}
	}
```

Para poder utilizar el método *clone*, las clases *MapSite* y *Maze* implementan la interfaz *Cloneable* de la siguiente manera:

```
class MapSite implements Cloneable{
	public void enter() {}
	public final Object clone() {
		try {
			return super.clone();
			}
		catch (CloneNotSupportedException e) {
			//manejo de la excepcion,
			return null;
			}
		}
	}
class Maze implements Cloneable{
	public Maze() {}
	public void addRoom(Room _room) {}
	public final Room roomNo(int _roomNo) {
		return null;
		}
	public final Object clone() {
		try {
			return super.clone();
			}
		catch (CloneNotSupportedException e) {
			//manejo de la excepcion,
			return null;
			}
		}
	}
```

Ahora utilizamos la clase *MazePrototypeFactory* para obtener un prototipo de un *Maze* creando una instancia de ella, de la siguiente manera:

```
MazeGamePrototype game =new MazeGamePrototype();
MazeFactory simpleFactory =new MazePrototypeFactory(new Maze(),new Wall(), new Room(), new
Door());
game.createMaze(simpleFactory);
```

Patrones relacionados

Prototype y *Abstract Factory* se encuentran muy relacionados. Los diseños que utilizan los patrones *Composite* y *Decorador* por lo regular se benefician del uso de *Prototype*.

2.4.5 *Singleton*

Intención

Lograr que una clase posea solo una instancia y proveer un único punto de acceso global a ella.

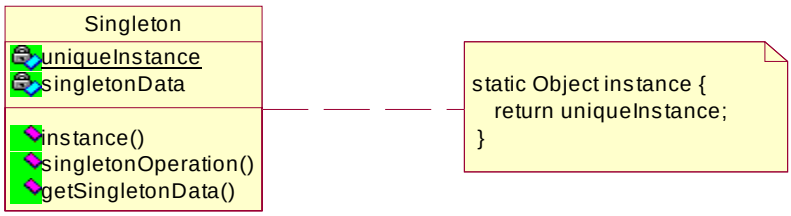
Motivación

Para algunas clases es importante tener exactamente una instancia. Una solución es hacer a la misma clase responsable de llevar un registro de su única instancia, asegurándose que no se pueda crear otra instancia de ella, y proveer un único punto de acceso a dicha instancia.

Aplicabilidad

Este patrón debe utilizarse cuando debe existir una única instancia de una clase y debe ser accedida por los clientes mediante un punto de acceso en común.

Figura 19. Estructura patrón *Singleton*



Participantes

Singleton: define un método para que los clientes accedan a su única instancia. El método *instance*() es un método de clase (*static*). También es responsable de crear su única instancia.

Colaboraciones

Los clientes acceden a la instancia única solo a través de el método *instance*().

Consecuencias

Provee un acceso controlado a la instancia de la clase. Este patrón mejora el uso de espacio en memoria, en lugar de variables globales.

Implementación

Para asegurarse que solo una instancia es creada, se debe ocultar el método que crea la instancia detrás de un método de clase, el cual garantiza que solo una instancia ha sido creada.

Muestra de código y utilización

En este ejemplo utilizaremos la clase *MazeFactory* definida en la sección 2.4.2, *Abstract factory*. Suponiendo que la aplicación necesita una sola instancia de *MazeFactory* para crear los laberintos, implementamos la clase *MazeFactory* como un *singleton*, de la siguiente manera:

```
class MazeFactory {
        private static MazeFactory instance;

        public static MazeFactory MazeFactory() {
                if (instance ==null)
                        instance = new MazeFactory();
                return instance;
                }
        public Maze makeMaze() {
                return new Maze();
                }
        public Wall makeWall() {
                return new Wall();
                }
        public Room makeRoom(int _roomNo) {
                return new Room(_roomNo);
                }
        public Door makeDoor(Room _r1, Room _r2) {
                return new Door(_r1,_r2);
                }
        }
```

En el constructor, si la instancia ya ha sido creada, devuelve la referencia hacia esa instancia, y si no ha sido creada se crea en ese instante.

Patrones relacionados

Muchos patrones pueden ser implementados utilizando este patrón, como por ejemplo *Abstract factory*, *builder* y *prototype*.

2.5 Patrones estructurales

Estos patrones se enfocan en la forma en que las clases y objetos se relacionan para formar estructuras más grandes o complejas. Los patrones estructurales de clase utilizan la herencia para formar interfaces o

implementaciones. Los patrones estructurales de objetos utilizan la composición de objetos para lograr nueva funcionalidad. Este enfoque presenta una mayor flexibilidad ya que se puede modificar las relaciones de los objetos en tiempo de corrida, lo que no es posible mediante la herencia.

2.5.1 *Adapter*

Intención

Convertir la interfaz de una clase en otra compatible con las necesidades del cliente. Tambien es conocida como *wrapper.*

Motivación

En ocasiones un *toolkit* que ha sido diseñado para ser reutilizado no se puede reutilizar solo por incompatibilidad con la interfaz requerida por una aplicación específica. Por ejemplo, tenemos un editor gráfico que permite dibujar y acomodar elementos gráficos (líneas, polígonos, texto, etc) en imágenes y diagramas.

La abstracción clave en este diseño es el objeto gráfico, que posee una forma que se puede editar y se puede dibujar a sí mismo. La interfaz para los objetos gráficos esta definida por una clase abstracta llamada *Shape*. El editor define una subclase de *Shape* para cada clase de objeto gráfico: *LineShape* para líneas, *PolygonShape* para polígonos, entre otras. Estas clases son fáciles de implementar debido a sus características, pero una subclase *TextShape* para desplegar y editar texto no es fácil de implementar. Ahora, suponiendo que encontramos un *toolkit* que provee una clase sofisticada llamada *TextView* para editar y desplegar texto gráficamente, podriamos reutilizar *TextView* para implementar *TextShape*,

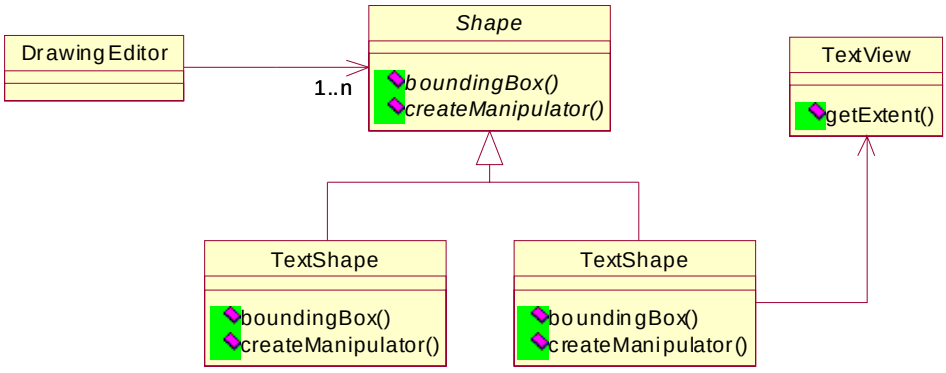
pero el *toolkit* no fue diseñado para ser utilizado con la clase *Shape*, por lo que no podemos utilizar los objetos *TextView* y *Shape* de la misma manera (polimorfismo).

La solución consiste en definir a *TextShape* de una forma que adapte la interfaz de *TextView* en *Shape*. Esto se puede hacer mediante dos formas:

1. Implementando la interfaz *Shape* y heredando la implementación de *TextView.*
2. Mediante composición de objetos: definir una instancia de *TextView* dentro de *TextShape* e implementar a *TextShape* en terminos de la interfaz de *TextView*.

Estos dos enfoque corresponden a las dos versiones del patrón adaptador: de clase y de objeto. A la clase *TextShape* se le llama adaptadora. En la siguiente figura se presenta el caso de el patrón adaptador de objeto:

Figura 20. Ejemplo adaptador de objeto



Por lo regular, la clase adaptadora es responsable por la funcionalidad que la clase adaptada no posee.

Aplicabilidad

- Utilizar una clase existente cuya interfaz no corresponde con las necesidades específicas de la aplicación.
- Crear una clase reutilizable que coopere con clases imprevistas, es decir, con interfaces incompatibles.
- Para utilizar varias subclases existentes, en las cuales no sería práctico adaptar sus interfaces mediante herencia para cada una. Un adaptador de objeto puede adaptar la interfaz de su clase padre.

Figura 21. Estructura adaptador de clase

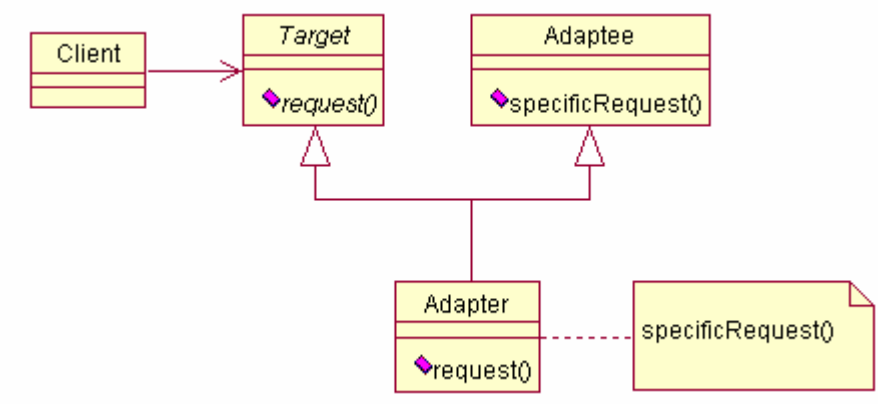
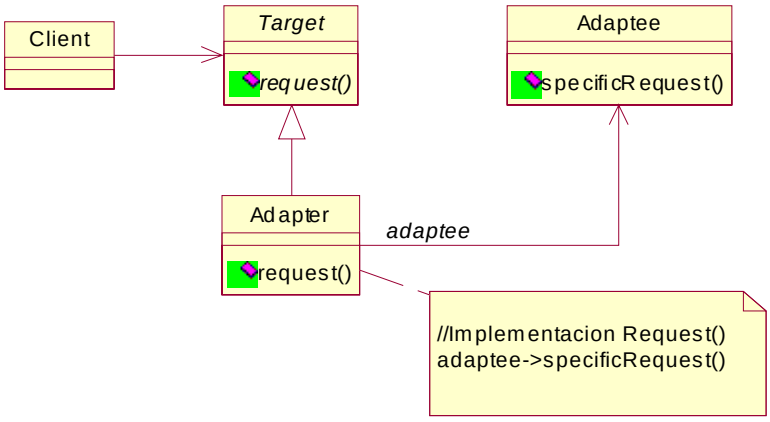


Figura 22. Estructura adaptador de objeto



Participantes

- *Target*. Define la interfaz específica que el cliente utiliza.
- *Client*. Maneja los objetos conforme a la interfaz de *Target*.
- *Adaptee*. Define una interfaz que necesita ser adaptada.
- *Adapter*. Adapta la interfaz de *Adaptee* hacia la interfaz de *Target*.

Colaboraciones

Los clientes invocan los métodos de la instancia adaptadora, la cual invoca los métodos en la instancia adaptada, que es quien realiza la petición invocada por el cliente.

Consecuencias

Las consecuencias entre adaptadores de clase y de objeto son diferentes. Un adaptador de clase puede sobrescribir los métodos de la clase adaptada (*Adaptee*) ya que esta es su clase padre. Un adaptador de objeto permite a un solo adaptador trabajar con múltiples objetos adaptados, es decir, la clase adaptada y todas sus subclases. Con un adaptador de objeto es más difícil sobrescribir el comportamiento del adaptado, ya que se requiere heredar de la clase adaptada y hacer que el adaptador haga referencia a las subclases en vez de a *Adaptee*.

El trabajo de un adaptador va desde una simple conversión de interfaces, como por ejemplo cambiar el nombre de los métodos hasta soportar un conjunto completamente diferente de operaciones o métodos. El grado de trabajo que el adaptador realice dependerá de qué tan similares son las interfaces de la clase objetivo (*Target*) y la clase adaptada (*Adaptee*).

Muestra de código y utilización

Para demostrar el uso de este patrón, utilizaremos el ejemplo descrito al inicio, en la sección de motivación. Primero definimos las clases objetivo y adaptadas:

```
interface Shape {
        public void Shape() ;
        public void boundingBox (Point bottomLeft, Point bottomRight);
        public Manipulator createManipulator();
        }

class TextView {
        public void TextView() {}
        public void getOrigin(Integer x, Integer y) {}
        public void getExtent(Integer width, Integer height) {}
        public boolean isEmpty() {
                return true;
                }
        }
```

La interfaz *Shape* define un área limitada por sus esquinas opuestas, mientras que *TextView* define coordenadas origen, alto y ancho. La interfaz *Shape* también define el método *createManipulator*(), para crear un objeto *manipulator*, el cual se encarga de animar el objeto cuando el usuario lo esta manipulando. La clase *TextView* no tiene un método equivalente. La clase *TextShape* es la adaptadora entre estas dos clases:

```
class TextShape extends TextView implements Shape {
        public void Shape() {}
        public void boundingBox (Point bottomLeft, Point bottomRight) {
                Integer bottom, left, width, height;
                bottom =new Integer(0);
                left =new Integer(0);
                width =new Integer(0);
                height =new Integer(0);
                getOrigin(bottom,left);
                getExtent(width,height);
                bottomLeft.setXY(bottom,left);
                bottomRight.setXY(bottom, left);
                }
        public Manipulator createManipulator() {
                return null;
                }
        public boolean isEmpty() {
                return super.isEmpty();
                }
        public void TextShape() {}
        }
```

Un adaptador de clase utiliza herencia múltiple. Java no permite realizar herencia múltiple, pero sí permite implementar más de una interfaz, por lo que la clase *TextShape* hereda la funcionalidad de *TextView* e implementa la interfaz definida por *Shape*. Esto es un aspecto clave en el patrón adaptador de clase: utilizar una rama de herencia para la interfaz y otra rama para la implementación.

El adaptador de objeto utiliza composición de objetos para combinar las clases con distintas interfaces. En este enfoque, la clase adaptadora *TextShape* debe mantener un puntero hacia una instancia de *TextView*.

Patrones relacionados

El patrón *Bridge* posee una estructura similar a un adaptador de objeto, pero este patrón tiene otra intención: separar una interfaz de su implementación para que puedan ser cambiadas fácilmente de manera independiente.

2.5.2 *Bridge*

Intención

Separar una abstracción de su implementación para que ambas puedan variar independientemente. Es conocido como handlebody.

Motivación

Cuando una abstracción posee una de varias implementaciones, la forma normal de manejarlo es utilizando herencia. Una clase abstracta define la interfaz para la abstracción y varias subclases concretas

implementan la interfaz en diferentes formas. Este enfoque no es siempre lo suficientemente flexible, ya que la herencia enlaza la implementación con la interfaz de forma permanente, lo cual hace difícil la reutilización de la abstracción y la implementación de manera independiente.

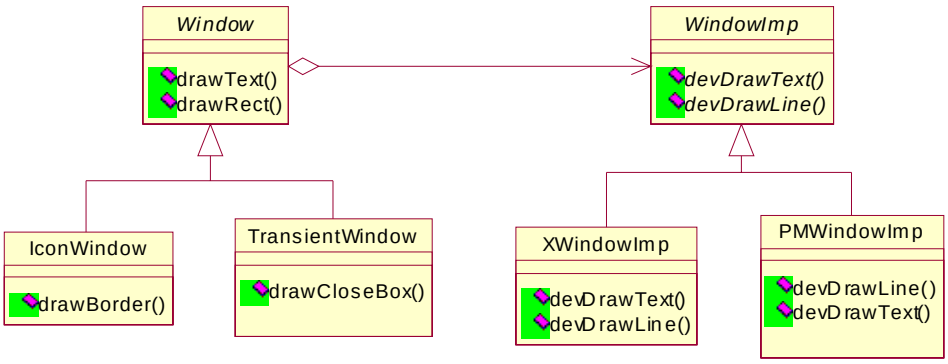
Por ejemplo, consideremos la implementación de una abstracción de ventanas en un *toolkit* para manejo de interfaces de usuario. Esta abstracción nos permitiría escribir aplicaciones multiplataforma, digamos el sistema *XWindow* y *Presentation manager* (IBM). A través de la herencia podemos definir una clase abstracta *Window*, y sus subclases *XWindow* y *PMWindow* que implementen la interfaz definida por *Window* para diferentes plataformas. Hasta ahora, todo parece funcionar bien, pero existen dos inconvenientes en este enfoque:

1. No es recomendable extender de *Window* para manejar los distintos objetos específicos de cada plataforma, ya que se crearía una jerarquía extensa de clases que sería difícil de entender y manejar.

2. El código generado es dependiente de la plataforma, ya que cuando un cliente crea una ventana, ésta es instanciada directamente de la clase concreta que posee su implementación.

El patrón *Bridge* soluciona estos problemas colocando la abstracción de la ventana y su implementación en jerarquías de clase separadas. Existe una jerarquía de clases para las interfaces de las ventanas y una jerarquía separada para las implementaciones específicas para cada plataforma.

Figura 23. Ejemplo patrón *Bridge*

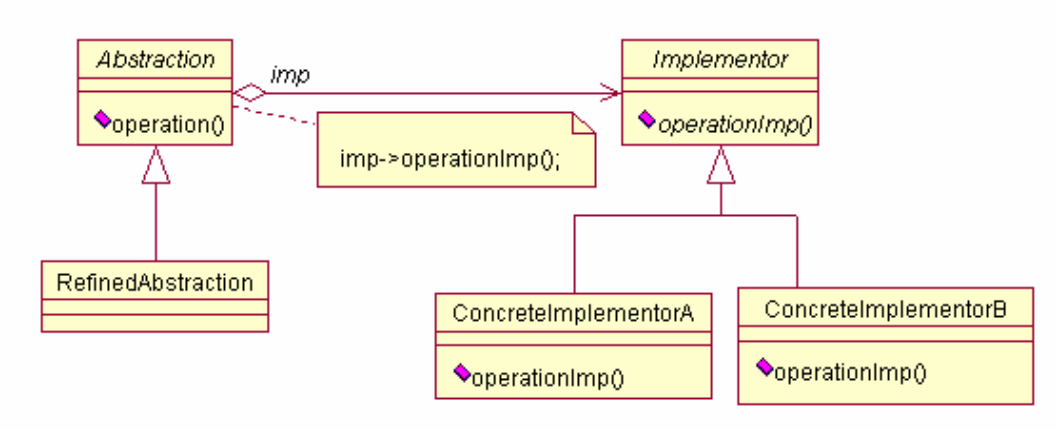


La relación entre *Window* y *WindowImp* es llamada *Bridge*, ya que crea una conexión entre la abstracción y la implementación, permitiendo a ambas cambiar de manera independiente.

Aplicabilidad

- Permite evitar enlaces permanentes entre una abstracción y su implementación. Esto es muy útil cuando la implementación debe seleccionarse o cambiarse dinámicamente.
- Cuando la abstracción y la implementación son susceptibles de extensión mediante herencia.
- Cuando necesitamos que los cambios en la implementación de una interfaz no provoquen impacto en el cliente, es decir, que no sea necesario recompilar el código.
- Cuando queremos compartir una implementación entre múltiples objetos, ocultando esto a los clientes.

Figura 24. Estructura patrón *Bridge*



Participantes

- *Abstraction*. Define la interfaz y mantiene una referencia a una instancia de Implementor.
- *RefinedAbstraction*. Extiende la interfaz definida por *Abstraction*.
- *Implementor*. Define la interfaz para las clases de implementación. Típicamente esta interfaz provee solo operaciones o métodos básicos mientras que *Abstraction* define métodos más completos, basados en los métodos definidos por *Implementor*.

Colaboraciones

Abstraction envía las peticiones de los clientes hacia un objeto *Implementor*.

Consecuencias

- Separar una interfaz de su implementación, para no crear una relación permanente entre ambas, con lo cual, una implementación puede ser configurada hacia una interfaz de forma dinámica.

- Se mejora la extensibilidad de las clases, es decir, se puede extender la jerarquía de implementación e interfaz de forma independiente.
- Se ocultan detalles de la implementación a los clientes.

Implementación

- Cuando solo se tiene una implementación para una interfaz podría no ser necesario crear una clase abstracta para la implementación. Esta situación es un caso especial del patrón *Bridge*, donde existe una relación de uno a uno entre la abstracción y la implementación. Sin embargo, esta división es útil cuando un cambio en la implementación no debe afectar a los clientes existentes, por ejemplo, recompilando código.

- Creación del implementador correcto. Cuando se tiene más de un implementador, la decisión de cúal utilizar se puede tomar de las siguientes formas:

    - Si la interfaz conoce todos las clases implementadoras concretas, entonces en su constructor puede decidir qué clase crear en base a parámetros recibidos en su constructor. Por ejemplo, si tenemos un arreglo de clases que soporta múltiples implementaciones, la decisión de qué implementación utilizaremos se basará en el tamaño del arreglo: si es un arreglo pequeño utilizamos una implementación de pila y si es un arreglo grande utilizamos una implementación de tablas de hash.

    - Definir una implementación por *default* inicialmente, y cambiarla de acuerdo a su utilización.

Muestra de código y utilización

Utilizaremos el ejemplo definido en la sección motivación del presente patrón. A continuación se definen las clases abstractas para *Window* y para *WindowImp*.

```
abstract class WindowImp {
        protected void WindowImp() {}

        public void impTop() {}
        public void impBottom() {}
        public void impSetExtent(Point _p) {}
        public void impSetOrigin(Point _p) {}
        public void deviceRect(int x0,int y0,int x1,int y1) {}
        }

abstract class Window {
        private WindowImp imp;
        private View contents;

        public Window(View contents) {}
        //metodos manejados por Window
        public void open() {}
        public void close() {}
        //metodos manejados por la implementacion
        public void setOrigin(Point at) {}
        public void setExtent(Point extent) {}
        public void raise() {}
        public void lower() {}

        public void drawLine(Point _ini, Point _fin) {}
        public void drawRect(Point _ini, Point _fin) {}
        public void drawPolygon(Point[] _p, int n) {}
        public void drawText(char _c, Point _p) {}
        //
        protected WindowImp getWindowImp() {
                return null;
                }
        protected View getView() {
                return null;
                }
        }
```

Como se puede observar, la definición de *WindowImp* comprende métodos básicos, los cuales serán utilizados para implementar métodos más complejos en la interfaz de la abstracción. La clase *Window* tiene una referencia a un objeto *WindowImp*, que es la interfaz que servirá para manejar las implementaciones específicas de las ventanas, dependiendo de la plataforma.

Patrones relacionados

Utilizando el patrón *Abstract factory* se puede crear y configurar un *Bridge*.

2.5.3 *Composite*

Intención

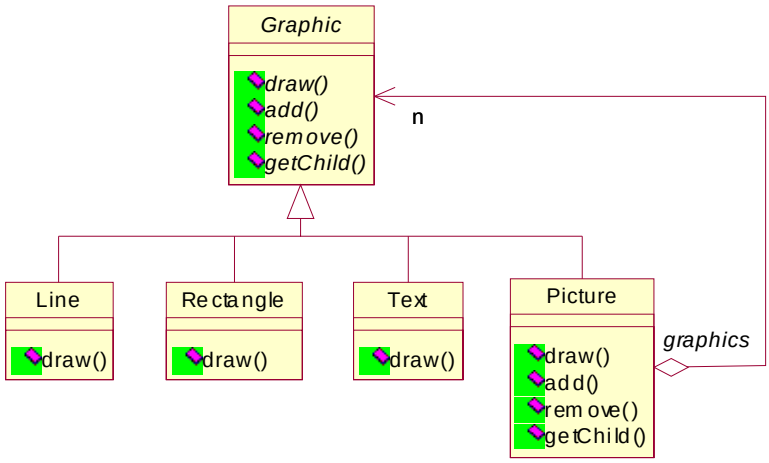
Permite el manejo de objetos individuales y compuestos de manera uniforme.

Motivación

Algunas aplicaciones permiten construir diagramas complejos en base a componentes simples, donde el usuario puede agrupar los componentes para formar componentes mayores y más complejos, los cuales también pueden agruparse para formar componentes aun más grandes y complejos.

Por ejemplo, un editor gráfico podría definir clases elementales como *Text* y *Line* (para texto y para líneas), y también definir otras clases que funcionen como contenedores para las clases elementales. El problema con esta solución es que el código debe tratar de manera diferente a las instancias de las clases elementales y a las de las clases contenedoras. Manejar de forma diferente estos objetos puede generar aplicaciones más complejas. El patrón *Composite* describe una solución, utilizando composición de objetos de forma recursiva, para que los clientes no tengan que hacer esta distinción.

Figura 25. Ejemplo patrón *Composite*

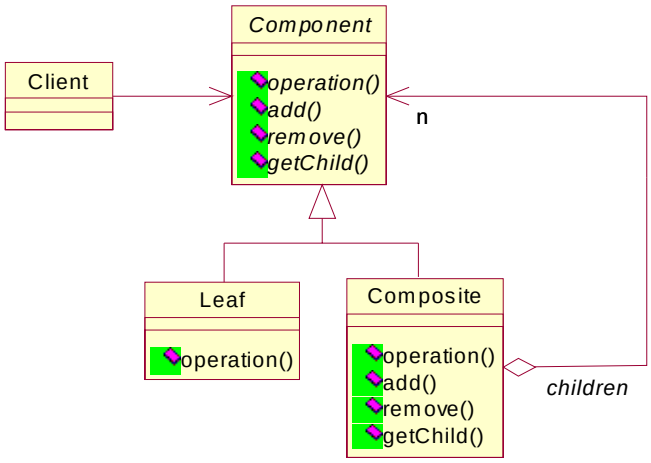


Un aspecto clave en la aplicación de este patrón es la identificación de una clase abstracta que represente a las clases elementales y a las clases contenedoras.

Aplicabilidad

- Cuando se quiere representar jerarquías de objetos compuestas.
- Cuando se quiere que los clientes manejen de igual forma los objetos compuestos y los individuales.

Figura 26. Estructura patrón *Composite*



Participantes

- *Component.*
  - Declara la interfaz para las clases hijas e implementa una funcionalidad *default*.
  - Declara la interfaz para acceder y manipular a los componentes hijos.
  - Puede definir una interfaz para acceder al componente padre, aunque esta interfaz no es obligatoria.
- *Leaf*. Representa un objeto hoja dentro de la composición de objetos. Un objeto hoja no tiene hijos. Define una funcionalidad específica para cada objeto en la composición.
- *Composite*. Define la funcionalidad para los componentes que tienen componentes hijos. Se encarga de almacenar los objetos hijos e implementar métodos para su manipulación.
- *Client*. Maneja los objetos de la composición a través de la interfaz definida por *Component*.

Colaboraciones

Los clientes utilizan la interfaz definida por *Component* para interactuar con los objetos definidos.

Consecuencias

- Define jerarquías de clases compuestas de objetos individuales o elementales y de objetos compuestos.
- Facilita la manipulación de los objetos a los clientes, ya que estos pueden manejar de igual manera objetos compuestos y objetos individuales.
- Facilita agregar nuevos objetos a la estructura.
- Una desventaja de facilitar la agregación de nuevos objetos es que se puede tener un diseño demasiado general, lo que hace difícil la restricción de los componentes que pueden pertenecer a una estructura.
- El performance puede verse afectado de manera negativa, al incrementarse los niveles de recursividad.

Implementación

Mantener una referencia de un componente hijo hacia su componente padre puede facilitar el recorrido y manipulación de una estructura compuesta. El mejor lugar para definir la referencia hacia el padre es la clase *Component*, para que las clases *Leaf* y *Composite* hereden la referencia y los métodos para manipularla.

Uno de los objetivos del patrón *Composite* es mantener la uniformidad de manejo de los objetos simples y los compuestos. Para lograrlo, se debe

definir en la clase *Component* la mayor cantidad de operaciones comunes para las clases *Leaf* y *Composite*.

Un objeto compuesto (*Composite*) es responsable de eliminar sus objetos hijos cuando es destruido, a menos que se cuente con recolector de basura (*Garbage colllector*[22]) proporcionado por el lenguaje de programación. Java proporciona un recolector de basura en la clase *Runtime*[23].

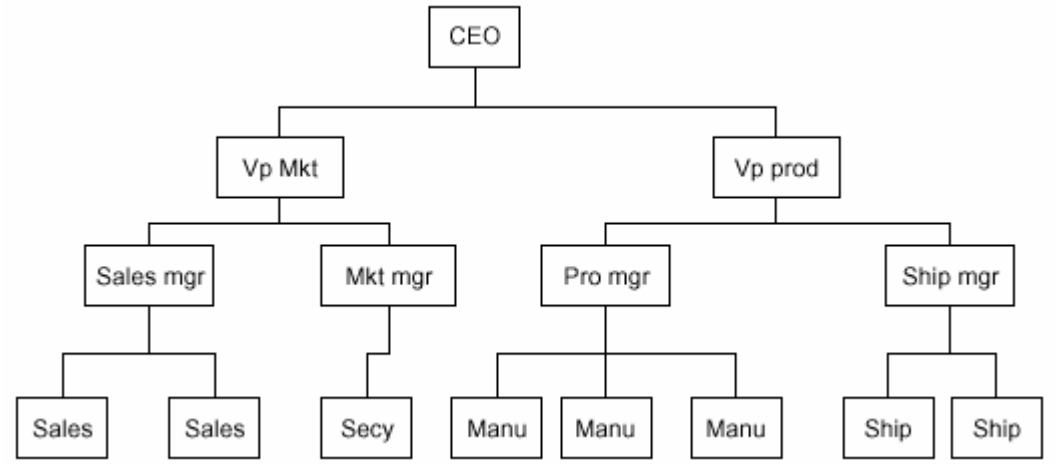
Los objetos compuestos (*Composite*) pueden utilizar distintas estructuras de datos para almacenar sus hijos: listas, árboles, tablas de *hash*, entre otras. La elección de una estructura de datos dependerá de el tipo de aplicación y la eficiencia que representa cada una.

Muestra de código y utilización

Podemos considerar el caso de crear el organigrama de una empresa. En un organigrama se representa la posición que ocupa el empleado dentro de la jerarquía de la organización. Cada empleado recibe un salario, el cual puede ser consultado en cualquier momento para cualquier empleado representado en el organigrama.

Al consultar el costo de un empleo, el resultado debe ser: el salario del empleado que se consulta más la suma de salarios de cada empleado subordinado.

Figura 27. Organigrama



Por ejemplo, el costo del departamento de *marketing* (VP Mkt, vicepresidente *marketing*) es la suma del salario de VP Mkt y todos los nodos subordinados a el (Sales mgr, Mkt mgr, Sales, Secy). Y el costo en recursos humanos de toda la empresa se obtiene consultando el costo del CEO y sus subordinados. Para lograr lo anterior, debemos definir una clase empleado que implemente los métodos necesarios para manipular la información de salarios y puestos dentro de la empresa.

```
public class Employee {
        String name;
        float salary;
        Vector subordinates;

        public Employee(String _name, float _salary) {
                name = _name;
                salary = _salary;
                subordinates = new Vector();
                }
        public float getSalary()        { return salary; }
        public String getName()         { return name; }
        public void add(Employee e)   { subordinates.addElement(e); }
        public void remove(Employee e) { subordinates.removeElement(e); }
        public float getSalaries() {
                float sum = salary;
                for(int i = 0; i < subordinates.size(); i++) {
                sum +=((Employee)subordinates.elementAt(i)).getSalaries();
                return sum;
                }
```

La clase *Employee* representa a la clase *Component* de la estructura del patrón, y se implementa una funcionalidad *default*. Esta clase la podemos utilizar para realizar el organigrama de la empresa representada en el diagrama anterior, de la siguiente manera:

```
boss = new Employee("CEO", 200000);

boss.add(marketVP =new Employee("Marketing VP", 100000));
boss.add(prodVP =new Employee("Production VP", 100000));

marketVP.add(salesMgr =new Employee("Sales Mgr", 50000));
marketVP.add(advMgr =new Employee("Advt Mgr", 50000));

for (int i=0; i<5; i++)
        salesMgr .add(new Employee("Sales "+new Integer(i).toString(), 30000.0F));

advMgr.add(new Employee("Secy", 20000));
prodVP.add(prodMgr =new Employee("Prod Mgr", 40000));
prodVP.add(shipMgr =new Employee("Ship Mgr", 35000));

for (int i = 0; i < 4; i++)
        prodMgr.add( new Employee("Manuf "+new Integer(i).toString(), 25000.0F));

for (int i = 0; i < 3; i++)
        shipMgr.add( new Employee("ShipClrk "+new Integer(i).toString(), 20000.0F));
```

Una limitación de esta forma de implementar el patrón *composite* es que cualquier nodo puede tener subordinados, y en un organigrama real nos interesa que algunos empleados permanezcan sin subordinados. Para evitar este problema sin la necesidad de agregar una clase separada para *Leaf* y otra para *Composite* podemos agregar una bandera (llamada *isLeaf*) que nos indique si es un nodo simple o un nodo compuesto dentro de *Employee*, y modificar el método *add()* de la siguiente manera:

```
public class Employee {
        ...
        boolean isLeaf;
        ...
        public void setLeaf(boolean b) {
                isLeaf = b;  //si es TRUE, no permite hijos, es un nodo simple
                }
        ...
        public boolean add(Employee e) {
                if (! isLeaf) subordinates.addElement(e);
                return isLeaf; //retorna FALSO si es un nodo simple
                }
        }
```

Usos conocidos

Se pueden encontrar ejemplos de este patrón en casi cualquier sistema orientado a objetos que manejen jerarquías de objetos compuestos y simples de manera uniforme.

Patrones relacionados

El patrón *Decorator* es utilizado frecuentemente con este patrón.

2.5.4 *Decorator*

Intención

Agregar responsabilidades a un objeto dinámicamente. Un decorador ofrece una alternativa flexible ante la herencia para extender la funcionalidad de un objeto.

Motivación

En algunas ocasiones es necesario agregar responsabilidades a un objeto específico, no a una clase, por ejemplo, un *toolkit* para una interfaz gráfica debe permitir agregar propiedades como bordes o barras de desplazamiento (*Scroll*) a cualquier componente de la interfaz del usuario. Una forma de lograrlo es mediante herencia, pero heredar un borde desde otra clase coloca un borde sobre cada instancia de esta subclase. De esta manera la elección de un borde se hace de manera estática, lo cual es poco flexible. Otro enfoque más flexible consiste en encapsular el componente dentro de otro objeto que agregue el borde. El objeto que encapsula el objeto básico es llamado *Decorator* (decorador). El decorador se ajusta a la

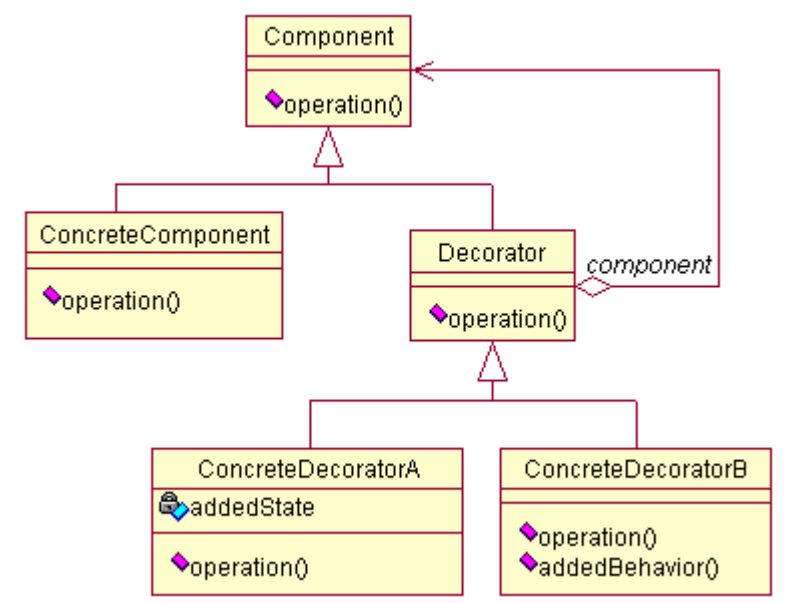
interfaz del componente que encapsula para que su presencia sea transparente a los clientes.

Por ejemplo, tenemos un objeto *TextView* que despliega texto en una ventana, y por *default* no posee barras de desplazamiento ya que no siempre son necesarias, pero cuando las necesitemos, las podemos agregar mediante un decorador: *ScrollDecorator*. Al suponer que también queremos un borde alrededor de *TextView*, podemos utilizar otro decorador: *BorderDecorator*.

Aplicabilidad

- Para agregar responsabilidades a objetos específicos de manera dinámica y transparente, es decir, sin afectar a otros objetos
- Para responsabilidades que pueden ser removidas
- Cuando la herencia no resulta funcional

Figura 28. Estructura patrón *Decorator*



Participantes

- Component. Define la interfaz para los objetos a los que se puede agregar responsabilidad o funcionalidad de forma dinámica.
- *ConcreteComponent*. Define los objetos a los cuales se puede agregar funcionalidad.
- *Decorator*. Mantiene una referencia hacia un objeto *Component* y define una interfaz que se ajusta a la de *Component*.
- *ConcreteDecorator*. Agrega funcionalidad a un objeto.

Consecuencias

- Presenta mayor flexibilidad que la herencia.
- Una aplicación no necesita pagar el costo de tener componentes que no se están utilizando. En lugar de tratar de soportar todas los componentes previsibles en una clase compleja y adaptable, se puede definir clases simples y agregarles funcionalidad al incrementarlos con decoradores.
- Generalmente un diseño que utiliza decoradores está compuesto de gran cantidad de objetos pequeños similares, y difieren entre sé solo en la forma en que se encuentran interconectados. Estos sistemas son fácilmente adaptables, pero pueden ser difíciles de entender y depurar.

Implementación

Se deben considerar los siguientes puntos al aplicar este patrón:

- La interfaz de un objeto decorador se debe adaptar a la interfaz del componente que decora.

- Cuando solamente se necesita agregar un decorador no es necesario definir una clase abstracta decoradora.
- Para asegurar una interfaz en común, los componentes y los decoradores deben descender de una clase en común. Esta clase en común se debe enfocar solo en definir la interfaz para manejar los componentes y los decoradores.

Muestra de código y utilización

En este ejemplo se mostrará de manera general la implementación de una interfaz de usuario utilizando decoradores. Definimos la interfaz para manejar los componentes y sus decoradores con la clase *VisualComponent* de la siguiente manera:

```
class VisualComponent {
        public void VisualComponent() {}

        public void draw() {}
        public void reSize() {}

        }
```

Ahora definimos la clase que define la interfaz para manejar los decoradores de la siguiente manera:

```
class Decorator extends VisualComponent {
        private VisualComponent component;
        public void Decorator(VisualComponent _vc) {}
        public void draw() {
                component.draw();
                }
        public void reSize() {
                component.reSize();
                }
        }
```

Un decorador decora una instancia de *VisualComponent* referenciada mediante el atributo privado llamado *component*, el cual debe ser inicializado en el constructor. Las subclases de *Decorator* definen

responsabilidades o decoraciones específicas, por ejemplo, la clase siguiente, *BorderDecorator*, que se encarga de agregar un borde a un componente.

```
class BorderDecorator extends Decorator {
        private int borderWith;
        public void BorderDecorator(VisualComponent _vc, int _borderWith) {}
        private void drawBorder() {}
        public void draw() {
                super.draw();
                drawBorder();
                }
        }
```

Una implementación similar requerirá otros decoradores específicos como una barra de desplazamiento (*ScrollDecorator*) por ejemplo. Ahora podemos relacionar instancias de estas clases para proveer diferentes decoraciones. Definiremos un objeto de *TextView* con un borde de la siguiente manera:

```
Window window =new Window();
TextView textView =new TextView();
window.setContents(new BorderDecorator(textView,1));
```

En este ejemplo asumimos que la clase Window provee el método *setContents(VisualComponent contents)* que permite agregar un componente visual dentro de la ventana.

Usos conocidos

Utilizado por interfaces de usuario orientadas a objetos para agregar adornos a componentes visuales.

Patrones relacionados

Un decorador se diferencia de un adaptador en que el decorador cambia las responsabilidades de un objeto, no su interfaz. Un adaptador proporciona a un objeto una interfaz completamente nueva.

2.5.5 *Facade*

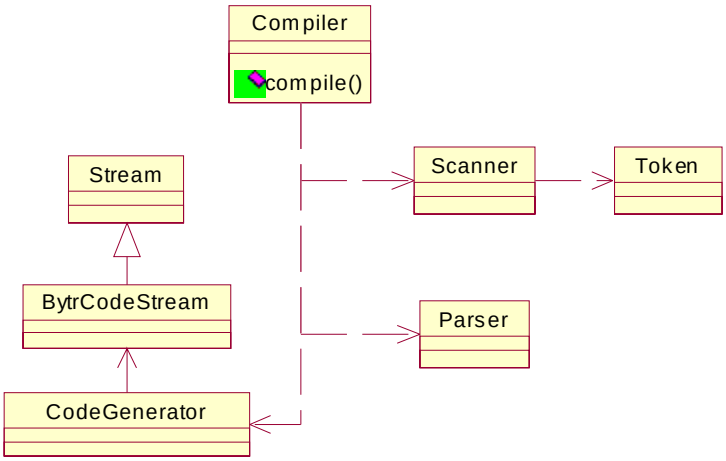
Intención

Su intención es proveer una interfaz en común para un conjunto de interfaces definidas en un subsistema, es decir, crear una interfaz a un nivel más alto para facilitar el uso de el subsistema.

Motivación

Descomponer un sistema en un conjunto de subsistemas ayuda a reducir la complejidad total. El patrón *Facade* pretende facilitar la comunicación y minimizar las dependencias entre subsistemas a través de una interfaz general. Podemos considerar un ambiente de programación que proporciona acceso a su compilador. El compilador es un subsistema que contiene clases como un *Scanner*, *Parser*, *ByteCodeStream*, etc. Algunas aplicaciones particulares necesitaran acceder directamente a estas clases, pero la mayoría de aplicaciones solo les interesa compilar código sin importarles detalles particulares del proceso de compilación. El subsistema de compilación también incluye una clase *Compiler*, que define una interfaz general que une todas las clases que implementan la funcionalidad de un compilador. Esta clase actúa como una fachada o *Facade*. En la siguiente figura se puede ver las relaciones de las clases del subsistema de compilación.

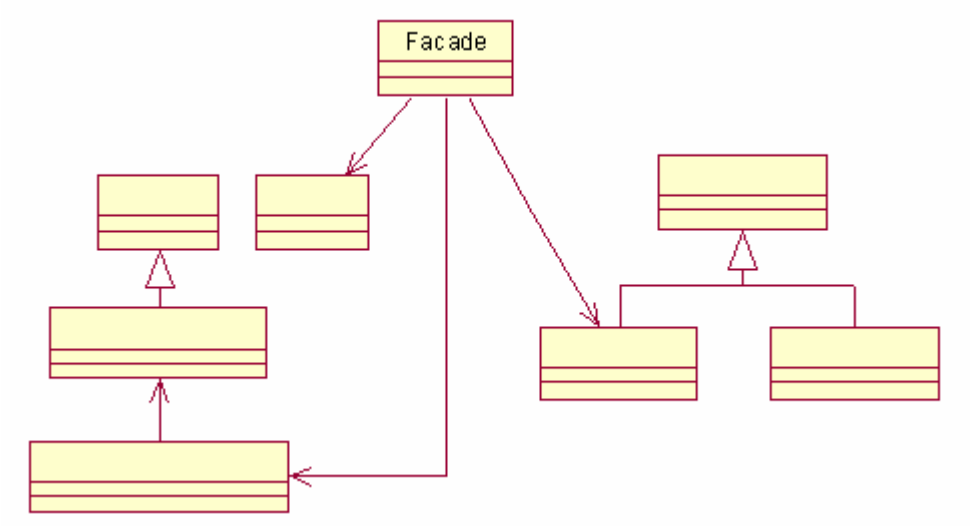
Figura 29. Subsistema compilador



Aplicabilidad

- Proveer una interfaz simple para manejar un subsistema más complejo. Una fachada provee una vista sencilla del subsistema, que para la mayoría de clientes es satisfactoria.
- Promover la independencia y portabilidad entre subsistemas.
- Cuando se desea dividir los subsistemas en niveles o capas, se puede definir un objeto *Facade* por cada nivel, para tener un solo punto de acceso.

Figura 30. Estructura patrón *Facade*



Participantes

- *Facade*: define la interfaz para acceder a las clases del subsistema, al conocer qué clases son las responsables de las peticiones hechas por los clientes.
- Clases del subsistema: implementan la funcionalidad del subsistema.

Colaboraciones

Los clientes se comunican con el subsistema a través de el objeto *Facade*, el cual traslada las peticiones hacia los objetos del subsistema.

Consecuencias

- Oculta la complejidad de un subsistema a los clientes y fácilita su utilización.

- Promueve una dependencia débil entre subsistemas y clientes, lo cual permite realizar cambios a los componentes del subsistema sin afectar a los mismos.

Implementación

La dependencia entre clientes y subsistemas se puede reducir aun más definiendo la fachada como una clase abstracta con subclases concretas para diferentes implementaciones del subsistema.

Un subsistema es similar a una clase en que ambos poseen interfaces, y ambos encapsulan algo: una clase encapsula atributos y métodos, y un subsistema encapsula clases.

Muestra de código y utilización

En este ejemplo sólo definiremos la clase *Compiler*, mencionada en la sección de motivación del presente patrón. Esta clase es la que actúa como fachada al subsistema de compilación y se encarga de unir todas las piezas que forman el compilador.

```
class Compiler {
        public void Compiler() {}
        public void compile(String _file) {
                Scanner scanner =new Scanner(_file);
                Parser parser =new Parser();
                parser.parse(scanner);
                //...

                }
}
```

El método *compile*() es el encargado de manejar instancias de las clases del subsistema para proporcionar la funcionalidad correcta solicitada por los clientes.

Patrones relacionados

El patrón *Abstract factory* puede utilizarse junto con *Facade* o como una alternativa a este. Por lo general, solamente un objeto *Facade* es necesario para un subsistema, por lo que podría implementarse con el patrón *Singleton*.

2.5.6 *Flyweight*

Intención

Compartir una gran cantidad de objetos pequeños finamente diseñados de una forma eficiente.

Motivación

La mayoría de aplicaciones se puede beneficiar de utilizar objetos en su diseño, pero una implementación sencilla o simplemente una mala implementación puede resultar demasiado costosa. Por ejemplo, la mayoría de implementaciones de editores de texto utilizan objetos para representar elementos incrustados, como tablas y figuras, pero no utilizan objetos para representar cada carácter en el documento, lo cual representaría una mayor flexibilidad, ya que la aplicación podría manejar distintos *sets* de caracteres de manera uniforme. Pero este diseño tiene como desventaja su alto costo ya que se necesitaría una gran cantidad de objetos para representar todos los caracteres de un documento.

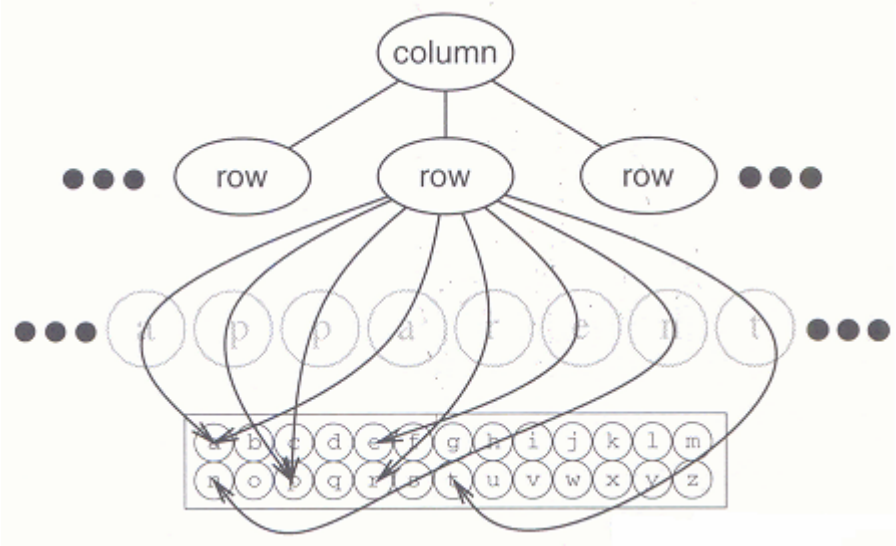
El patrón *Flyweight* describe cómo compartir objetos eficientemente para evitar el alto consumo de memoria y posibles problemas de desempeño ocasionados por la gran cantidad de objetos creados.

Un *Flyweight* es un objeto que puede ser utilizado en múltiples contextos de manera simultanea. Este objeto posee un estado intrínseco y un estado extrínseco. El estado intrínseco se almacena dentro del *Flyweight*, y consiste en información independiente del contexto, por lo que puede compartirse, mientras que el estado extrínseco no puede compartirse ya que depende y varía con el contexto del *Flyweight*. Los objetos cliente son los encargados de manejar el estado extrínseco de los *flyweights*.

El patrón *Flyweight* sirve para representar conceptos o entidades que son demasiado numerosas o abundantes para ser representadas con objetos. Por ejemplo, un editor de texto puede crear un *flyweight* para cada letra del alfabeto. Cada *flyweight* almacena un carácter (estado intrínseco), pero su posición y formato en el documento son manejadas de manera independiente (estado extrínseco).

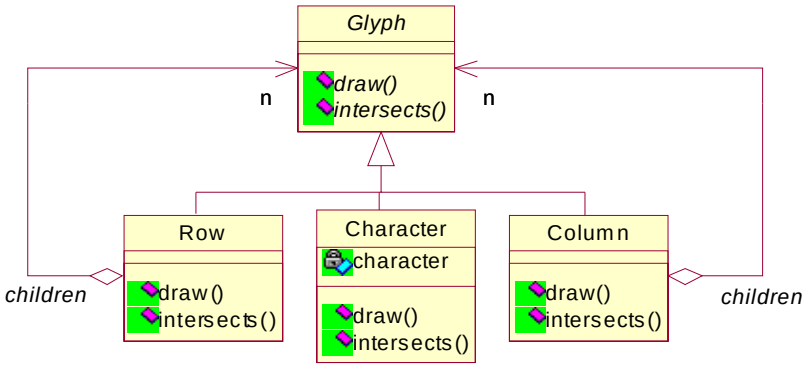
Lógicamente existe un objeto por cada ocurrencia de un carácter en el documento, pero físicamente existe un objeto *flyweight* por carácter, y este aparece en distintos contextos dentro de la estructura del documento. Cada ocurrencia de un carácter en particular hace referencia a una sola instancia dentro de un conjunto de objetos compartidos. La siguiente figura representa los objetos *flyweight* en memoria:

Figura 31. Diagrama de objetos *Flyweight*



La estructura de las clases para los objetos presentados en la figura anterior es la siguiente:

Figura 32. Ejemplo patrón *Flyweight*



Glyph representa la abstracción de los objetos manejados en el editor, donde varios de estos objetos son *flyweights*. Los métodos que dependen del estado extrínseco del *flyweight* lo reciben a través de parámetros, por ejemplo, los métodos *draw* e *intersects* estarían declarados de la siguiente manera:

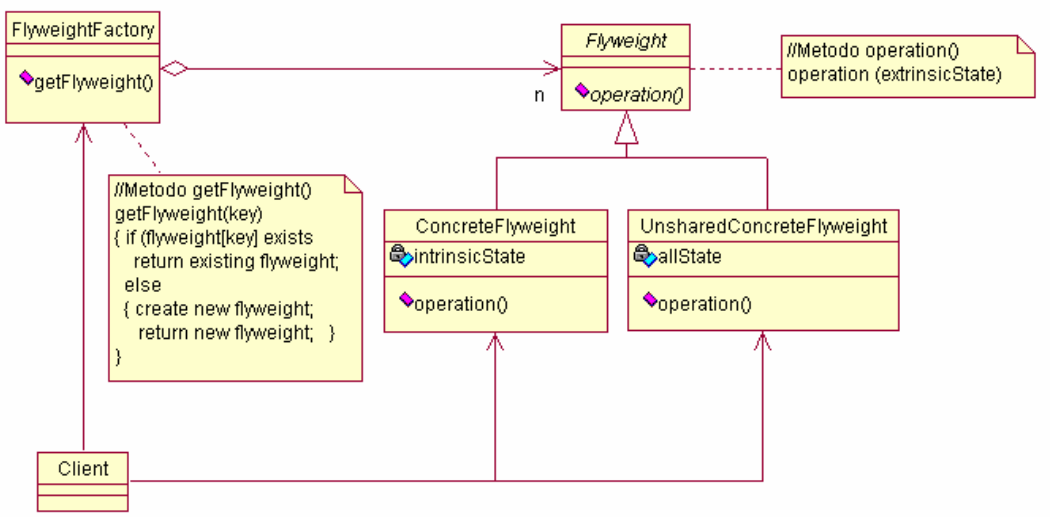
```
...
public void draw(Context _context) ...
public void intersects (Point _p, Context _context)...
...
```

Aplicabilidad

La efectividad de este patrón dependerá en gran medida de cómo y dónde es utilizado, por lo que se recomienda utilizar este patrón solo cuando se cumplan todas las condiciones siguientes:

- La aplicación utiliza un gran número de objetos.
- El costo de uso de memoria es alto debido a la gran cantidad de objetos.
- La mayoría de los objetos tienen o se puede definir un estado extrínseco.
- Muchos grupos de objetos se pueden reemplazar por relativamente pocos objetos compartidos una vez que su estado extrínseco se ha removido.
- La aplicación no depende de la unicidad de los objetos, ya que los *flyweights* son objetos compartidos, por lo que conceptualmente, se tratara de distintos objetos, pero físicamente serán los mismos objetos compartidos.

Figura 33. Estructura patrón *Flyweight*



Participantes

- *Flyweight*. Declara la interfaz sobre la cual los objetos *flyweight* reciben y manipulan su estado extrínseco.
- *ConcreteFlyweight*. Implementa la interfaz definida por *Flyweight* y agrega atributos para manejar el estado intrínseco.
- *UnsharedConcreteFlyweight*. No todos los objetos deben ser compartidos. La interfaz declarada por *Flyweight* habilita el poder compartir los objetos, pero no necesariamente debe ser así para todas sus subclases. Como se puede ver en el ejemplo definido en la sección de motivación, los objetos *Row* y *Column* no son compartidos aunque extienden la clase *Glyph*.
- *FlyweightFactory*. Crea y administra los objetos *flyweight*. Cuando un cliente solicita un *flyweight*, esta clase se encarga de proporcionarlo si existe, y si no, lo crea y devuelve la nueva instancia.
- *Client*. Mantiene una referencias hacia los *flyweights* y se encarga de calcular o almacenar el estado extrínseco de los *flyweights*.

Colaboraciones

El estado intrínseco es almacenado por las instancias de *ConcreteFlyweight* y el estado extrínseco es almacenado o calculado por los objetos cliente (instancias de *Client*). Los clientes pasan el estado extrínseco o contexto al *flyweight* al invocar sus métodos. Los clientes deben obtener los *flyweights* a través de *FlyweightFactory*, que es quien los administra.

Consecuencias

Los *flyweights* pueden introducir en la aplicación problemas de rendimiento asociados a la transferencia o cálculo del estado extrínseco de los objetos compartidos, pero ese costo de rendimiento se compensa con el ahorro de espacio en memoria, el cual aumenta mientras más objetos son compartidos.

Implementación

Un factor importante a considerar al implementar este patrón es el manejo adecuado de los objetos compartidos. Los clientes no deben instanciar los *flyweights* directamente. Estos deben acceder a ellos a través de la clase *FlyweightFactory*, que administra los objetos compartidos, permitiendo a los clientes identificar el *flyweight* que necesitan de un conjunto de flyweights creados.

Muestra de código y utilización

Como se mencionó anteriormente en el ejemplo del editor de documentos, definimos una clase Glyph para manejar los objetos compartidos (flyweights) y los no compartidos (filas o columnas).

Como se puede ver, las instancias de *Glyph* son objetos compuestos definidos en base al patrón *Composite*. La definición de la clase *Glyph* quedará de la siguiente manera:

```
class Glyph {
        public void Glyph() {}
        public void draw(Window _w, GlyphContext _gc) {}
        public void setFont(Font _f, GlyphContext _gc) {}
        public Font getFont(GlyphContext _gc) {}

        public void insert(Glyph _g, GlyphContext _gc) {}
        public void remove(GlyphContext _gc) {}
        }
```

Los métodos *draw*(), *setFont*() y *getFont*() son los encargados de manipular el estado extrínseco de los objetos *flyweight*, que en este ejemplo son instancias de la clase *MyCharacter*. Asumimos que ya existe la clase *GlyphContext* para pasar como parámetro el contexto en el que se encuentra cada *flyweight* (estado extrínseco). La definición de *MyCharacter* (*flyweight*) es la siguiente:

```
class MyCharacter extends Glyph{
        public void MyCharacter() {}
        public void draw(Window _w, GlyphContext _gc) {}
        public void setChar(char _c) {
                charcode=_c;
                }
        private char charcode;
        }
```

Y finalmente, la clase que administrará todos los flyweights para que puedan ser compartidos de la manera correcta:

```
class GlyphFactory {
        public GlyphFactory() {
                character =new ArrayList();
                char _c =' ';
                for (int i=0; i<MAXCHARCODES; i++)
                        character.add(new MyCharacter());
                }
        public MyCharacter createCharacter(char _c) {
                if (existsChar(_c) ==null)
                        character.add(new MyCharacter(_c));
                return existsChar(_c);
                }
        public Row createRow() {
                return new Row();
                }
```

```
public Column createColumn() {
            return new Column();
            }
private MyCharacter existsChar(char _c) {
            /* Implementa busqueda de _c y devuelve referencia,
               si no retorna null */
            }
private ArrayList character;
private final int MAXCHARCODES =128;
}
```

Los clientes accederán a las instancias compartidas a través de la clase *GlyphFactory*. Define una lista para manejar las instancias de los objetos compartidos (*character*) y una constante, MAXCHARCODES que define el número de objetos inicialmente a ser creados. Se debe notar que solo a través del método *createCharacter*() los clientes podrán acceder a los *flyweights*.

Usos conocidos

Este patrón no es muy utilizado a nivel de una aplicación, es más utilizado como una técnica de administración de recursos por los sistemas operativos a más bajo nivel, pero resulta beneficioso tener en cuenta su existencia para utilizarlo cuando sea necesario.

Patrones relacionados

Este patrón por lo regular, es utilizado junto con el patrón *Composite*. También se recomienda implementar el patrón *State* junto con el *Flyweight*.

2.6 Patrones de Comportamiento

Los patrones de comportamiento describen formas de comunicación y flujos de control entre las clases y objetos, que permitien un mayor enfoque en la forma en que los objetos se encuentran interconectados.

Los patrones de comportamiento de objetos utilizan la composición de objetos en lugar de herencia. Algunos patrones describen la forma en que un grupo de objetos realizan una tarea que un solo objeto no puede realizar de manera individual. Algo importante es que estos objetos que cooperan entre sí, conocen a sus compañeros, es decir, mantienen referencias entre ellos. En esta sección examinaremos los patrones *Chain of responsibility*, *Command*, *Memento*, *Observer* y *State*, los cuales son patrones de comportamiento de objeto.

2.6.1 *Chain of responsibility*

Intención

Evitar el acoplamiento entre los objetos emisor y receptor a través de un conjunto de objetos que puedan responder la petición del emisor, es decir, existe más de un objeto receptor capaz de manejar los mensajes del emisor.

Motivación

Consideremos una interfaz gráfica del usuario, en la cual se puede obtener ayuda sobre cualquier parte de la interfaz simplemente al hacer clic sobre el componente o área deseada. La ayuda presentada depende del componente de la interfaz y de su contexto.

Por ejemplo, un botón puede presentar distinta información de ayuda si se encuentra en un cuadro de diálogo que si se encuentra en la ventana principal de la aplicación.

Además, si no existe información de ayuda sobre un componente específico de la interfaz, se debe desplegar información de ayuda sobre el contexto inmediato, por ejemplo, la ventana que contiene al botón.

El problema al implementar este sistema de ayuda a la interfaz es que el objeto que hace la solicitud (el cliente) no conoce de manera explícita el objeto que debe proveer la ayuda. El patrón *Chain of responsibility* (cadena de responsabilidad) define la forma de separar o no crear dependencia entre el objeto que inicia la petición (ayuda) y los objetos que deben responder a la petición solicitada.

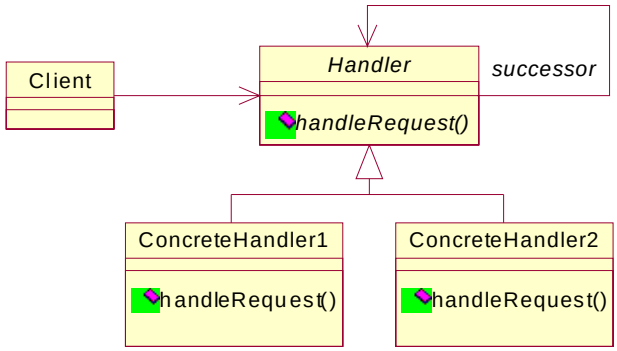
De esta manera, se crea una cadena de objetos que deben manejar la petición hecha por el emisor. El primer objeto en esta cadena recibe la solicitud y dependiendo del contexto, puede manejar la solicitud o enviarla al siguiente objeto en la cadena, el cual se comportará de la misma manera. El objeto que hace la solicitud no sabe de manera explícita qué objeto manejará su petición, por lo que se dice que la solicitud tiene un receptor implícito.

Para poder trasladar la petición hecha por el objeto emisor hacia los distintos objetos en la cadena, estos deben compartir una interfaz en común. En nuestro ejemplo, se puede definir una clase para manejar las peticiones de ayuda llamada *HelpHandler* con el método *handleHelp*(). De esta manera, las clases que necesiten manejar las peticiones de ayuda deberán extender la clase *HelpHandler*.

Aplicabilidad

- Cuando más de un objeto puede manejar una petición realizada y el receptor específico no es conocido de antemano.
- Cuando se desea trasladar una petición hacia uno de varios objetos sin especificar el receptor explícitamente.
- Cuando la cadena de objetos que debe manejar las peticiones debe ser especificada dinámicamente.

Figura 34. Estructura patrón *Chain of responsibility*



Participantes

- *Handler*. Define la interfaz para manejar las peticiones y opcionalmente una referencia hacia el objeto sucesor.
- *ConcreteHandler*. Maneja la petición específica del objeto. Si puede manejar o responder a la petición lo hace, y si no, lo traslada hacia su sucesor.
- *Client*. Realiza la petición hacia una instancia de las clases *ConcreteHandler*.

Colaboraciones

Cuando un objeto cliente realiza una petición, esta es enviada a lo largo de la cadena de objetos hasta que uno de ellos asume la responsabilidad de manejarla.

Consecuencias

Cuando un objeto realiza una petición, no necesita saber qué objeto la recibirá, solo sabe que esta será manejada de la manera correcta. Los objetos receptor y el emisor no se conocen explícitamente. Además la cadena de objetos interconectados es bastante simple, ya que cada objeto solo mantiene una referencia hacia su sucesor.

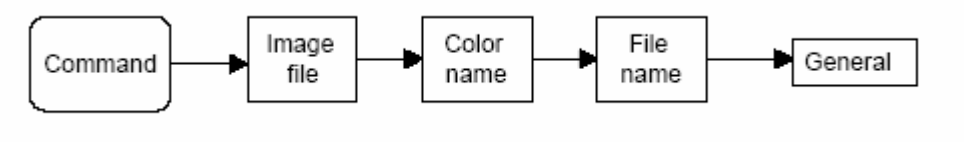
Una desventaja es que no se puede garantizar que una petición será manejada, ya que puede alcanzar el final de la cadena y ser desechada, o bien por una mala configuración en la cadena no encontrar un objeto que responda a la petición.

Implementación

La implementación de la referencia hacia el objeto sucesor se puede realizar de dos maneras: definiendo nuevas referencias (en la clase *Handler* o en las clases concretas *ConcreteHandler*) o utilizando referencias existentes. Si ya se tiene referencias entre los objetos no es necesario crear referencias redundantes para crear la cadena de objetos, además se ahorra más espacio en memoria que definiendo nuevas referencias entre los objetos.

Muestra de código y utilización

Consideremos un sistema para desplegar el resultado de una petición ingresada, que puede ser el nombre de un archivo gráfico (jpg), nombres de archivos en general, un color (rojo, verde y azul) o cualquier otra petición. Si se ingresa el nombre de un archivo .jpg, este será desplegado en un componente para desplegar imágenes; si se ingresa el nombre de un archivo, este será resaltado en la lista de archivos disponibles; si se ingresa un color, se deberá desplegar en un panel el color ingresado; y finalmente si se ingresa cualquier otra cosa no manejada por los anteriores objetos, simplemente será desplegado en una lista. La cadena se comportaría de la siguiente manera:



Primero definimos la clase abstracta *chain*, para manejar los objetos entrelazados:

```
public interface Chain {
        public abstract void addChain(Chain c);
        public abstract void sendToChain(String mesg);
        public Chain getChain();
        }
```

El método *addChain* permite añadir otro objeto a la cadena, *getChain* retorna el objeto actual, y el método *sendToChain* envía un mensaje al siguiente objeto en la cadena. Las clases que deben implementar esta interfaz son: *Imager* (clase encargada de desplegar la imagen), *ColorImage* (clase encargada de desplegar el color ingresado), *FileList* (clase encargada de resaltar el nombre del archivo ingresado si se encuentra dentro de la lista) y *RestList* (clase que despliega la instrucción ingresada si ninguna de

las clases anteriores puede manejarla). A continuación se mostrará la definición de estas clases, más no su implementación:

```
public class Imager extends JPanel implements Chain { ...
public class ColorImage extends JPanel implements Chain { ...
public class RestList extends JawtList implements Chain { ...
public class FileList extends RestList { ...
```

Todas las clases anteriores implementan los métodos definidos por *Chain*, y manejan el siguiente atributo para entrelazar los objetos de la cadena: `private Chain nextChain = null;`. Luego, para configurar la cadena, creamos instancias de las clases antes mencionadas y las entrelazamos de la siguiente manera:

```
sender.addChain(imager);
imager.addChain(colorImage);
colorImage.addChain(fileList);
fileList.addChain(restList);
```

De esta manera los objetos quedan enlazados y la petición inicial es enviada por la cadena desde una clase encargada de recibir la petición de el usuario.

Usos conocidos

En la mayoría de interfaces de usuario gráficas, cuando el usuario realiza alguna acción (clic o presiona una tecla), se genera un evento que es enviado hacia una cadena de objetos para que sea manejado.

Patrones relacionados

Este patrón generalmente es implementado junto al patrón *Composite*, donde la referencia hacia el padre definida (por el patrón *Composite*) puede actuar como el sucesor.

2.6.2 *Command*

Intención

Tiene como objetivo encapsular las peticiones realizadas en forma de objeto, un comando.

Motivación

A veces es necesario realizar peticiones hacia objetos sin saber realmente que operación se está solicitando o bien que objetos manejaran la petición. Por ejemplo, un *toolkit* para una interfaz de usuario incluye objetos como botones y menús para responder a las peticiones hechas por los usuarios, pero el *toolkit* no puede implementar las peticiones explícitamente en estos objetos, ya que solo las aplicaciones que lo utilicen conocen lo que debe realizarse sobre cada objeto.

Este patrón permite tratar las peticiones hechas como un objeto. La clave es definir una clase abstracta, *Command*, que declara una interfaz para ejecutar las operaciones, que en su forma más simple solo define un método abstracto para ejecutarlas.

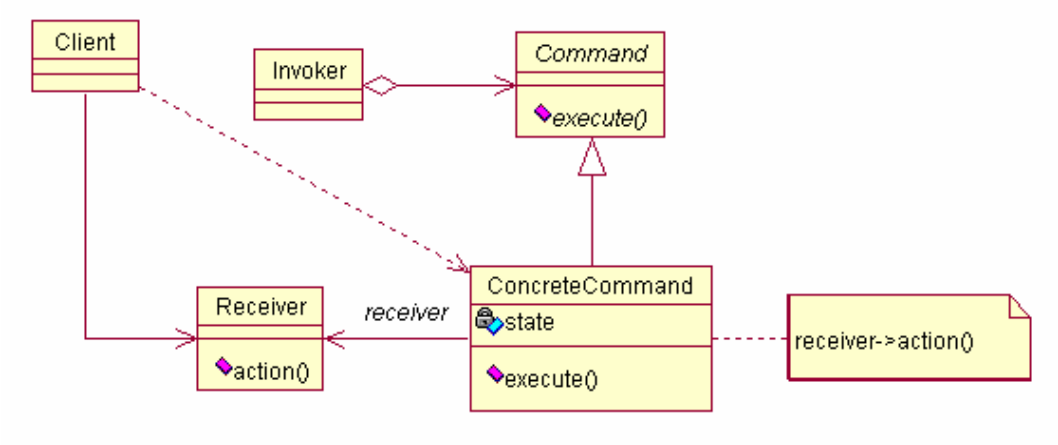
Aplicabilidad

- Cuando se necesita especificar, poner en cola o ejecutar peticiones en distintos tiempos. Ya que la petición es un objeto, esta puede tener un tiempo de vida independiente de la acción inicial que dió origen a la petición.
- Cuando se desea llevar un registro de los cambios realizados en el sistema en caso de falla. Esto se puede hacer implementando la

interfaz *Command* con para almacenar y cargar las operaciones realizadas. La recuperación del sistema en caso de falla sería ejecutando las operaciones almacenadas en el registro de cambios del sistema. Esto también permitiría la implementación de una operación deshacer o *Undo*.

- Cuando se desea estructurar un sistema con operaciones de alto nivel, basados en operaciones de más bajo nivel.

Figura 35. Estructura patrón *Command*



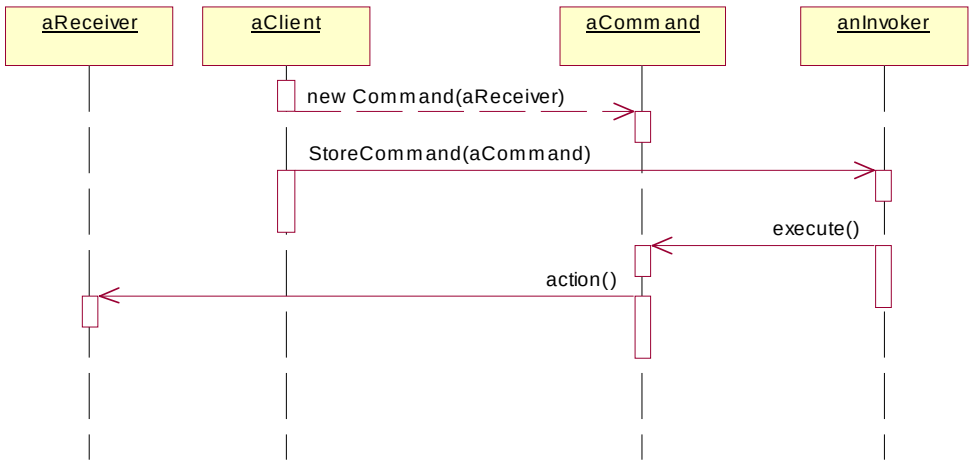
Participantes

- *Command*. Declara la interfaz para la ejecución de una operación.
- *ConcreteCommand*. Define una unión entre el objeto receptor y una acción específica e implementa el método de ejecución, *execute*() invocando la operación correspondiente en el objeto receptor.
- *Client*. Crea una instancia de *ConcreteCommand* y define su objeto *receptor*, *Receiver*.
- Invoker. Solicita a la clase *Command* que ejecute una petición.
- *Receiver*. Conoce como realizar una operación asociada a una petición.

Colaboraciones

El siguiente diagrama de interacción muestra como se relacionan las instancias de estas clases.

Figura 36. Diagrama de secuencia patrón *Command*



En esta figura, un objeto cliente crea una instancia de la clase *ConcreteCommand*, y específica su objeto receptor, el cual se asume ya esta creado, llamado *aReceiver*. El objeto *anInvoker*, almacena el objeto *aCommand* creado y ejecuta una petición ejecutando el método *execute*(). El objeto *aCommand* invoca el método en su receptor para que maneje la petición.

Consecuencias

- La clase *Command* se encarga de separar o independizar los objetos que invocan una petición de los que conocen como ejecutarla.
- Se pueden ensamblar peticiones dentro de un objeto compuesto con el patrón *Composite*.

- Facilita la adición de nuevas peticiones o comandos, ya que no es necesario cambiar las clases existentes.

Implementación

Al implementar un comando, se debe considerar que tanta responsabilidad debe tener. Por ejemplo, puede servir simplemente para crear la unión entre el objeto receptor y las acciones que deben ejecutarse para manipular la petición hecha, o implementar todo sin delegar responsabilidades sobre el receptor.

Los comandos también pueden soportar operaciones de deshacer y rehacer, manejando datos adicionales para determinar el estado del sistema. Los datos para determinar el estado que deben ser almacenados, pueden ser: El objeto receptor, los argumentos de la operación ejecutada en el receptor y los valores originales que cambiarán como resultado de la petición realizada. Además, se puede soportar la operación deshacer en más de un nivel, almacenando en una lista el historial de comandos ejecutados, donde el límite de la lista será el número de niveles de deshacer posibles.

Muestra de código y utilización

En este ejemplo se creará un programa con un menú de dos opciones y un botón. Primero definimos la interfaz que define el comando:

```
public interface Command {
        public void Execute();
        }
```

Luego, tenemos que definir las clases que implementarán esta interfaz, de la siguiente manera:

```
class btnRedCommand extends Button implements Command {
    public btnRedCommand(String caption)    {
```

```
        super(caption);
      }
      public void Execute()   {
        p.setBackground(Color.red);
      }
    }
class fileOpenCommand extends MenuItem implements Command {
      public fileOpenCommand(String caption)   {
        super(caption);
      }
      public void Execute()  {
        FileDialog fDlg=new FileDialog(fr,"Open file");
        fDlg.show();
      }
    }
class fileExitCommand extends MenuItem implements Command {
      public fileExitCommand(String caption)  {
        super(caption);
      }
      public void Execute() {
        System.exit(0);
      }
    }
```

Cuando un clic es hecho sobre cualquiera de estos elementos se genera un evento *actionPerformed*, el cual debe ser manejado de la siguiente manera:

```
public void actionPerformed(ActionEvent e) {
      Command obj = (Command)e.getSource();
      obj.Execute();
    }
```

Este evento es generado sobre una clase que implemente la interfaz *ActionListener*. En nuestro ejemplo, la definición de la clase es la siguiente:

```
public class testCommand extends Frame implements ActionListener {
```

Patrones relacionados

Se puede utilizar el patrón *Memento* para mantener la información del estado que un comando necesita para implementar la operación deshacer.

2.6.3 *Memento*

Intención

Su intención es capturar el estado interno de un objeto y almacenarlo, sin violar la encapsulación con el objetivo de que este objeto pueda ser restaurado a este estado en otro momento.

Motivación

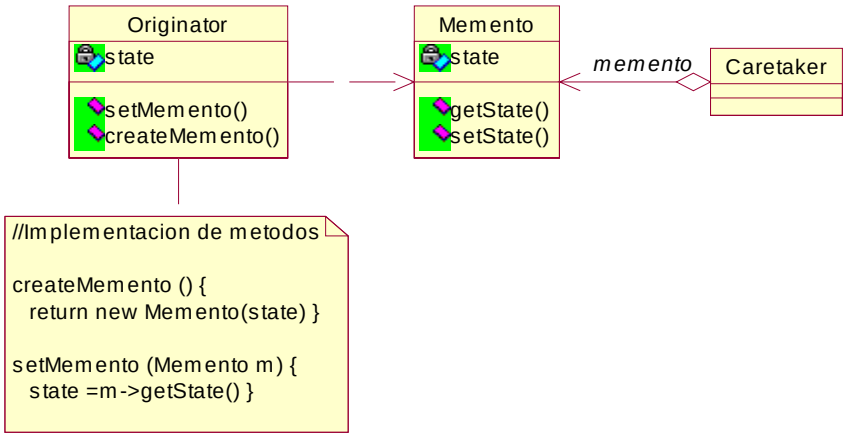
Cuando es necesario implementar operaciones de deshacer (*Undo*) o puntos de verificación (*checkpoints*) es necesario almacenar el estado interno de los objetos, para permitir a los clientes retornar a un estado pasado del objeto, pero los objetos normalmente encapsulan su información interna haciéndola inaccesible a otros objetos.

Un objeto *Memento* (recuerdo) almacena un *snapshot* o instantánea del estado interno de otro objeto, que es el que origina el objeto *memento*. En un mecanismo de deshacer, se solicitara una instantánea del objeto origen cuando sea necesario crear un punto de verificación de su estado. El objeto origen es quien inicializa la instantánea (objeto *memento*) con la información de su estado actual. Solo el objeto origen puede almacenar y recuperar la información de la instantánea generada para no violar la encapsulación.

Aplicabilidad

Cuando se necesita almacenar la instantánea de un objeto para poder retornar a un estado previo y no exponer los detalles internos del objeto, es decir, violar su encapsulación.

Figura 37. Estructura patrón *Memento*



Participantes

- *Memento*. Almacena el estado interno de un objeto. Debe almacenar la información necesaria para poder retornar el objeto a un estado previo. Solo el objeto origen, es decir, quien produce el *snapshot* puede acceder al objeto *memento* creado.
- *Originator*. Es el objeto origen que crea la instantánea de su estado actual y la utiliza para restaurarse a ese estado en un tiempo posterior.
- *Caretaker*. Se encarga de mantener la seguridad del objeto *memento* generado. Este objeto nunca manipula o examina el contenido del *memento*.

Colaboraciones

Un objeto *Caretaker* solicita un *snapshot* a un objeto origen, lo almacena por un tiempo y luego lo retorna a su origen para restaurar el objeto al estado previo. Algunas veces no se devuelve el *snapshot* a su origen porque este no necesita retornar a su estado previo.

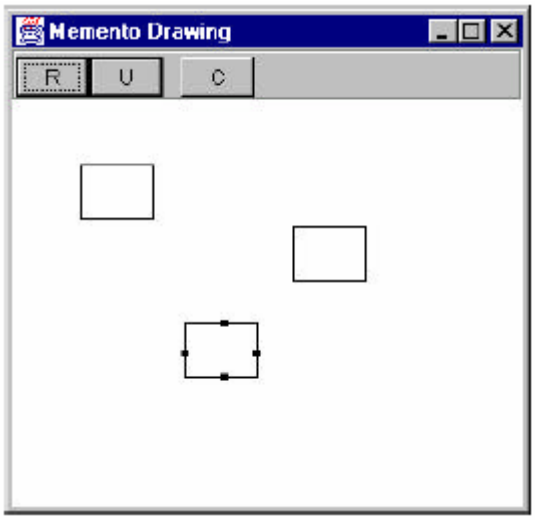
Consecuencias

Los objetos *memento* evitan exponer la información interna de un objeto almacenada fuera de el objeto origen. Una desventaja es que estos objetos pueden producir una sobrecarga ya sea por que se debe almacenar mucha información sobre el estado del objeto, o porque se solicitan instantáneas del objeto con mucha frecuencia. Este patrón es recomendable solo si no se produce una sobrecarga en el rendimiento del sistema.

Muestra de código y utilización

Tomaremos como ejemplo una aplicación sencilla para ejemplificar el uso de este patrón. Tenemos un programa que crea rectángulos y permite moverlos con el *mouse*. Este programa consta de tres botones: uno para dibujar un rectángulo (*RecButton*), otro para deshacer el último cambio (*UndoButton*) y un último botón para limpiar el área para dibujar los rectángulos. El programa se verá de la siguiente manera:

Figura 38. Programa para crear rectángulos



Los botones *RectButton*, *ClearButton* y *UndoButton* son implementados como un comando, de la misma forma en que se ejemplificó en el patrón *Command* en la sección 2.6.2.9. En este ejemplo solo grabamos y por lo tanto podemos deshacer, solo dos acciones: crear un rectángulo y cambiar su posición. Para esto, definimos la clase *Memento*, encargada de almacenar el estado de los objetos para luego restaurarlos.

```
class Memento {
  visRectangle rect;
  int x, y, w, h;
  public Memento(visRectangle r)  {
    rect = r;
    x = rect.x;  y = rect.y;
    w = rect.w;  h = rect.h;
  }
  public void restore()  {
    rect.x = x;  rect.y = y;
    rect.h = h;  rect.w = w;
  }
}
```

La clase *visRectangle* es la encargada de manejar las instancias de los rectángulos gráficados, por lo que necesitamos almacenar una referencia a la instancia de esta clase para almacenar el estado actual del objeto. Este es pasado al *Memento* en su constructor como parámetro. Luego posee el método *restore*(), que devuelve al objeto su estado anterior. Para almacenar el estado del objeto, creamos una instancia de *Memento*: `Memento m = new Memento(selectedRectangle);` . Y para restaurar el rectángulo a su estado previo, `m.restore()`.

Patrones relacionados

Los objetos *Command* (definidos en el patrón *Command*) pueden utilizar instantáneas (objetos *memento*) para implementar operaciones del tipo deshacer.

2.6.4 *Observer*

Intención

Su intención es definir una dependencia de uno a muchos entre los objetos, para que cuando un objeto cambie su estado todos los objetos dependientes sean notificados del cambio y se actualicen automáticamente.

Motivación

Cuando dividimos un sistema en una colección de clases es necesario mantener la consistencia entre los objetos, pero sin crear las clases con un alto grado de acoplamiento. Este patrón describe cómo crear este tipo de relaciones entre las clases.

Los objetos claves de este patrón son dos: un objeto a ser observado, llamado *subject*, y el objeto que lo observa, *observer*. Un objeto *subject* puede tener cualquier número de observadores que dependen de él.

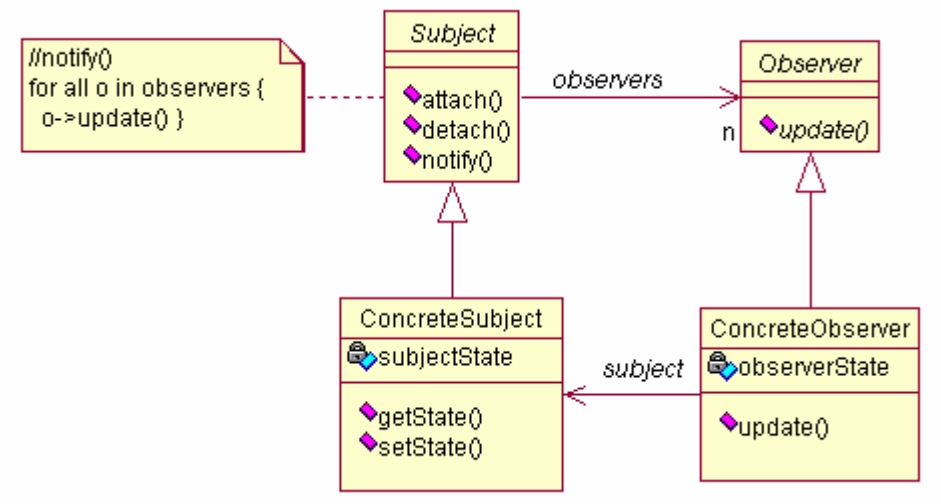
La idea es que todos los objetos observadores sean notificados de cualquier cambio que experimente el *subject*, para que estos sincronicen o actualicen su estado de acuerdo al *subject*.

Aplicabilidad

- Cuando una abstracción posee dos aspectos, uno que depende de otro. Al encapsular estos aspectos en objetos separados proporciona una mayor facilidad al modificarlos y promueve una mayor reutilización.

- Cuando un cambio en un objeto requiere cambios en uno o más objetos, sin saber cuántos objetos son los que deben actualizarse.
- Cuando un objeto debe ser capaz de enviar notificaciones a otros objetos sin conocerlos realmente.

Figura 39. Estructura patrón *Observer*



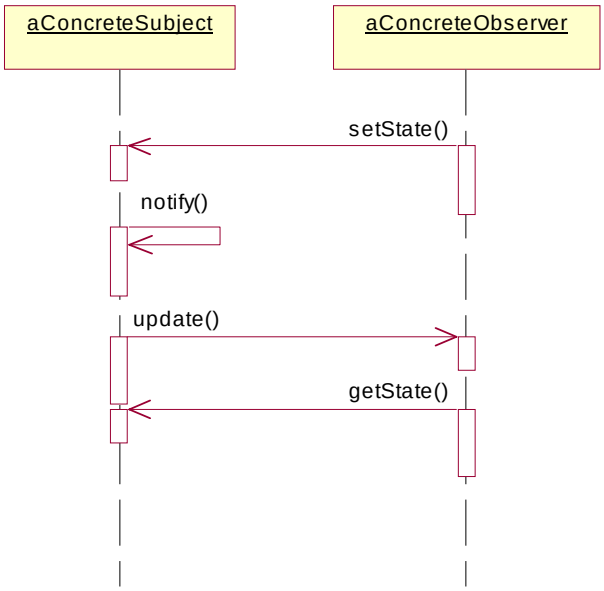
Participantes

- *Subject*. Provee una interfaz para agregar y eliminar objetos observadores. El número de observadores puede ser ilimitado.
- *Observer*. Define la interfaz para los objetos que deben ser notificados de los cambios en el *subject*.
- *ConcreteSubject*. Almacena el estado de interés para los objetos observadores, y envía notificación a estos de sus cambios de estado.
- *ConcreteObserver*. Mantiene una referencia hacia un *ConcreteSubject* y mantiene un estado que debe ser consistente con el estado del *subject*.

Colaboraciones

El siguiente diagrama muestra la colaboración entre un objeto *subject* y un *observer*. Si existe más de un observador, el *subject* debe notificar a cada observador(*update*), y cada observador obtener el estado del *subject* (*getState*).

Figura 40. Diagrama de colaboración



Consecuencias

Todo lo que un *subject* sabe es que posee una lista de observadores que comparten una interfaz definida por la clase abstracta *Observer*. El *subject* no conoce la clase concreta de cada observador, por lo que el acoplamiento que existe entre el *subject* y sus observadores es abstracto y mínimo.

A diferencia de una petición ordinaria que hace un objeto, la notificación que un objeto envía no necesita especificar a quien es enviada, por lo que se envia un tipo de *broadcast* a todos los objetos suscritos al *subject*. Al subject no le interesa cuántos observadores existen, su única responsabilidad es notificarles los cambios experimentados. El manejar la notificación queda a discreción del observador. Esto facilita la agregación y eliminación de observadores en cualquier momento.

Implementación

Para que un *subject* conozca a que objetos debe notificar de sus cambios, se debe almacenar referencias a estos objetos de manera explicita en el *subject*.

En algunas situaciones un observador puede depender de más de un *subject*. En estos casos es necesario extender la interfaz que define el método de actualización, *observer*, para que el observador conozca que *subject* es el que envía la notificación. El *subject* simplemente se puede pasar a si mismo como parámetro en el método *update*(), para que el observador sepa que objeto ha cambiado y actualizarse a sí mismo.

Otro aspecto importante a considerar en la implementación de este patrón es la eliminación de un *subject*. Al eliminar un *subject*, este no debe dejar referencias colgadas, es decir, referencias apuntando a objetos que ya no existen. Se recomienda que al eliminar un *subject*, este lo notifique a sus observadores para que puedan eliminar su referencia.

Muestra de código y utilización

En este ejemplo solo definiremos las clases *Observer* y Subject y su implementación *default*. Las clases que extiendan a *Subject* deberán definir atributos y métodos para manipular su información interna y simplemente utilizar la implementación default de *Subject*. Las clases son las siguientes:

```
class Subject {
        public void Subject() {}
        public void attach(Object _observer) {
                observers.add(_observer);
                }
        public void detach(Object _observer) {
                observers.remove(_observer);
                }
        public void Notify() {
                for (int i =0; i< observers.size(); i++)
                        ((Observer)(observers.get(i))).update(this);
                }
        private ArrayList observers;
        }
class Observer {
        public void Observer() {}
        public void update(Subject _changedSubject) {}
        }
```

Patrones relacionados

Se puede utilizar con el patrón *Singleton*, para definir un objeto *subject* único y globalmente accesible.

2.6.5 *State*

Intención

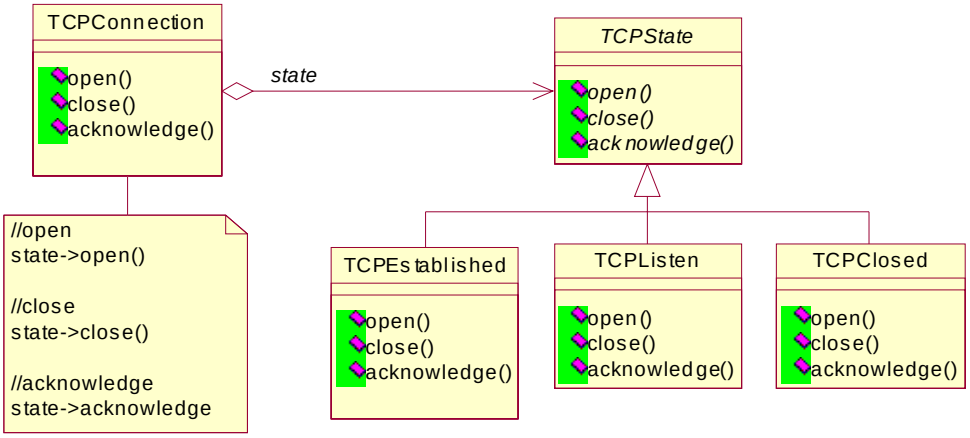
El objetivo de este patrón es permitir a un objeto cambiar su comportamiento dependiendo de su estado interno.

Motivación

Por ejemplo, tenemos una clase para representar una conexión de red TCP, *TCPConnection*. Una instancia de esta clase puede tener distintos estados para la conexión, como son: establecida, escuchando y cerrada. Cuando una instancia de *TCPConnection* recibe una petición de otros objetos, esta responderá a ellos dependiendo de su estado actual.

Este patrón describe cómo una clase puede presentar distinto comportamiento según su estado actual. Esto se logra introduciendo una clase abstracta para representar los estados posibles. Por ejemplo, declaramos una clase *TCPState* que define una interfaz en común para todas las subclases que representen los distintos estados de la clase *TCPConnection*. Las clases que implementen a *TCPState* implementarán el comportamiento específico para cada estado de la clase *TCPConnection*. En la figura siguiente se presenta el diagrama de clases de estas clases.

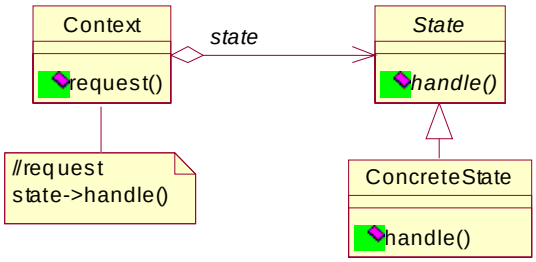
Figura 41. Diagrama de clases



Aplicabilidad

Básicamente este patrón es aplicable cuando el comportamiento de un objeto depende de su estado interno y este debe cambiar de forma dinámica.

Figura 42. Estructura patrón *State*



Participantes

- *Context*. Define una interfaz de interés para los clientes y mantiene una referencia hacia una instancia de *ConcreteState* que define su comportamiento actual.
- *State*. Define una interfaz para encapsular el comportamiento asociado a un comportamiento particular de la clase *Context*.
- *ConcreteState*. Cada una de estas subclases implementan el comportamiento asociado a un estado particular de *Context*.

Consecuencias

Este patrón encapsula un comportamiento específico asociado a un estado en particular en un objeto. Esto facilita la agregación y eliminación de nuevos estados y transiciones, simplemente definiendo nuevas subclases de *State*.

También define de forma explícita el estado actual del objeto, sin necesidad de realizar chequeos de variables para identificar el estado actual, simplemente se crea una referencia al objeto que representa el estado actual.

Implementación

El patrón *State* no define qué participantes definen el criterio para realizar las transiciones de estado aunque por lo regular es definido por la clase *Context*.

Otro aspecto a considerar al implementar este patrón es la creación de los objetos de estado, si solo se deben crear los estados cuando se necesiten y luego destruirlos, o crear todos los estados posibles y nunca destruirlos. La primera opción es recomendable cuando los estados posibles son desconocidos y el estado de *Context* no cambia con mucha frecuencia. La segunda opción es recomendable cuando los cambios de estado en *Context* ocurren constantemente y de forma rápida.

Muestra de código y utilización

En este ejemplo se definirán las clases *TCPConnection* y *TCPState* definidas en la sección de motivación del presente patrón. Esta es una versión muy simplificada del protocolo TCP, y solo nos enfocaremos en la definición de las clases concernientes al patrón *State*. La definición de *TCPConnection* es la siguiente:

```
class TCPConnection {
        public void TCPConnection() {}
        public void activeOpen()      {       state.activeOpen(this);       }
        public void passiveOpen()     {       state.passiveOpen(this);      }
        public void close()           {       state.close(this);            }
        public void send() {}
        public void acknowledge()     {       state.acknowledge(this);      }
```

```
        public void synchronize()       {       state.synchronize(this);        }
        public void changeState(TCPState _state) {      state=_state;       }
        private TCPState state;
        }
```

Todos los métodos definidos por TCPConnection dependerán del estado actual, manejado por el atributo *state*, que es una referencia a un objeto tipo *TCPState*. Debe notarse que en cada método, lo único que hace la clase *TCPConnection* es trasladar la petición realizada hacia el objeto que representa su estado actual.

```
class TCPState {
        public void TCPState() {}
        public void transmit(TCPConnection _tcpc) {}
        public void activeOpen(TCPConnection _tcpc) {}
        public void passiveOpen(TCPConnection _tcpc) {}
        public void close(TCPConnection _tcpc) {}
        public void synchronize(TCPConnection _tcpc) {}
        public void acknowledge(TCPConnection _tcpc) {}
        public void send(TCPConnection _tcpc) {}

        private void changeState(TCPConnection _tcpc, TCPState _state) {
                _tcpc.changeState(_state);
                }
        }
```

La clase *TCPState* define la interfaz a compartir por las clases concretas que implementarán el comportamiento en cada estado de la clase *TCPConnection*. Por ejemplo, podríamos definir las clases *TCPEstablished*, *TCPListen* y *TCPClosed* y estas implementarán el comportamiento específico para el estado que representá, extendiendo a *TCPState*.

```
class TCPEstablished extends TCPState { ...
class TCPListen extends TCPState { ...
class TCPClosed extends TCPState { ...
```

Patrones relacionados

Los objetos *State* pueden ser compartidos si no poseen variables de instancia, con el patrón *Flyweight*. Otra forma de implementar estos objetos es con el patrón *Singleton*.

3. APLICACIÓN PARA EJEMPLIFICAR EL USO DE LOS PATRONES DE DISEÑO

Hasta este punto, se ha definido la parte teórica sobre los patrones de diseño de *software* y cómo nos puede ayudar a resolver problemas de una manera elegante, promoviendo la reutilización de componentes y la eficiencia en el diseño de un sistema. En el presente capítulo definiremos los requerimientos de una aplicación sencilla, en la cual se utilizarán algunos de los patrones descritos en el capítulo anterior con el objetivo de mostrar la forma en que se utilizan al diseñar una aplicación transaccional utilizando una base de datos orientada a objetos.

Definiremos un *framework*[24] para el manejo de un catálogo de productos en línea. Para el desarrollo de este *framework* se utilizarán algunos conceptos definidos por el RUP[25]. Primero es necesario hacer el modelado del negocio[26] para comprender los procesos de la organización, pero para este ejemplo no es necesario realizar el modelado del negocio debido a que es la definición de un *framework* para un dominio específico de aplicación, no para una organización en particular. Este *framework* podrá adaptarse y extenderse para un negocio en particular, reutilizando el diseño que se desea definir.

3.1 Modelado de requerimientos

En un análisis orientado a objetos, los casos de uso son una parte integral que ayuda a definir los requerimientos funcionales de una aplicación particular. En el diseño de un *framework*, se tiene conocimiento del dominio de la aplicación a la que se enfoca, es decir, la esencia de este dominio, lo cual puede no ser expresado en términos de casos de uso.

A continuación se describen las componentes principales en el manejo de un catálogo de productos y su funcionamiento básico.

Clientes

Los datos básicos que deben manejarse de un cliente son: un número de identificación único, nombre, dirección, teléfonos, correo electrónico. Las organizaciones manejan distintos tipos de clientes, como clientes individuales, corporativos, etc. Por ejemplo, para un cliente corporativo nos interesa almacenar información sobre su límite de crédito y el nombre del contacto con la organización y para un cliente individual nos interesa llevar registro de su identificación personal, tipo de identificación, y número de tarjeta de crédito.

Productos

Se debe llevar registro de los siguientes datos: número de identificación único, nombre, descripción, marca, precio, peso, fecha de expiración, precio y existencia. La existencia de cada producto puede estar distribuida a través de múltiples almacenes. Los productos también deben estar clasificados de acuerdo a una categoría. Debe tomarse en cuenta el caso en que un producto está compuesto de uno o más productos y una categoría puede contener a una o más categorías.

Orden

Las ventas de los productos se manejan mediante una orden, la cual debe llevar registro de los siguientes datos: número de orden, fecha de la transacción, forma de pago, cliente que realiza la compra, el estado de la orden y los productos asociados a dicha orden. Una organización puede aceptar distintas formas de pago, como por ejemplo, efectivo, cheque,

tarjeta, etc. El estado de la orden puede ser: en proceso, cerrada o anulada. En una orden en proceso se pueden realizar las siguientes operaciones: agregar productos, eliminar productos y cancelar o finalizar la orden. Una orden cerrada solo puede ser anulada, y sobre una orden anulada no se puede realizar operaciones. Dependiendo de la organización, una orden puede tener otros estados, y su comportamiento dependerá de estos estados.

Como se puede observar en la definición anterior de los tres elementos principales de un catálogo de productos, los requerimientos están descritos de una forma muy general, ya que el objetivo de un *framework* es definir un diseño lo suficientemente general para poder adaptarse a las necesidades de una organización en particular, y lo suficientemente específico para cubrir los aspectos principales de un dominio de aplicación. A continuación se definen las clases identificadas en base a los requerimientos anteriores.

3.2 Etapa de análisis

En esta sección se realizará una definición más detallada de los requerimientos, enfocándonos en los datos mediante la abstracción de datos[27]. Además se definirán las acciones o métodos para manipular los datos. Con este enfoque podemos extraer la esencia del problema y no complicarlo con detalles de representación y manipulación de datos. Las operaciones de manipulación de los atributos setXXX y getXXX son implícitas por cada atributo definido en la clase.

Identificación de clases

Cliente: representa los clientes del sistema, que realizan compras de nuestro catálogo de productos. Los datos a manejar por esta clase son: número de identificación, nombre, dirección(es), teléfono(s) y correo electrónico. Esta clase puede extenderse mediante herencia para definir clases más específicas para cada tipo de cliente, por ejemplo, puede definirse la subclase ClienteCorporativo, del cual nos podría interesar almacenar además de la información definida en la clase padre, el nombre del contacto con dicho cliente y su límite de crédito.

Telefono: representa un número telefónico asociado a cada cliente, el cual puede tener más de uno. Los datos a manejar por esta clase son: código de área y número.

Direccion: representa la dirección asociada a un cliente. Los datos son: calle, avenida, número, aldea, municipio, departamento, ciudad, país y código postal.

Producto: representa los productos disponibles en el catálogo para que los clientes los adquieran. Un producto debe manejar la siguiente información: número de identificación, nombre, descripción, marca, dimensiones, fecha de caducidad, precio, categoría y existencia, la cual puede estar distribuida en distintos almacenes o bodegas.

Almacen: representa las distintas localidades de almacenamiento de los productos disponibles. Se identifican por un número de identificación y la dirección del lugar.

Inventario: esta clase se utiliza para llevar control de la existencia de cada producto en los distintos almacenes. Se lleva control de los productos y su existencia en cada almacén.

Categoría: representa una clasificación para los productos, para agruparlos de acuerdo a un criterio en particular. Los datos a almacenar son el número de identificación, nombre de la categoría y su descripción.

Orden: una orden representa la transacción realizada por un cliente al adquirir los productos del catálogo. Esta se identifica por un número único, la fecha de la transacción, el cliente que realiza la transacción, la forma de pago y el listado de productos que a adquirido el cliente. Cada organización puede manejar distintas formas de pago, por ejemplo, puede manejar solo efectivo, o *money orders*, cheques, etc. Además, una orden puede ser cancelada en más de una forma de pago, es decir, se puede cancelar una parte en efectivo, y la otra parte con cheque. Otro aspecto a manejar por esta clase es el estado de la orden, el cual puede ser: en proceso, cerrada o anulada. Estas son los estados básicos de una orden, pero pueden existir otros, por lo que el estado de la orden será manejado por la clase EstadoOrden.

EstadoOrden: representa el estado de una orden. Esta es una clase abstracta que define el comportamiento de la orden de acuerdo a su estado. Cada estado será representado por una subclase de EstadoOrden, la cual implementa el comportamiento específico de acuerdo al estado de la orden. Las operaciones que una orden puede realizar son las siguientes:

- Obtener total
- Agregar item
- Eliminar item

- Cancelar orden
- Anular orden
- Cerrar orden

ItemOrden: representa los items asociados a una orden y lleva registro del producto, las unidades despachadas de este producto y su precio.

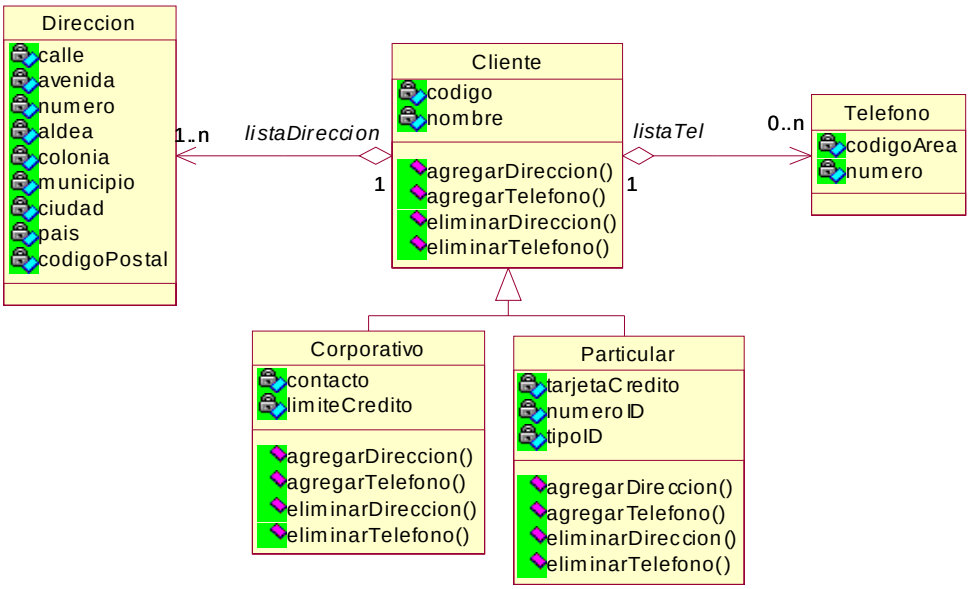
FormaDePago: representa la abstracción para las distintas formas de pago que una empresa podría aceptar y permitir extender esta clase para implementar formas específicas de pago, como efectivo, cheque, entre otras.

Detalle de las clases

A continuación se presenta la definición de las clases con sus atributos y métodos. Como se mencionó antes, las operaciones de manipulación de los atributos *set* y *get* son implícitas por cada atributo definido en la clase.

Cliente

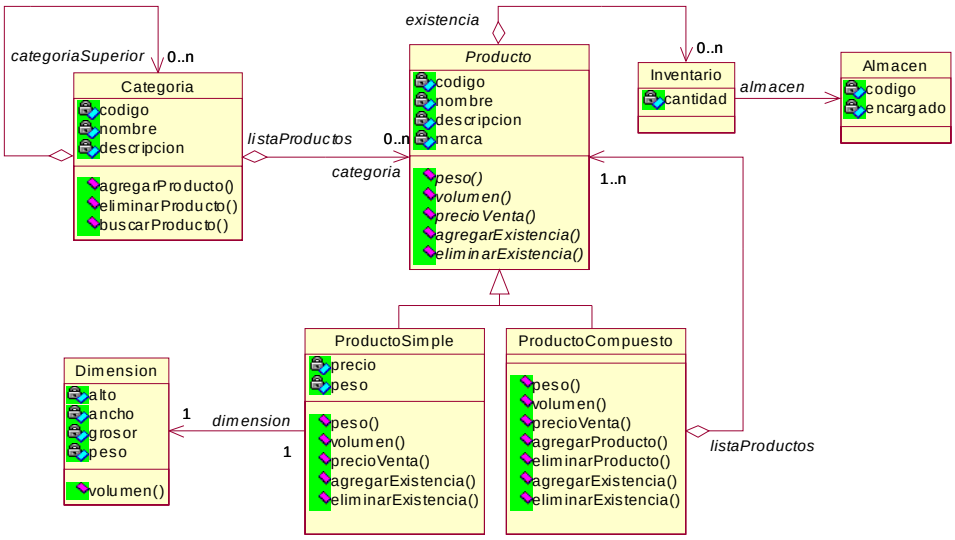
Figura 43. Clase cliente y clases relacionadas



Los atributos teléfono y dirección se modelan como clases separadas ya que un cliente puede tener más de un número de teléfono y más de una dirección.

Producto

Figura 44. Clase producto y clases relacionadas

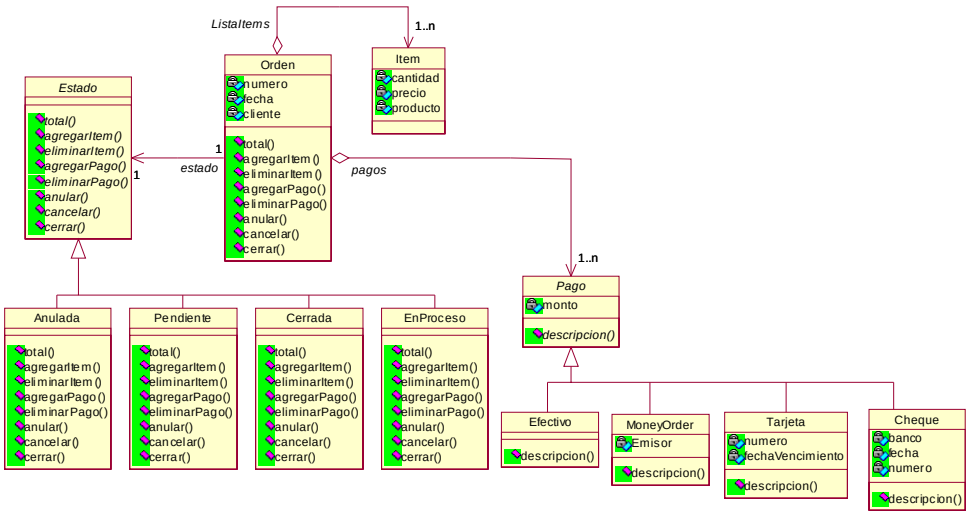


La clase producto define la interfaz para manejar instancias de ProductoSimple y ProductoCompuesto, de esta forma se facilita el manejo de los productos, ya que se puede manipular de igual forma los productos simples y compuestos sin necesidad de comprometer la funcionalidad y comportamiento específico de cada uno. Por ejemplo, el peso de un una instancia de ProductoCompuesto se obtiene sumando el peso de todos sus componentes, mientras que el de una instancia simple es el valor del atributo que define el peso.

Como se puede observar, este es un claro ejemplo de la utilización del patrón ***Composite***, descrito en la sección 2.5.3, en donde las clases Producto, ProductoSiimple y ProductoCompuesto representan a las clases *Component, Leaf* y *Composite* respectivamente, en la figura 26, que presenta la estructura de este patrón.

Orden

Figura 45. Clase orden y clases relacionadas

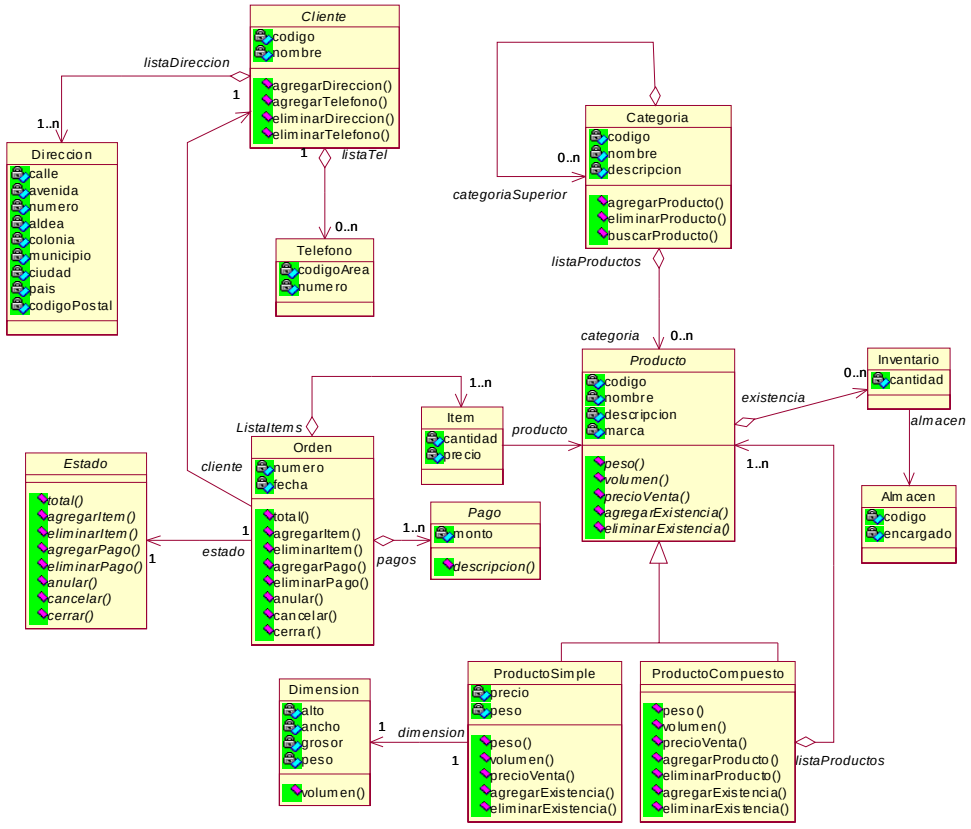


Dentro de la clase orden, el atributo cliente es una referencia hacia la clase cliente y el atributo producto de la clase item es una referencia hacia la clase producto.

El estado de la orden es una referencia hacia la clase estado. Esta clase define la interfaz para los distintos estados que puede manejar una orden. El comportamiento de la orden depende de su estado, por lo que existe una clase para representar cada estado e implementar su comportamiento. Esta es la implementación del patrón *state* descrito en la sección 2.6.5.

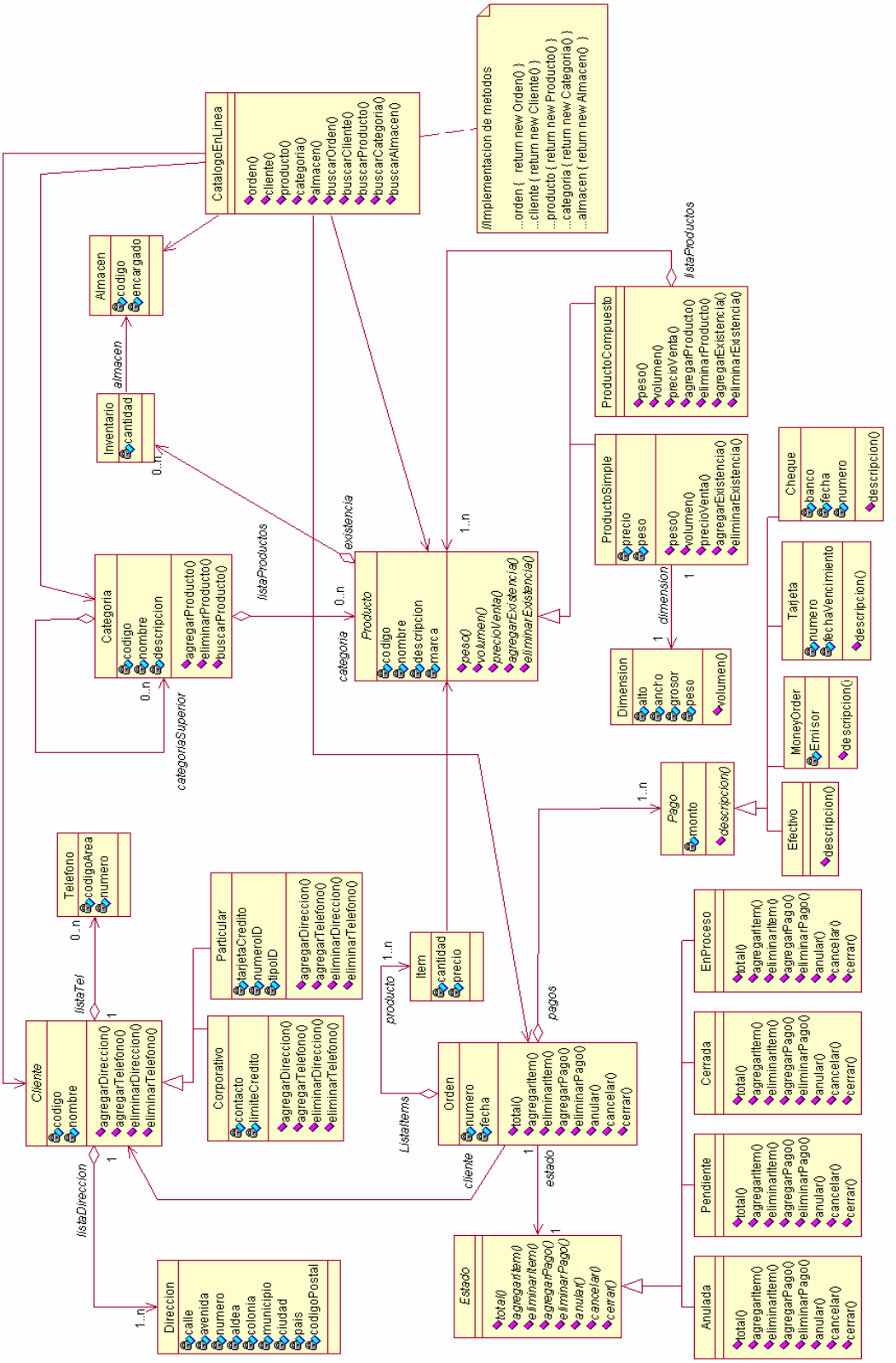
La clase pago representa una abstracción para los distintos tipos de pago. A continuación se presenta el Diagrama de clases completo, que forma el *framework* para un catálogo de productos.

Figura 46. *Framework* catálogo



En este diagrama no se presentan las subclases de pago, estado y cliente ya que lo que se pretende al definir un *framework* es describir un diseño y arquitectura flexible, la cual pueda implementarse para una aplicación en particular. A continuación se presenta un diagrama de clases basado en este *framework*, donde se extiende a las clases pago, estado y cliente.

Figura 47. Catálogo de productos



La clase CatálogoEnLínea define una interfaz para acceder las clases del sistema lo cual provee una mayor facilidad para la manipulación de las instancias de estas clases.

3.3 Etapa de diseño

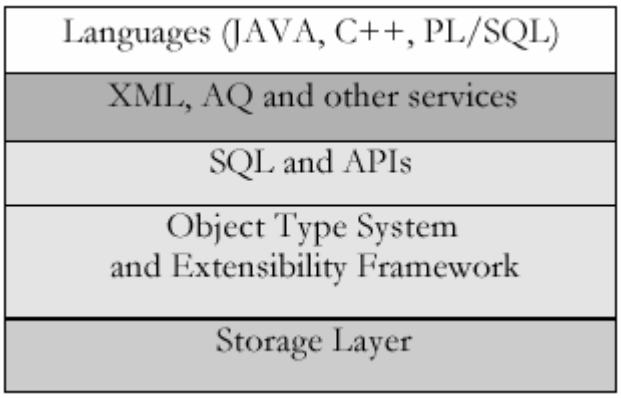
En la presente sección definiremos las especificaciones técnicas al implementar el *framework* definido en la sección anterior en una base de datos orientada a objetos, enfocándonos en las ventajas de utilizar este tipo de base de datos, al implementar un análisis orientado a objetos.

Para el presente ejemplo, se utilizará la base de datos *Oracle* 9i, que proporciona la tecnología objeto-relacional[28], la cual provee una completa capacidad para el modelado de objetos. Se seleccionó esta base de datos debido a que en el mercado actual no existe una base de datos completamente orientada a objetos. La base de datos que Oracle proporciona incluye características para manejar y modelar los datos como objetos, lo cual es una gran ventaja al implementar un análisis orientado a objetos, además de ser uno de los productos líderes en el mercado de las bases de datos. Otra característica importante es la completa compatibilidad y soporte que brinda *Oracle* para Java[29]. Primero se describirá la tecnología objeto-relacional, y luego la forma de interactuar con los objetos definidos en la base de datos a través de clases de Java.

3.3.1 Base de datos objeto-relacional

La tecnología objeto-relacional es una característica que proporciona la base de datos *Oracle* 9i para implementar un nivel de abstracción mayor al que se define en una base de datos relacional[30]. Proporciona una completa capacidad para el modelado de objetos, herencia y capacidad de evolución de tipos[31]. Además soporta Java dentro de la base de datos para la definición de métodos para las clases definidas, los cuales son manejados como procedimientos almacenados. Dentro del marco de trabajo objeto-relacional también se puede almacenar, manipular y consultar datos a través de XML de una forma eficiente. A continuación se presenta la arquitectura objeto-relacional manejada por *Oracle*.

Figura 48. Arquitectura objeto-relacional



Sobre la capa de *Storage* (almacenamiento de datos) se encuentra el *object type system* que es la implementación de SQL99 para definir y manipular objetos. A continuación se presenta una breve descripción de los componentes del sistema de tipos objeto-relacional.

Tipos objeto

Permiten la definición de tipos (clases) para representar los objetos de un negocio y sus relaciones (herencia, agregación) y almacenar sus instancias dentro de la base de datos en una columna de una tabla o como una tabla; además permite consultar, actualizar e insertar estas instancias. Un objeto puede estar contenido dentro de otro objeto a través de referencias, y ser accedido y manipulado como una colección o conjunto (*sets*) utilizando estructuras llamadas VARRAYS y tablas anidadas (*nested tables*). Se puede definir operaciones sobre los objetos, definidos como métodos, los cuales pueden implementarse como procedimientos almacenados dentro de la base de datos en Java o PL/SQL. Los objetos también poseen un identificador único, llamado OID (*object* ID), el cual se utiliza para crear referencias entre los objetos. Oracle permite manejar los datos definidos como objetos de la misma forma en que se acceden los datos definidos de forma relacional mediante SQL DML's[32].

Los componentes objeto-relacional definidos por el *object type system* de *Oracle* 9i poseen una correspondencia estrecha con los componentes de una base de datos relacional, por ejemplo, las referencias entre objetos (REF) son similares a los constraints de llave foránea, los métodos de los tipos definidos son procedimientos almacenados (los cuales pueden ser escritos en Java, PL/SQL o C/C++), y la seguridad de los datos y los modelos de transacciones para objetos son exactamente los mismos que están definidos para una base de datos relacional.

Vistas objeto

La tecnología objeto-relacional implementada por *Oracle* 9i también permite manipular datos almacenados en tablas (relacional) como objetos a través de vistas objeto, las cuales permiten sintetizar objetos del negocio a partir de datos que continúan almacenados en tablas relacionales. Las vistas objeto permiten definir objetos para ser utilizados en una aplicación sin necesidad de migrar los datos almacenados en forma relacional hacia una forma objeto-relacional.

Herencia

Es un concepto fundamental en cualquier sistema orientado a objetos. *Oracle* provee herencia simple al igual que Java (una clase puede tener un solo padre). Permite la definición de nuevos tipos (clases) en base a una clase padre, es decir, hereda los atributos y métodos definidos la clase padre y puede agregar nuevos atributos y sobrescribir los métodos heredados, lo cual proporciona la capacidad de utilizar referencias hacia instancias de un subtipo en un contexto de referencia declarado en términos de un supertipo, es decir, polimorfismo.

Colecciones

Son tipos que contienen múltiples elementos. *Oracle* provee dos tipos de colecciones: *varrays* y n*ested tables* (tablas anidadas). Un *Varray* contiene un número variable de elementos ordenados. Las tablas anidadas se crean como tipos de dato manejados en forma de tablas. En la sección de mapeo del modelo de objetos en la base de datos se explicará la forma de utilizar estos tipos de colecciones. Estas colecciones se pueden utilizar como una columna de una tabla o como un atributo de un tipo objeto.

Tipos referencia

Una referencia es un apuntador hacia una fila objeto. Las referencias son muy importantes para modelar las relaciones de agregación y asociación, y para permitir la navegación entre los objetos relacionados.

3.3.2 Java y la tecnología objeto-relacional

El sistema objeto-relacional definido por *Oracle* proporciona una completa compatibilidad con varios lenguajes: PL/SQL, C/C++ y Java. Las instancias de los tipos definidos en la base de datos pueden ser accedidas y manipuladas a través de un API como JDBC. Además, *Oracle* proporciona la herramienta *Jpublisher*[33], la cual mapea o genera clases de Java a partir de los tipos objeto definidos en la base de datos[34].

La tecnología objeto-relacional provee una forma más natural y productiva para mantener una consistencia entre un conjunto de clases en Java a nivel de la aplicación y el modelo de datos a nivel de *storage*. Los objetos definidos mediante SQL[35] se pueden mapear a clases de Java, utilizando la herramienta *JPublisher*. Cada clase de Java generada contiene métodos asociados para leer y escribir el estado de un objeto[36] desde y hacia el servidor de base de datos, con lo cual, una aplicación en Java puede utilizar estas clases para almacenar y recuperar objetos de la base de datos. El siguiente fragmento de código presenta una clase generada por *JPublisher*:

```
public class JPurchaseOrder implements SQLData {
        ...
        public void readSQL (SQLInput stream, String typeName) throws SQLException {...}
        public void writeSQL (SQLOutput stream) throws SQLException {...}
        ...
}
```

Esta clase ahora puede utilizarse en una aplicación para recuperar objetos de la base de datos, de la siguiente manera:

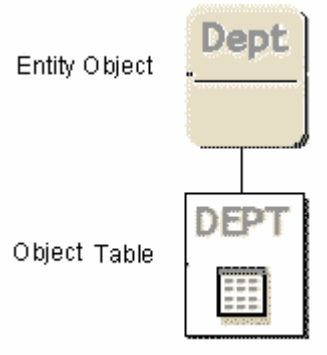
```
ResultSet rs = stmt.executeQuery("select value(p) from purchase_order_tab p");
rs.next();
JPurchaseOrder jp = (JPurchaseOrder) rs.getObject(1);
String streetName = jp.shipAddr.street;
```

Otra opción para manejar los objetos definidos en la base de datos objeto-relacional es a través de *Jdeveloper*[37] y su marco de trabajo integrado, *Business Components for Java* (BC4J[38]), el cual permite desarrollar aplicaciones de base de datos en una arquitectura n-capas a partir de componentes reutilizables modelados como *entity objects*.

Un *entity object* es un objeto real que interactúa en el negocio. Es una clase que encapsula una abstracción del negocio, incluyendo reglas, datos y relaciones entre otros *entity objects*. Un *Entity object* puede ser la clase orden definida en el ejemplo, la cual maneja datos propios, define reglas de comportamiento y maneja relaciones con otras clases que pueden también ser modeladas como *entity objects*.

Dependiendo de la forma en que se desea implementar la aplicación, se puede crear automáticamente los *entity objects* a partir de tablas definidas en la base de datos, o de manera inversa, crear las tablas objeto a partir de los *entity objects*. Por ejemplo, la siguiente figura muestra un *entity object* asociado a una tabla objeto de la base de datos.

Figura 49. *Entity object* asociado a una tabla objeto-relacional



En el catálogo de productos en línea utilizaremos el primer enfoque, definiendo los objetos en la base de datos, para que luego puedan ser generadas las clases de Java con *JPublisher*, o bien generar *entity objects* a través de BC4J. A partir de la definición de los objetos en la base de datos se tiene un proceso de desarrollo sumamente rápido, ya que la lógica del negocio y comportamiento de este se encuentra inmerso en el modelo definido en la base de datos. En la siguiente sección se realizará el mapeo del modelo realizado en UML hacia un esquema objeto-relacional en la base de datos *Oracle* 9i.

3.3.3 Mapeo de modelo en UML hacia esquema objeto-relacional

La capacidad de realizar un mapeo directo del modelo UML hacia un esquema en la base de datos, con el sistema de tipos objeto-relacional definido por *Oracle* 9i es una de las mayores ventajas al implementar un modelo orientado a objetos. A continuación se muestra la creación de los tipos en un esquema de la base de datos, basados en la implementación de *Oracle* 9i para el estándar SQL99. Este proceso lo podemos dividir en dos: la definición de los tipos asociados a cada clase del modelo UML, y la definición de las tablas asociadas a los tipos creados.

Definición de tipos objeto

Basados en la figura 47, empezaremos definiendo los tipos tipoDireccion y tipoTelefono, que representaran las clases Direccion y Teléfono de la siguiente manera:

```
CREATE OR REPLACE TYPE tipoDireccion AS OBJECT (
  calle NUMBER(2),
  avenida NUMBER(2),
  número VARCHAR2(10),
  aldea VARCHAR2(30),
  colonia VARCHAR2(30),
  municipio VARCHAR2(30),
  ciudad VARCHAR2(30),
  pais VARCHAR2(30),
  codigoPostal VARCHAR2(5));

CREATE OR REPLACE TYPE tipoTelefono AS OBJECT (
  codigoArea NUMBER(3),
  número NUMBER(10));
```

Un cliente puede tener más de una dirección y más de un número de teléfono, por lo que se debe manejar una lista de estos objetos, lo cual se implementa mediante colecciones. Una colección es un tipo que nos permite manejar un conjunto de objetos, en forma de *varrays* o *nested tables*, descritas anteriormente. Para manejar el conjunto de direcciones y teléfonos de un cliente utilizaremos el tipo *nested tables*. Primero definimos el nombre del tipo, luego indicamos que es una colección tipo *nested table*, y por último indicamos el tipo de objetos que contendrá dicha colección. La definición de las colecciones de dirección y teléfono es la siguiente:

```
CREATE OR REPLACE TYPE nstDireccion AS TABLE OF tipoDireccion;
CREATE OR REPLACE TYPE nstTelefono AS TABLE OF tipoTelefono;

CREATE OR REPLACE TYPE tipocliente AS OBJECT (
  codigo NUMBER(10),
  nombre VARCHAR2(50),
  direccion nstDireccion,
  telefono nstTelefono,
  MEMBER PROCEDURE agregaDireccion (dir IN tipoDireccion),
  MEMBER PROCEDURE agregaTelefono (tel IN tipoTelefono),
  MEMBER PROCEDURE eliminaDireccion (dir IN tipoDireccion),
  MEMBER PROCEDURE eliminaTelefono (tel IN tipoTelefono))
  NOT INSTANTIABLE NOT FINAL;
```

El tipo tipoCliente representa la clase cliente definida en la figura 47. Debe notarse que esta clase define como atributo una colección tipo nstDireccion y nstTelefono, para manejar las listas de direcciones y teléfonos del cliente. TipoCliente define una abstracción muy general, la cual puede extenderse para tipos de cliente más específicos, como un cliente corporativo. Por esta razón, esta clase debe declararse *NOT INSTANTIABLE NOT FINAL*, lo que indica que esta clase puede extenderse mediante herencia, como se muestra a continuación en la definición de los tipos corporativo y particular, que extienden a tipoCliente:

```
CREATE OR REPLACE TYPE Corporativo UNDER tipoCliente (
  contacto VARCHAR2(20),
  límiteCredito NUMBER(10,2));

CREATE OR REPLACE TYPE Particular UNDER tipoCliente (
  tarjetaCredito VARCHAR2(20),
  tipoID VARCHAR2(20),
  numeroID VARCHAR2(20));
```

A continuación definimos los tipos tipoInventaio y tipoAlmacen, que representan a las clases inventario y almacen respectivamente. Debe notarse que en tipoInventario se define un atributo tipo REF, el cual es una referencia hacia una instancia de tipoAlmacen.

```
CREATE OR REPLACE TYPE tipoAlmacen AS OBJECT (
  codigo NUMBER(5),
  encargadO VARCHAR2(20));

CREATE OR REPLACE TYPE tipoInventario AS OBJECT (
  cantidad NUMBER(10),
  almacen REF tipoAlmacen);
```

La clase categoría será representada por tipoCategoria. Este tipo maneja una lista de productos asociados a cada categoría, lo cual se implementa como una lista de referencias hacia instancias de tipoProducto, que es la representación de la clase producto (esta lista esta implementada por nstProducto,). En tipoProducto se define una referencia hacia una instancia de tipoCategoria para representar la categoría a que pertenece

cada producto. En el siguiente bloque se presenta la definición de estos tipos y otros relacionados:

```
CREATE OR REPLACE TYPE nstInventario AS TABLE OF tipoInventario;
CREATE OR REPLACE TYPE tipoProducto;
CREATE OR REPLACE TYPE nstProducto AS TABLE OF REF tipoProducto;

CREATE TYPE tipoCategoria AS OBJECT (
  codigo NUMBER(5),
  nombre VARCHAR2(30),
  descripcion VARCHAR2(50),
  categoriaSuperior REF tipoCategoria,
  productos nstProducto,
  MEMBER PROCEDURE agregaProducto (prod IN REF tipoProducto),
  MEMBER PROCEDURE eliminaProducto (prod IN REF tipoProducto),
  MEMBER FUNCTION buscaProducto (codigo IN NUMBER) RETURN REF tipoProducto);

CREATE OR REPLACE TYPE tipoProducto AS OBJECT (
  codigo NUMBER(10),
  nombre VARCHAR2(30),
  descripcion VARCHAR2(50),
  marca VARCHAR2(30),
  categoria REF tipoCategoria,
  existencia nstInventario,
  MEMBER FUNCTION peso RETURN NUMBER,
  MEMBER FUNCTION volumen RETURN NUMBER,
  MEMBER FUNCTION precioVenta RETURN NUMBER,
  MEMBER PROCEDURE agregaExistencia (inv IN tipoInventario),
  MEMBER PROCEDURE eliminaExistencia (inv IN tipoInventario),
  MEMBER PROCEDURE agregaProducto (prod IN REF tipoProducto),
  MEMBER PROCEDURE eliminaProducto (prod IN REF tipoProducto))
  NOT INSTANTIABLE NOT FINAL;

CREATE OR REPLACE TYPE tipoDimension AS OBJECT (
  alto NUMBER(10,2),
  ancho NUMBER(10,2),
  grosor NUMBER(10,2),
  peso NUMBER(10,2),
  MEMBER FUNCTION volumen RETURN NUMBER);

CREATE OR REPLACE TYPE productoSimple UNDER tipoProducto (
  precio NUMBER(10,2),
  dimension tipoDimension,
  OVERRIDING MEMBER FUNCTION peso RETURN NUMBER,
  OVERRIDING MEMBER FUNCTION volumen RETURN NUMBER,
  OVERRIDING MEMBER FUNCTION precioVenta RETURN NUMBER);

CREATE OR REPLACE TYPE productoCompuesto UNDER tipoProducto (
  componentes nstProducto,
  OVERRIDING MEMBER FUNCTION peso RETURN NUMBER,
  OVERRIDING MEMBER FUNCTION volumen RETURN NUMBER,
  OVERRIDING MEMBER FUNCTION precioVenta RETURN NUMBER,
  OVERRIDING MEMBER PROCEDURE agregaProducto (prod IN REF tipoProducto),
  OVERRIDING MEMBER PROCEDURE eliminaProducto (prod IN REF tipoProducto));
```

Ahora veremos la representación de las clases pago e item, definidas por los tipos tipoPago y tipoItem respectivamente:

```
CREATE OR REPLACE TYPE tipoPago AS OBJECT (
  monto NUMBER(10,2),
  MEMBER FUNCTION descripcion RETURN VARCHAR2)
  NOT INSTANTIABLE NOT FINAL;

CREATE OR REPLACE TYPE Efectivo UNDER tipoPago (
  OVERRIDING MEMBER FUNCTION descripcion RETURN VARCHAR2);

CREATE OR REPLACE TYPE MoneyOrder UNDER tipoPago (
  emisor VARCHAR2(30),
  OVERRIDING MEMBER FUNCTION descripcion RETURN VARCHAR2);

CREATE OR REPLACE TYPE Tarjeta UNDER tipoPago (
  número VARCHAR2(20),
  fechaVencimiento DATE,
  OVERRIDING MEMBER FUNCTION descripcion RETURN VARCHAR2);

CREATE OR REPLACE TYPE Cheque UNDER tipoPago (
  número VARCHAR2(20),
  banco VARCHAR2(20),
  fecha DATE,
  OVERRIDING MEMBER FUNCTION descripcion RETURN VARCHAR2);

CREATE OR REPLACE TYPE tipoItem AS OBJECT (
  cantidad NUMBER(10),
  precio NUMBER(10,2),
  producto REF tipoProducto);

CREATE OR REPLACE TYPE nstItem AS TABLE OF tipoItem;
CREATE OR REPLACE TYPE nstPago AS TABLE OF tipoPago;
```

Los tipos nstItem y nstPago representan la relación de agregación definida sobre la clase orden y las clases item y pago.

Una clase muy importante para este diseño es estado *(*tipoEstado*)*, ya que define el comportamiento de la clase orden *(*tipoOrden*).* tipoEstado define todos los métodos definidos por tipoOrden y permite que los tipos que lo extiendan definan el comportamiento específico para el estado que representan, los cuales son: anulado, pendiente, cerrado y enProceso.

```
CREATE OR REPLACE TYPE tipoEstado AS OBJECT (
  VALOR NUMBER(1),
  MEMBER FUNCTION total RETURN NUMBER,
  MEMBER PROCEDURE agregaItem (item IN tipoItem),
  MEMBER PROCEDURE eliminaItem (item IN tipoItem),
  MEMBER PROCEDURE agregaPago (pag IN tipoPago),
  MEMBER PROCEDURE eliminaPago (pag IN tipoPago),
  MEMBER PROCEDURE anular,
  MEMBER PROCEDURE cancelar,
  MEMBER PROCEDURE cerrar)
  NOT INSTANTIABLE NOT FINAL;

CREATE OR REPLACE TYPE Anulado UNDER tipoEstado (
  OVERRIDING MEMBER FUNCTION total RETURN NUMBER,
  OVERRIDING MEMBER PROCEDURE agregaItem (item IN tipoItem),
  OVERRIDING MEMBER PROCEDURE eliminaItem (item IN tipoItem),
  OVERRIDING MEMBER PROCEDURE agregaPago (pag IN tipoPago),
  OVERRIDING MEMBER PROCEDURE eliminaPago (pag IN tipoPago),
  OVERRIDING MEMBER PROCEDURE anular,
  OVERRIDING MEMBER PROCEDURE cancelar,
  OVERRIDING MEMBER PROCEDURE cerrar);
CREATE OR REPLACE TYPE Pendiente UNDER tipoEstado ( ...
CREATE OR REPLACE TYPE Cerrado UNDER tipoEstado ( ...
CREATE OR REPLACE TYPE EnProceso UNDER tipoEstado ( ...
```

Por último definimos tipoOrden, y la implementación de sus métodos. Los métodos son implementados con la instrucción *CREATE OR REPLACE TYPE BODY nombre_tipo AS métodos...* como se muestra a continuación:

```
CREATE OR REPLACE TYPE tipoOrden AS OBJECT (
  número NUMBER(10),
  fecha DATE,
  cliente REF tipoCliente,
  estado tipoEstado,
  pagos nstPago,
  items nstItem,
  MEMBER FUNCTION total RETURN NUMBER,
  MEMBER PROCEDURE agregaItem (item IN tipoItem),
  MEMBER PROCEDURE eliminaItem (item IN tipoItem),
  MEMBER PROCEDURE agregaPago (pag IN tipoPago),
  MEMBER PROCEDURE eliminaPago (pag IN tipoPago),
  MEMBER PROCEDURE anular,
  MEMBER PROCEDURE cancelar,
  MEMBER PROCEDURE cerrar);

CREATE OR REPLACE TYPE BODY tipoOrden AS
  MEMBER FUNCTION total RETURN NUMBER IS
   BEGIN
          RETURN (estado.total());
   END;
  MEMBER PROCEDURE agregaItem (item IN tipoItem) IS
   BEGIN
          estado.agregaItem(item);
   END;
  MEMBER PROCEDURE eliminaItem (item IN tipoItem) IS
   BEGIN
          estado.eliminaItem(item);
   END;
  MEMBER PROCEDURE agregaPago (pag IN tipoPago) IS
   BEGIN
          estado.agregaPago(pag);
```

```
 END;
MEMBER PROCEDURE eliminaPago (pag IN tipoPago) IS
 BEGIN
        estado.eliminaPago(pag);
 END;
MEMBER PROCEDURE anular IS
 BEGIN
        estado.anular;
 END;
MEMBER PROCEDURE cancelar IS
 BEGIN
        estado.cancelar;
 END;
MEMBER PROCEDURE cerrar IS
 BEGIN
        estado.cerrar;
 END;
END;
```

Definición de tablas objeto

Hasta este punto solamente se han definido los tipos que representan los objetos del negocio, pero no hemos definido el manejo de la persistencia de estos objetos. La tecnología objeto-relacional nos permite definir tablas en base a los tipos creados.

Identificamos los tipos para los cuales debemos manejar la persistencia de sus instancias, y asociamos una tabla para cada uno. En una tabla objeto, cada fila representara una instancia del tipo que representa. Por ejemplo, definimos una tabla objeto para manejar la persistencia de las instancias de tipoCliente. Una gran ventaja de este tipo de tabla es que si posteriormente a su creación se definen nuevos subtipos de tipoCliente, esta tabla podrá almacenar las instancias de todos los nuevas tipos.

```
CREATE TABLE cliente OF tipoCliente
  (CONSTRAINT PKCLIENTE PRIMARY KEY(codigo))
  NESTED TABLE direccion STORE AS cli_nstDireccion,
  NESTED TABLE telefonO STORE AS cli_nstTelefono;
```

Para manejar la persistencia de las colecciones tipo *nested tables*, estas deben declararse en la creación de la tabla. *Oracle* crea una tabla anidada para manejar cada colección. Por ejemplo, esta instrucción crea tres tipos de objeto dentro del esquema de la base de datos: las tablas cliente,

cli_nstDirección y cli_nstTeléfono. Las tablas cli_* no podrán ser accedidas directamente (esto es controlado por el manejador de base de datos para evitar que los datos sean corrompidos o manipulados por otros objetos que no son los dueños de los datos).

Además de tipoCliente debemos crear tablas para tipoCategoría, tipoAlmacén, tipoProducto y tipoOrden, de la siguiente manera:

```
CREATE TABLE Categoria OF tipoCategoria
  (CONSTRAINT pkCategoria PRIMARY KEY(codigo))
  NESTED TABLE productos STORE AS cat_nstProducto;

CREATE TABLE Almacen OF tipoAlmacen
  (CONSTRAINT PKALMACEN PRIMARY KEY (CODIGO));

CREATE TABLE Producto OF tipoProducto
  (CONSTRAINT pkProducto PRIMARY KEY (codigo))
  NESTED TABLE existencia STORE AS pro_nstInventario;

CREATE TABLE Orden OF tipoOrden
  (CONSTRAINT pkOrden PRIMARY KEY(número))
  NESTED TABLE pagos STORE AS ord_nstPago,
  NESTED TABLE items STORE AS ord_nstItem;
```

A partir de este punto, se puede iniciar la fase de implementación a nivel de aplicación a través de *JDevelopper*, con una de las dos opciones que provee: Su *framework* de desarrollo BC4J o a través de *JPublisher* para generar clases de Java a partir de los tipos definidos en la base de datos.

3.4 Puntos específicos de la aplicación de los patrones de diseño

A continuación se describen los patrones utilizados y los puntos del diseño donde se implementaron.

Composite

Este patrón se utilizo en las clases producto, productoSimple y productoCompuesto, con el objetivo de obtener los beneficios que representa este patrón, entre los cuales tenemos la facilidad para manipular objetos simples y objetos compuestos de la misma forma y permitir la formación de jerarquías de productos.

La clase producto, define los métodos y atributos elementales de un producto, pero no implementa los métodos, los cuales deben ser implementados específicamente por sus clases hijas.

ProductoSimple define algunos atributos adicionales específicos de un producto simple, e implementa los métodos definidos por producto a excepción de agregarProducto*()* y eliminarProducto*()*. Los métodos son implementados en base a los atributos de esta clase, por ejemplo, precioVenta*()* devuelve el valor de su atributo peso y peso devuelve el valor de su atributo dimensiones.peso.

ProductoCompuesto también implementa los métodos definidos por producto pero lo hace en base a los productos que lo conforman, por ejemplo, la implementación de precioVenta*()* debe ser la suma del precio de los productos que lo componen, y de igual manera se implementaría peso*()* y volumen*()*.

State

La clase orden puede presentar distinto comportamiento dependiendo de su estado. Los posibles estados que presenta una clase pueden ser: pendiente*,* cerrada o anulada*,* además la transición de un estado a otro debe ser controlada, ya que una orden anulada no puede pasar a estar

cerrada o pendiente. Una orden pendiente puede agregar pagos, pero no puede agregar o eliminar productos, y una orden cerrada no puede agregar productos o pagos. Al implementarse cada método de la clase orden, este tipo de condiciones se convertirían en una serie de if's que no promueven la reutilización y modificación posterior del código. Por ejemplo, en cada método deberá ir una serie de *ifs*, de la siguiente forma:

```
If estado="PENDIENTE" then
        //serie de acciones;
if estado="ANULADA" then
        //serie de acciones
...
```

Esta solución, aparte de no ser elegante y no promover la facilidad de interpretación del código, límita la posibilidad de agregar en el futuro nuevos estados para una orden y definir un comportamiento específico para dicho estado, ya que se está definiendo de manera explicita (*hardCode*) el comportamiento de la clase orden.

Por los motivos anteriormente expuestos se implementó el patrón *state*, el cual permite a la clase orden definir su comportamiento en base a su estado actual, y la implementación de sus métodos no conlleva una serie de *ifs*, si no que están ligados al estado de la clase, de la siguiente manera:

```
MEMBER FUNCTION total RETURN NUMBER IS ...
        RETURN (estado.total()); ...
MEMBER PROCEDURE agregaItem (item IN tipoItem) IS ...
        estado.agregaItem(item);
MEMBER PROCEDURE eliminaItem (item IN tipoItem) IS ...
        estado.eliminaItem(item); ...
MEMBER PROCEDURE agregaPago (pag IN tipoPago) IS ...
        estado.agregaPago(pag);  ...
MEMBER PROCEDURE eliminaPago (pag IN tipoPago) IS ...
        estado.eliminaPago(pag); ...
MEMBER PROCEDURE anular IS ...
        estado.anular; ...
MEMBER PROCEDURE cancelar IS ...
        estado.cancelar; ...
MEMBER PROCEDURE cerrar IS ...
        estado.cerrar; ...
```

Singleton

Este patrón puede ser implementado con las subclases de estado, ya que estas clases solo representan un comportamiento en particular par la clase orden pero no almacenan atributos o valores de importancia para ella.

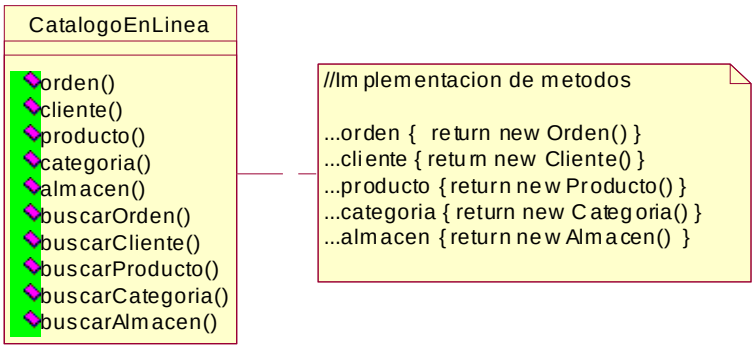
Se puede crear instancias únicas de cada una de las subclases de estado y que las instancias de orden apunten hacia la instancia que represente su estado actual.

Facade

Este patrón se introdujo en la implementación del *framework*, descrito en la figura 3.6, a través de la clase CatálogoEnLinea. Esta clase simplemente provee una interfaz de mayor nivel para acceder a las clases que componen el sistema de catálogo de productos, facilitando la comunicación y ocultando la complejidad del sistema a las clases que a nivel de aplicación hagan uso de las clases que conforman el catálogo. Como se puede ver, define métodos para acceder directamente a las clases más importantes del sistema.

En la siguiente figura se presenta la clase CatálogoEnLinea de manera más amplia:

Figura 50. Definición de clase catálogoEnLinea



Esta clase puede ser implementada con el patrón *singleton* al igual que las subclases de estado.

Actualmente en Guatemala las características de objetos que proporciona la base de datos *Oracle*[39] no son muy utilizadas, posiblemente por falta de conocimiento. Distintas herramientas propietarias de *Oracle* hacen uso de estas características, como por ejemplo: *Internet File System IFS*, *Workflow*, *Oracle Internet Directory* OID, *Single Sign On* SSO y XML DB entre otras. Esta última, XML *DataBase* es una herramienta que permite almacenar datos en formato XML dentro de la base de datos en forma estructurada, utilizando las características objeto-relacional.

La base de datos *Oracle* posee dos arquitecturas para almacenar datos en XML: con XML DB, y mediante CLOB's[40]. Ambas arquitecturas almacenan los datos dentro de la base de datos, pero XML DB define un esquema para manipular los datos en XML, utilizando tipos y tablas objeto, lo cual proporciona un mejor desempeño de la aplicación. Utilizar la opción de almacenar XML en la base de datos mediante CLOB prácticamente condena a todas las consultas realizadas sobre los datos a efectuar por lo menos una lectura física, ya que los CLOB's no son colocados en *cache* por la base de datos, a diferencia de los datos almacenados de forma estructurada. *Oracle*

Internet Directory OID[41], es otra poderosa herramienta que hace uso de las características objeto-relacional en la definición de su esquema dentro de la base de datos.

Como usuarios finales de este tipo de productos posiblemente no nos interese la forma en que se almacena la información (relacional u objeto-relacional) ya que no manipularemos directamente los esquemas definidos, pero si es importante saber que *Oracle* hace uso de la tecnología objeto-relacional en la mayoría de herramientas y soluciones que provee. Esto nos da una idea de la importancia y gran potencial que presenta las características de objetos dentro de una base de datos, y nosotros también podemos aprovecharlas en la implementación de soluciones comunes en nuestro medio.

CONCLUSIONES

1. El conocimiento y entendimiento de los patrones de diseño nos ayuda a generar modelos de objetos completos para el dominio de la aplicación que se desarrolla, fáciles de modificar y extender, para implementar los constantes cambios que se dan en los requerimientos.

2. La herencia como mecanismo de reutilización es una poderosa técnica para extender o ampliar la funcionalidad de un sistema siempre que sea utilizada de forma adecuada. Debe utilizarse para definir una interfaz común para familias de objetos con diferente implementación y debe realizarse únicamente de clases abstractas o de clases que posean poca implementación, para evitar una fuerte dependencia entre clases padre e hijas, ya que esto limita la flexibilidad y reusabilidad en el diseño.

3. Utilizar la composición de objetos en mayor grado que la herencia de clases como un medio para obtener nueva funcionalidad ayuda a mantener la propiedad de encapsulación de las clases, enfocar estas clases en tareas específicas y mantener las jerarquías de clases pequeñas. Además permite definir la funcionalidad de un objeto en tiempo de corrida, lo cual no es permitido con la herencia de clases.

4. Un sistema correctamente implementado, basado en los distintos patrones no requerirá mayores modificaciones cuando se produzcan cambios en los requerimientos, ya que al crear un sistema utilizando los patrones de diseño, la estructura básica del *software* es flexible y reutilizable, pues se ha tomado en cuenta en la etapa de diseño los posibles cambios que pueden surgir en el futuro para la aplicación.

5. Conocer los patrones ayuda a mejorar la comunicación entre los distintos diseñadores y desarrolladores de *software*, a través de un vocabulario en común, el cual lo proporcionan los patrones de diseño y el UML.

6. Una base de datos objeto-relacional permite realizar un mapeo directo de un diseño orientado a objetos hacia una aplicación de base de datos, lo que incluye algunas de las ventajas que ofrece la metodología de objetos, como lo son el encapsulamiento, polimorfismo y herencia. Esto permite manejar el mismo nivel de abstracción a nivel de almacenamiento de los datos (*storage*) y a nivel de la lógica de la aplicación, pues de lo contrario se debe realizar una traducción de los tipos y clases definidas en la lógica de la aplicación hacia la lógica de los datos, definida en una base de datos relacional como tablas, campos, llaves foráneas, etc.

7. Al implementar un modelo de objetos en una base de datos objeto-relacional se deben implementar los métodos sobre los objetos definidos, los cuales definen las reglas del negocio, mientras que en una base de datos relacional se debe definir las tablas que almacenarán los datos, y las reglas del negocio deben implementarse en el nivel de la aplicación, no en el de implementación de la base de datos. Con esto vemos que en una base de datos relacional se maneja de forma separada los datos y los procesos que se aplican a estos, pero en una base de datos objeto-relacional se integran los datos y los procesos en un solo modelo, definido en forma de objetos (datos) y métodos (procesos), lo cual brinda un menor tiempo de desarrollo y menor complejidad a nivel de la aplicación. Esto posiblemente implique mejorar el desempeño de la base de datos ya que habría más carga de trabajo sobre esta.

RECOMENDACIONES

1. Implementar dentro del plan de estudios de la carrera de Ciencias y Sistemas como un tema adicional en el área de desarrollo de *software* el estudio de los patrones de diseño, con el objetivo de promover su utilización y así facilitar la comunicación entre los diseñadores y desarrolladores de *software*.

2. Continuar investigando, estudiando y aplicando los distintos patrones que surgen de la experiencia de otros diseñadores y desarrolladores en las diferentes ramas de la ingeniería de *software* para obtener los beneficios derivados de estos, y probablemente podremos contribuir documentando patrones que sean producto de la experiencia propia.

3. No aplicar los patrones de diseño de forma indiscriminada, ya que esto puede complicar el diseño y la implementación de un sistema. El objetivo de los patrones es generar sistemas reutilizables, portables, escalables, con código legible, entre otros, pero alcanzar estos objetivos puede generar mayores problemas de diseño e implementación, si no se aplican de la manera correcta en la situación adecuada.

REFERENCIAS

1. Erich Gamma, Richard Helm, John Vlissides y Ralph Jonson. *Design Patterns, Elements of Reusable Object-Oriented Software* (Addison-Wesley, 994).

2. Christopher Alexander (977).

3. Para mayor información sobre estos patrones o estilos de arquitectura consultar: David Garlan, Mary Shaw. *An Introduction to Software Architecture* (*School of Computer Science*, Carnegie Mellon University, Pittsburgh, 994)

4. Este patrón tiene como objetivo definir los tipos de objetos a crear utilizando una instancia como prototipo, y crear nuevos objetos simplemente copiando o clonando este prototipo. Este patrón se encuentar detallado en la seccion 2.4.4.

5. El detalle de la implementación de este método en Java se explicara en la sección 2.4.4 Prototype.

6. Para mayor información sobre patrones organizacionales consultar: Yonat Sharon, http://www.kinetica.com/home/yon/OrgPat.html.

7. Para mayor información sobre patrones de proceso consultar: Scott W. Ambler. *An Introduction to Process Patterns* (*Object-Oriented Consultant AmbySoft* Inc., 998)

8. Ver sección 1.3.2

9. *Gang of Four*

10. La utilización de un lenguaje común para documentar los diseños, como el UML ayuda a incrementar la legibilidad de *software* altamente parametrizado.

11. Clasificación de los patrones según su propósito (creación, estructura y comportamiento) y según su alcance (clase u objeto).

12. Los nombres, métodos y atributos de las clases presentadas de aquí en adelante serán definidos en ingles.

13. Un *framework* es conjunto de clases relacionadas que describen una estructura de *software* fácilmente ampliable y reutilizable

14. Las jerarquías de clase paralelas se presentan cuando una clase delega responsabilidades a otra clase.

15. Definido en la sección 2.4. Ver figura 10 y la codificación del modelo.

16. *Toolkit* es un conjunto de clases relacionadas y reutilizables que proveen alguna utilidad de propósito general.

17. *Look-and-Feel* se utiliza para definir un sistema de interfaz para el usuario. Entre ellos tenemos Motif que es el sistema de interfaz del usuario estándar para estaciones Unix, y *Presentation Manager*, entre otros.

18. *Hard-code*. Codificación inflexible.

19. Patrón de creación que permite crear una sola instancia de una clase. Ver en la sección 2.4.5

20. Método encargado de la fabricación o creación de un objeto. Ver sección 2.4.

21. Formato de texto enriquecido, *Rich Text Format*.

22. Un *Garbage Collector* es un proceso encargado de eliminar instancias de clases a las cuales no se tiene ninguna referencia, es decir, objetos aislados.

23. Clase que define diversos métodos para controlar el estado y comportamiento de la JVM.

24. Un *framework* o armazón es un conjunto de clases relacionadas que describen una estructura de *software* fácilmente ampliable y reutilizable.

25. *Rational Unified Process*. Es un proceso de ingeniería de *software*.

26. El modelado del negocio tiene como objetivo entender la estructura dinámica y estática de negocio, identificar problemas actuales y posibles soluciones y asegurarse que todos los usuarios comprendan el funcionamiento del negocio. En base a esto, se definen procesos, roles y responsabilidades.

27. Técnica para describir para que son los datos, en lugar de su apariencia o como se denominan.

28. Esta característica se encuentra disponible desde la base de datos Oracle 8i.

29. Además de Java, soporta PL/SQL y C/C++.

30. Oracle 9i implementa el estándar para bases de datos objeto-relacional, definido en ANSI SQL99.

31. La evolución de tipos es un mecanismo que permite cambiar la definición de un tipo (clase) y propagar estos cambios a todos los objetos que hagan referencia a el.

32. Lenguaje de manipulación de datos (DML) definido por SQL.

33. Es una herramienta escrita completamente en Java que genera clases de Java para representar entidades definidas por el usuario en la base de datos en una aplicación de Java. Estas entidades son: tipos objeto SQL, tipos referencia (REF), colecciones (VARRAY y *Nested Tables*) y paquetes PL/SQL.

34. Para realizar el mapeo hacia C/C++ Oracle provee la herramienta OTT (*Object Type Translator*).

35. SQL99 provee el estándar para la implementación de una base de datos Objeto-Relacional.

36. Valor de los atributos de un objeto.

37. Entorno de desarrollo integrado (IDE) para desarrollo y *deployment* de aplicaciones Java y XML. Para mas información sobre esta herramienta consultar: www.otn.oracle.com/documentation.

38. BC4J es un *framework* de desarrollo propietario de Oracle para crear aplicaciones que cumplen con el estandar J2EE (J2EE *Compliant*).

39. Características proporcionadas a partir de la versión 8i.

40. *Character Large Object* (CLOB): Tipo definido por Oracle para almacenar dentro de la base de datos archivos de gran tamaño.

41. OID es un servicio de directorio en internet que implementa el protocolo LDAP (*Lightweight Directory Access Protocol*) sobre una base de datos Oracle.

BIBLIOGRAFÍA

1. COOPER, James W. ***The design patterns java companion***. Editorial Addison-Wesley, 1998. 218pp.

2. GAMA, Helm y otros. ***Design patterns. Elements of reusable object-oriented software***. Editorial Addison-Wesley, 1995. 395pp.

3. HERNÁNDEZ Tejada, David. **Guía de patrones de diseño.** www.teleprogramadores.com/patrones, 2003.

4. LEE, ***Geoff. Simiple strategies for complex data: Oracle 9i object-relational technology***, 2002. 22pp.

5. ***Oracle 9iAS MVC Framework for J2EE User Guide***. 2003. 257pp.

6. ***Oracle 8i Application Developer's Guide – Object-Relational Features Release 2***.

7. ***Oracle 9iAS Documentation Library Release 2***.

8. ORNELIS Hoil, Edgar Rene. **Sistema de control de becas préstamo, un análisis y diseño orientado a objetos**. Tesis Ing. Ciencias y Sistemas. Guatemala, Universidad de San Carlos de Guatemala, Facultad de Ingeniería, 1998. 119pp.

9. ROJAS Morales, Claudia Liceth. **Estructura interna de las bases de datos orientadas a objetos**. Tesis Ing. Ciencias y Sistemas. Guatemala, Universidad de San Carlos de Guatemala, Facultad de Ingeniería, 1997. 101pp.

10. SCHILDT, Herbert. **Java 2. Manual de Referencia**. 4ta. Edición. Editorial McGraw-Hill, 2001. 959pp.

11. VENNAPUSA, Priya. ***Oracle 9i: XML Fundamentals for developers***. 2002.