UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ACATLÁN

PATRONES DE DISEÑO PARA EL DESARROLLO DE SOFTWARE ORIENTADO A OBJETOS

TESINA

QUE PARA OBTENER EL TÍTULO DE
LIC. EN MATEMÁTICAS APLICADAS Y COMPUTACIÓN

PRESENTA

EDGARDO ZAVALA VARGAS

ASESOR: Mtro. RUBÉN ROMERO RUIZ

DICIEMBRE 2011

Agradecimientos

A Dios:

Por amarme, por darme la vida y todas las cualidades buenas que el ser humano puede encontrar en mí.

A mis Padres:

Por su preocupación, apoyo y amor incansable; porque gracias a ellos se formó un carácter único y especial.

A mis Profesores:

Por su servicio, madurez, conocimiento y experiencia que se ven reflejados en sus alumnos, especialmente en mí.

Índice

Introducción 1

Breve historia de los patrones 2

¿Qué son los patrones de diseño? 3

Principios de la metodología Orientada a Objetos (OO) 4

¿Por qué estudiar patrones de diseño? 6

¿Cómo usar un patrón de diseño? 7

Capítulo 1: Lenguaje Unificado de Modelado 8

1.1 ¿Qué es el UML? 8

1.2 Importancia del UML 8

1.3 Terminología y conceptos UML 9

1.4 Diagramas del UML 12

Capítulo 2: Patrones de Creación 20

2.1 Abstract Factory (Fábrica Abstracta) 20

2.2 Builder (Constructor) 26

2.3 Factory Method (Método de Fabricación) 31

2.4 Prototype (Prototipo) 34

2.5 Singleton (Instancia Única) 38

Capítulo 3: Patrones Estructurales 41

3.1 Adapter (Adaptador) 41

3.2 Bridge (Puente) 45

3.3 Composite (Compuesto) 49

3.4 Decorator (Decorador) 55

3.5 Facade (Fachada) 59

3.6 Flyweight (Peso Ligero) 63
3.7 Proxy (Apoderado) 68
Capítulo 4: Patrones de Comportamiento 72
4.1 Chain of Responsibility (Cadena de Responsabilidad) 72
4.2 Command (Orden) 76
4.3 Interpreter (Intérprete) 81
4.4 Iterator (Iterador) 86
4.5 Mediator (Mediador) 90
4.6 Memento (Recuerdo) 94
4.7 Observer (Observador) 99
4.8 State (Estado) 104
4.9 Strategy (Estrategia) 109
4.10 Template Method (Método Plantilla) 114
4.11 Visitor (Visitante) 117
Conclusiones 124
Anexos 125
Anexo A. Índice de código fuente 125
Anexo B. Índice de figuras (Diagramas UML) 126
Anexo C. Catálogo de patrones de diseño. 128
Glosario 130
Bibliografía 135

¿Cómo está organizada la tesina?

La tesina está divida en dos partes principales. La primera parte (Introducción y Capítulo 1) describe qué son los patrones de diseño y su importancia, introduce las bases para entender la terminología usada en el diseño orientado a objetos, así como un resumen breve de los diagramas del lenguaje unificado de modelado utilizados en este documento para ilustrar de forma gráfica la mayor parte de los patrones de diseño.

En la *Introducción*, se presenta la historia de los patrones, se estudia el significado de los patrones de diseño, la importancia de estudiar patrones de diseño y definiciones de la metodología orientada a objetos.

En el *Capítulo 1: Lenguaje Unificado de Modelado*, se describen los conceptos básicos del lenguaje y un resumen de los principales tipos de diagramas del UML.

La segunda parte (Capítulos 2, 3 y 4) es un catálogo de los patrones de diseño en los cuales se utilizan diagramas UML para describir gráficamente la solución del patrón. Si bien el diagrama es suficiente para poder aplicar la solución en cualquier lenguaje de programación orientado a objetos, se ha utilizado el lenguaje de programación Java en el código fuente de ejemplo [SM01] que se proporciona. La elección del lenguaje de programación es importante, ya que influye en el punto de vista y esa elección determina lo que puede implementarse o no fácilmente.

En el *Capítulo 2: Patrones de Creación*, se discuten los patrones que tienen que ver con la creación de objetos tales como: Abstract Factory, Builder, Factory Method, Prototype, y Singleton.

En el *Capítulo 3: Patrones Estructurales*, nos enfocamos en los patrones que ofrecen una estructura a nuestra aplicación. Tratan con la forma en la que se componen las clases y objetos. Algunos ejemplos son: Adapter, Bridge, Composite, Decorator, Facade, etc.

En el *Capítulo 4: Patrones de Comportamiento*, analizamos los patrones de comportamiento que se caracterizan por el modo en que las clases y objetos interactúan, y la manera en que se distribuyen responsabilidades.

En la sección *Anexos*, incluimos el índice de código fuente, índice de figuras y un catálogo resumido de patrones de diseño presentados a lo largo del documento.

Introducción

La parte más difícil en el desarrollo de software no radica en la codificación o creación de código, el trabajo delicado radica en las primeras decisiones que se realizan en la etapa de diseño. Los encargados de tomar decisiones en el diseño lidian con el sistema por el resto del tiempo de vida del sistema.

Creemos que el diseño es la fase crítica en el ciclo de vida de desarrollo de software [YA03]. Buenas decisiones de diseño resultan a la larga en un buen producto y malas decisiones de diseño generalmente afectan la calidad del producto final. Pero, ¿cómo saber si se están tomando buenas decisiones en la etapa de diseño de desarrollo de software, y cómo podemos afirmar que nuestra decisión fue la correcta cuando aún no se tiene el producto final para probar nuestras tempranas decisiones de la etapa de diseño?

Los patrones de diseño prometen servirnos como un consejo proveniente de los expertos en diseño. Proporcionan las respuestas que se necesitan, como "Ya he implementado esto, pero no funciona porque…", ó "Ya he implementado esto antes, y esto es lo que hace, esta implementación tiene éstas ventajas y éstas desventajas". Los diseñadores experimentados y desarrolladores de software orientado a objetos han implementado y probado soluciones a problemas recurrentes. Los patrones de diseño capturan sus experiencias y presentan las soluciones a todos los diseñadores en una forma que define qué problema está siendo resuelto, cómo se resuelve, qué solución es la mejor y las implicaciones de usar esa solución.

En la vida laboral, no se pueden tomar buenas decisiones de diseño a menos que se tenga la experiencia necesaria que nos capacite para tomar esas buenas decisiones. Las decisiones que se tomen en la fase de diseño son cruciales para el desarrollo de la aplicación. Experimentados desarrolladores de software y diseñadores tiene esta experiencia, y ellos la transmiten a nosotros en la forma de patrones de diseño.

Algo que los expertos saben que no hay que hacer [GHJ95] es resolver cada problema partiendo desde cero. Por el contrario reutilizan soluciones que ya les han sido útiles en el pasado. Cada vez que encuentran una solución buena, la utilizan una y otra vez. Esa experiencia es parte de lo que los convierte en expertos. Muchos de los sistemas orientados a objetos se encuentran con patrones recurrentes de clases y comunicaciones entre objetos. Estos patrones resuelven problemas específicos de diseño y hacen que los diseños orientados a objetos sean más flexibles, elegantes y lo mejor de todo, reutilizables. Los patrones ayudan a los desarrolladores o diseñadores a reutilizar buenos diseños y basar los nuevos diseños en la experiencia previa, de esta

forma un diseñador familiarizado con estos patrones puede fácilmente aplicarlos en los problemas de diseño sin tener que comenzar desde cero. Es sumamente importante conocer y usar patrones de diseño para el desarrollo de software orientado a objetos porque cuando se carece de esa experiencia, que muchas veces se obtiene a lo largo de los años, los patrones de diseño hacen que sea más fácil obtener algo que los expertos saben, a saber, reutilizar buenos diseños y arquitecturas. Ya los desarrolladores expertos han dejado constancia de su experiencia en el diseño de software para que otros la usen a través de los patrones de diseño. Los patrones de diseño que son presentados a lo largo de este documento representan sólo una porción de lo que los expertos saben, aunque los diseños no son nuevos, se han expresado de la forma más sencilla posible, en forma de un catálogo.

Breve historia de los patrones

La idea de patrones de diseño en el software tiene su origen en el campo de la arquitectura. Christopher Alexander, un arquitecto, escribió dos libros revolucionarios que describen patrones en la arquitectura de edificación [AIS77] y planeación urbana: "*Un Lenguaje de Patrones: Ciudades, Edificios, Construcción*" y "*El Eterno Camino de Edificación*". Las ideas presentadas en estos libros son aplicables a un gran número de campos fuera de la arquitectura, incluyendo el desarrollo de software.

En 1987, Ward Cunningham y Kent Beck [Gra02] usaron algunas de las ideas de Alexander para desarrollar cinco patrones para el diseño de interfaz de usuario. Como resultado de su trabajo, dieron una presentación titulada "El uso de Lenguajes de Patrones para la Programación Orientada a Objetos" en la conferencia del OOP-SLA '87 por sus siglas en inglés *Object Oriented Programing Systems, Languages and Aplications*. Tiempo después varios documentos y presentaciones con relación a los patrones fueron publicados por distinguidas eminencias en mundo de la Orientación a Objetos (OO).

A principios de los 90's, Gamma, Richard Helm, John Vlissides, y Ralph Jonson comenzaron el trabajo de uno de los libros de computación con más influencia de la década pasada: "*Patrones de Diseño*". Libro publicado en 1994 en su primera edición y después llamado "La pandilla de los cuatro (Gang of Tour)", o "GoF", el libro populariza la idea de los patrones y es en mayor parte la influencia de esta tesina. En este libro [GHJ95] los autores documentan 23 patrones que encuentran en su trabajo de aproximadamente cuatro años y medio. Desde entonces, muchos otros libros han sido publicados, capturando patrones de diseño y otras muy buenas prácticas de la ingeniería de software.

¿Qué son los patrones de diseño?

En general, un patrón de diseño describe un problema que ocurre frecuentemente en el desarrollo o implementación de software, y además proporciona la solución a ese problema de tal manera que dicha solución pueda ser reutilizada.

Los patrones fueron introducidos para documentar las buenas prácticas de diseño; son los vehículos del conocimiento y experiencia transmitidos del experto hacia el novato.

Según Christopher Alexander, "cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema, de tal modo que se puede aplicar esta solución un millón de veces, sin hacer lo mismo dos veces" [AIS77]. Aunque Alexander se refería a patrones en ciudades y edificios, lo que dice también es válido en patrones de diseño orientados a objetos. Nuestras soluciones se expresan en términos de objetos e interfaces, en vez de paredes y puertas, pero en la esencia de ambos tipos de patrones se encuentra una solución a un problema dentro de un contexto.

Según Erich Gamma (y colaboradores) en su libro "*Patrones de Diseño*" [GHJ95] un patrón tiene cuatro elementos esenciales:

1. El *nombre del patrón* permite describir, en una o dos palabras, un problema de diseño junto con sus soluciones y consecuencias. Cuando se da nombre a un patrón, se incrementa nuestro vocabulario de diseño y esto nos permite diseñar con mayor abstracción. Tener un vocabulario de patrones nos permite hablar de ellos con otros colegas, mencionarlos en nuestra documentación y tenerlos en cuenta nosotros mismos. Así, nos resulta más fácil pensar en nuestros diseños y transmitirlos a otros, junto con sus ventajas y desventajas.
2. El *problema* describe en qué momento aplicar el patrón. Explica el problema y su contexto. Puede describir problemas concretos de diseño, así como las estructuras de clases u objetos que son características de un diseño inflexible. A veces el problema incluye una serie de condiciones que deben cumplirse para que tenga sentido aplicar el patrón.
3. La *solución* describe los elementos que forman parte del diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño o una implementación en concreto, más bien, un patrón es como una plantilla que puede aplicarse en muchas situaciones diferentes. El patrón proporciona una descripción abstracta de un problema de diseño y la forma en que lo resuelve una disposición general de elementos, en nuestro caso, las clases y objetos.

4. Las *consecuencias* son los resultados, tanto ventajas e inconvenientes de aplicar el patrón. Aunque cuando se describen decisiones de diseño muchas veces no se reflejan sus consecuencias, éstas son fundamentales para evaluar alternativas de diseño y comprender los costes y beneficios de aplicar el patrón. Las consecuencias en el software comúnmente se refieren al equilibrio entre espacio y tiempo. También pueden tratar cuestiones de lenguaje e implementación y, por otro lado, puesto que la reutilización suele ser una característica de los diseños orientados a objetos, las consecuencias de un patrón incluyen su impacto sobre flexibilidad, extensibilidad y portabilidad de un sistema. Incluir estas consecuencias de forma explícita ayudará a advertirlas y evaluarlas.

Se dividieron éstos patrones de diseño en tres tipos: patrones de creación, estructurales y de comportamiento.

Los ***patrones de creación*** son los que crean objetos por nosotros, en vez de tener que instanciar los objetos directamente. Esto da a nuestros programas más flexibilidad para decidir que objetos necesitan ser instanciados dependiendo de un caso dado.

Los ***patrones estructurales*** ayudan a componer grupos de objetos en estructuras más grandes, tal como interfaces de usuario complejas o arreglos de datos.

Los ***patrones de comportamiento*** ayudan a definir la comunicación entre objetos en nuestro sistema y la forma en que el flujo se controla en un programa complejo.

Téngase en cuenta que se estarán utilizando diagramas UML y ejemplos en Java para estos patrones de diseño orientados a objetos en los capítulos 2, 3 y 4.

Principios de la metodología Orientada a Objetos (OO)

La razón fundamental de usar los diferentes patrones de diseño es el de mantener las clases separadas y prevenirlas de tener información de más con respecto a otra clase. Existen un sin número de estrategias que un programador OO usa para conseguir ésta separación, entre ellas la encapsulación y la herencia.

La mayoría de los lenguajes con la capacidad OO soporta herencia. Una clase que hereda de una clase padre tiene acceso a todos los métodos de esa clase padre. Además tiene acceso a todas sus variables no privadas. La *herencia* de clases no es más que un mecanismo para extender la funcionalidad de una aplicación [GHJ95] reutilizando la funcionalidad de las clases padres. También es

importante la capacidad de la herencia para definir familias de objetos con interfaces idénticas (común cuando se hereda de clases abstractas), que es en lo que se basa el polimorfismo. Cuando la herencia se usa correctamente, todas las clases que derivan de una clase abstracta compartirán su interfaz. Así todas ellas se convierten en subtipos de la clase abstracta, logrando entonces responder todas ellas a las peticiones en la interfaz de su clase abstracta.

Cuando se manipulan los objetos solamente en términos de la interfaz definida por las clases abstractas se tienen dos ventajas. La primera es que los clientes no tienen que conocer los tipos específicos de los objetos que usan, es suficiente con que éstos se adhieran a la interfaz que esperan los clientes. La segunda ventaja es que como los clientes desconocen las clases que implementan dichos objetos; sólo conocen las clases abstractas que definen la interfaz. Esto reduce de tal manera las dependencias de implementación que nos lleva al siguiente principio del diseño orientado a objetos reutilizable: *Programe para una interfaz, no para una implementación*. Esto significa que no se deben declarar variables como instancias de clases concretas, más bien ajustarse a la interfaz definida por una clase abstracta.

La mayoría de la gente conoce los términos objetos, interfaces, clases y herencia. La dificultad radica en aplicarlos para construir software flexible y reutilizable, y los patrones de diseño nos pueden mostrar cómo hacerlo.

La *composición* de objetos es una alternativa a la herencia de clases, esto significa que la nueva funcionalidad se obtiene ensamblando o componiendo objetos para obtener funcionalidades más complejas. Aunque la herencia y la composición tienen sus ventajas, también tienen sus desventajas o inconvenientes. Por ejemplo, como la herencia expone a una subclase los detalles de la implementación de su padre, suele decirse que "la herencia rompe la encapsulación" [Sny86]. La implementación de una subclase se liga de tal forma a la de su clase padre que cualquier cambio a la implementación de la clase padre afectará y obligará a cambiar la subclase. La composición de objetos se define en tiempo de ejecución y requiere que los objetos tengan en cuenta las interfaces de los otros, esto implica que los objetos accedan sólo a través de sus interfaces por lo que no se rompe con la encapsulación. Optar por la composición de objetos frente a la herencia de clases ayuda a tener cada clase encapsulada y centrada en una sola tarea. Esto nos lleva al segundo principio de *Favorecer la composición de objetos frente a la herencia de clases*. El inconveniente de la composición es que al tener más objetos (menos clases), el comportamiento del sistema dependerá de sus relaciones en vez de estar definido en una clase.

La *delegación* es un modo de lograr que la composición sea tan potente para la reutilización como lo es la herencia [JZ91]. La delegación significa que dos son los objetos encargados de tratar una petición, ya que un objeto receptor delega operaciones en su delegado. El inconveniente de la delegación al igual que otras técnicas que hacen que el software sea más flexible mediante la composición de objetos, es que el software dinámico y altamente parametrizado (tipos genéricos) es más difícil de entender que el estático. Su principal ventaja es que hace que sea más fácil combinar comportamientos en tiempo de ejecución y cambiar la manera en que éstos se combinan.

¿Por qué estudiar patrones de diseño?

Ahora que se tiene una idea de qué son los patrones de diseño, uno se podrá preguntar "¿Por qué estudiarlos?" Existen muchas razones obvias y algunas que no son tan obvias.

Las razones más comunes para el estudio de patrones son:

- *Soluciones Reutilizables*. Cuando se usan diseños ya establecidos, se evitan problemas de cabeza. Se obtiene el beneficio de aprender de la experiencia de otros y no se tiene que reinventar la solución a problemas comunes y recurrentes.
- *Establecer una terminología común*. La comunicación y trabajo en equipo requiere una base de vocabulario común y un punto de perspectiva común del problema. Los patrones de diseño proveen un punto de vista de referencia común durante la fase de análisis y diseño de un proyecto.

Sin embargo, hay una tercera razón para estudiar los patrones de diseño:

- El darnos una *perspectiva de alto nivel* en el problema y en el proceso de diseño y orientación de objetos. De liberarnos de la tiranía de tratar con los detalles demasiado temprano [ST06].

¿Cómo usar un patrón de diseño?

Una vez que se haya elegido un patrón de diseño, ¿cómo usarlo? Lo que sigue a continuación es un enfoque paso a paso [GHJ95] para aplicar un patrón de diseño de manera efectiva:

- Lea el patrón de principio a fin para tener una perspectiva. Preste particular atención a las secciones de Aplicabilidad y Consecuencias para asegurarse de que el patrón es el adecuado para su problema.
- Vuelva atrás y estudie los diagramas UML, y observe las responsabilidades y colaboraciones de los objetos y clases en el patrón de diseño. Asegúrese de ver cómo se relacionan entre ellos.
- Examine el código de ejemplo para ver un ejemplo concreto del patrón en código. Estudiar el código ayuda a entender cómo implementar el patrón.
- Elija nombres significativos en el contexto de la aplicación para los participantes en el patrón, así sabrá qué patrón utilizó en su implementación. Por ejemplo si utiliza el patrón Observer su clase se podría llamar ObserverComposicionSimple.
- Defina las clases. Declare sus interfaces, establezca sus relaciones de herencia y defina las variables de instancia que representan datos y referencias de objetos. Identifique las clases existentes en su aplicación a las que afectará el patrón y modifíquelas en consecuencia.
- Defina nombres específicos de la aplicación para las operaciones del patrón. Los nombres generalmente dependen de la aplicación. Use las responsabilidades y colaboraciones asociadas con cada operación como una guía. También debe ser coherente en sus convenciones de nombres. Por ejemplo, podría usar el prefijo “Crear“ de forma constante para denotar un método de fabricación.
- Implemente las operaciones para llevar a cabo las responsabilidades y colaboraciones del patrón.

Los patrones de diseño no deberían ser aplicados indiscriminadamente. Muchas veces éstos consiguen flexibilidad y variabilidad a costa de introducir niveles adicionales de indirección, y esto puede complicar un diseño o disminuir el rendimiento. Un patrón de diseño sólo debería ser aplicado cuando la flexibilidad que proporcione sea realmente necesaria, por lo tanto se debe tomar en cuenta la sección consecuencias presentada en cada uno de los patrones de diseño.

Capítulo 1: Lenguaje Unificado de Modelado

El lenguaje unificado de modelado es una notación utilizada para el análisis y diseño orientado a objetos. Este capítulo contiene un breve repaso general de dicho lenguaje que nos ayudará a comprender la notación y las extensiones UML utilizadas en los capítulos posteriores.

El UML define varios tipos de diagramas, por lo tanto, a lo largo de éste capítulo se revisarán los diagramas más utilizados, así como los elementos que se muestran en ellos.

1.1 ¿Qué es el UML?

El Lenguaje Unificado de Modelado o UML por sus siglas en inglés (*Unified Modeling Language*) es el lenguaje de modelado de sistemas más conocido y utilizado en el mundo actual, es el sucesor de la oleada de métodos de análisis y diseño orientados a objetos [FS99] que surgió a finales de la década de los 80's y principios de los 90's. El UML unifica los métodos de Grady Booch, James Rumbaugh, Ivar Jacobson, y otros.

UML permite a los desarrolladores de sistemas generar diseños mediante un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema, en otras palabras, permite al desarrollador capturar sus ideas de forma convencional y fácil de comprender para comunicarlas a otras personas.

UML es un lenguaje [Ala00] porque proporciona un vocabulario y las reglas para utilizarlo, además es un lenguaje de modelado lo que significa que el vocabulario y las reglas se utilizan para la representación conceptual y física del sistema. Puede usarse para modelar desde sistemas de información hasta aplicaciones distribuidas basadas en Web.

UML provee la capacidad de modelar actividades de planificación de proyectos y de sus versiones, expresar requisitos y las pruebas sobre el sistema, representar todos sus detalles así como la propia arquitectura. Mediante estas capacidades se obtiene una documentación que es válida durante todo el ciclo de vida de un proyecto.

1.2 Importancia del UML

En los comienzos del desarrollo de software, los programadores no realizaban análisis profundos sobre los problemas a resolver, si acaso, realizaban ciertos dibujos en servilletas que sólo ellos entendían.

En la actualidad, es imperativo contar con un plan bien analizado y por si fuera poco, el cliente necesita entender qué es lo que hará el equipo de desarrolladores, además de tener la capacidad de realizar cambios al diseño si no se han captado de forma correcta las necesidades. El desarrollo es un esfuerzo orientado a equipos, por lo que cada programador necesita entender y saber su rol de trabajo en la solución final.

Un arquitecto no podría crear una compleja estructura como lo es un puente vehicular sin crear primero un anteproyecto detallado, de la misma manera un arquitecto o desarrollador de sistemas requiere escribir los *planos del software* [Ala00] que son el resultado de un cuidadoso análisis de las necesidades del cliente.

Hoy en día, la creación de sistemas demanda una reducción al período de desarrollo, un diseño sólido que facilite cambios en la implementación, así como la posibilidad de modificar aspectos importantes de un proyecto de desarrollo que esté en proceso.

La necesidad de diseños sólidos ha traído consigo la creación de una notación de diseño que los analistas, desarrolladores y clientes acepten. Por ello la importancia de esta misma notación, a saber, el UML.

1.3 Terminología y conceptos UML

A lo largo de los años se ha aprendido de la sabiduría de frases como "di lo que piensas, entiende lo que dices" y "ve al grano". Probablemente hemos aprendido que la mejor comunicación con otras personas ocurre cuando se dice lo que se necesita decir, no más, no menos.

Los expertos suelen usar sus propias palabras para describir este principio común de comunicación [CS03]. Los siguientes conceptos interpretan lo que los expertos quieren decir realmente con respecto a las definiciones de frases utilizadas en el diseño y análisis orientado a objetos [Boo04] en el desarrollo de sistemas:

Objeto

Se refiere a algo útil que posee identidad, estado y comportamiento.

Ejemplo: El teléfono que se encuentra en la sala de mi hogar es único al lado de los demás teléfonos. Mide unos 20 centímetros, las teclas son suaves y es color blanco. Funciona correctamente porque me sirve para hablar con mis familiares lejanos.

Clase

Una familia de objetos con estado y comportamiento similar.

Ejemplo: Cuando uno se refiere a mi teléfono y a todos los miles de teléfonos. Todas estas unidades aunque de diferentes tamaños y colores forman una sola clase de teléfonos.

Abstracción

Describe la esencia de un objeto para un propósito. Centrarse en lo que es y lo que hace un objeto.

Ejemplo: El diagrama de circuitos del teléfono describe la esencia y el funcionamiento del teléfono, así que no hay que preocuparse de meter mano al cableado interno para poder comunicarnos.

Encapsulación

Dime lo que necesito saber para utilizar el objeto.

Ejemplo: "Levanta el teléfono y marca este número para comunicarte con tu papá". Este enunciado encapsula todo el detalle de cómo la electricidad fluye mediante los cables o todo el proceso que ocurre detrás al presionar un botón para que una persona pueda comunicarse.

Ocultamiento de Información

Mantenlo simple ocultando los detalles.

Ejemplo: Las personas no necesitan saber que el teléfono requiere una pequeña bocina y un micrófono interno para poder funcionar, simplemente usan el teléfono.

Agregación

Sólo dime acerca del objeto completo *o* háblame de las partes del objeto completo.

Ejemplo: Un objeto que representa el "todo" contiene o está formado por objetos de otras clases. Un camión es el objeto completo, pero está formado por objetos como el motor, puertas, llantas, etc.

Composición

Un objeto no puede existir si no existen los objetos de los que está compuesto.

Se trata de una forma de agregación fuerte. Los expertos necesitan dos palabras para distinguir diferentes situaciones como lo es: agregación y composición.

Ejemplo: Si a un teléfono se le corta el cable que conecta al auricular con la base, todo el teléfono deja de funcionar porque una de sus partes u objetos que los componen deja de funcionar. A diferencia de una relación de agregación, en la relación de composición el teléfono requiere de todas sus partes para seguir funcionando.

Generalización

Sólo dime qué tienen en común esos objetos.

Ejemplo: Todos los teléfonos necesitan de interfaz con números para marcar, así como un auricular por ejemplo.

Especialización

Tan sólo dime en qué es diferente este objeto en particular.

Ejemplo: El modelo de teléfono que tengo es diferente porque tiene una superficie de metal cubierta de plástico para mayor durabilidad.

Herencia

No olvides que los objetos especializados heredan las características comunes de objetos generales.

Ejemplo: Un objeto "león" hereda de un objeto "felino", y un "tigre" hereda también de un objeto "felino". Se puede decir que los objetos león y tigre son especializados, y heredan de uno general que es el objeto felino.

Polimorfismo

Los objetos tienen el mismo comportamiento pero se realizan de forma diferente.

Ejemplo: Al encender cualquier radio sólo presionamos el botón de encendido, pero el mecanismo interno o método de encendido que está detrás del botón es diferente para cada radio.

1.4 Diagramas del UML

Un diagrama es la representación gráfica de un conjunto de elementos con sus relaciones. El UML se compone por diversos elementos gráficos que se combinan para conformar diagramas. Debido a que el UML es un lenguaje, cuenta con reglas para combinar tales elementos.

La finalidad de los diagramas es presentar las diversas perspectivas del sistema, a las cuales se les conoce como *modelo*. El modelo UML describe lo que supuestamente hará un sistema, pero no dice como implementar dicho sistema.

Para poder representar un sistema UML ofrece una amplia variedad de diagramas para visualizar el sistema desde varias perspectivas. Es importante recordar que es posible generar híbridos de estos diagramas y que dicho lenguaje otorga formas de organizarlos y extenderlos.

En la figura B-1 se muestra un breve resumen de la notación UML:

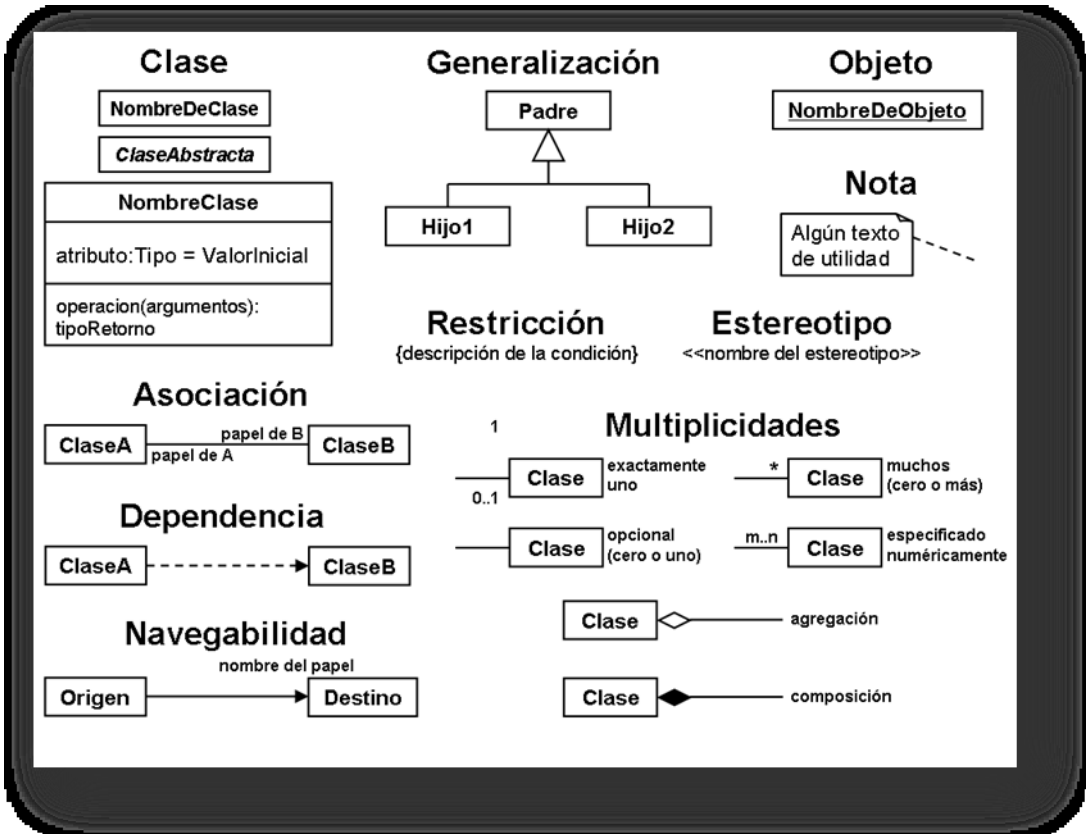

Figura B-1: Notación UML [FS99]

Jerarquía de los diagramas UML

En UML 2.0 se encuentran 13 tipos diferentes de diagramas [BRJ05]. Para comprenderlos de manera concreta, se categorizan jerárquicamente, como se muestra en la figura B-2

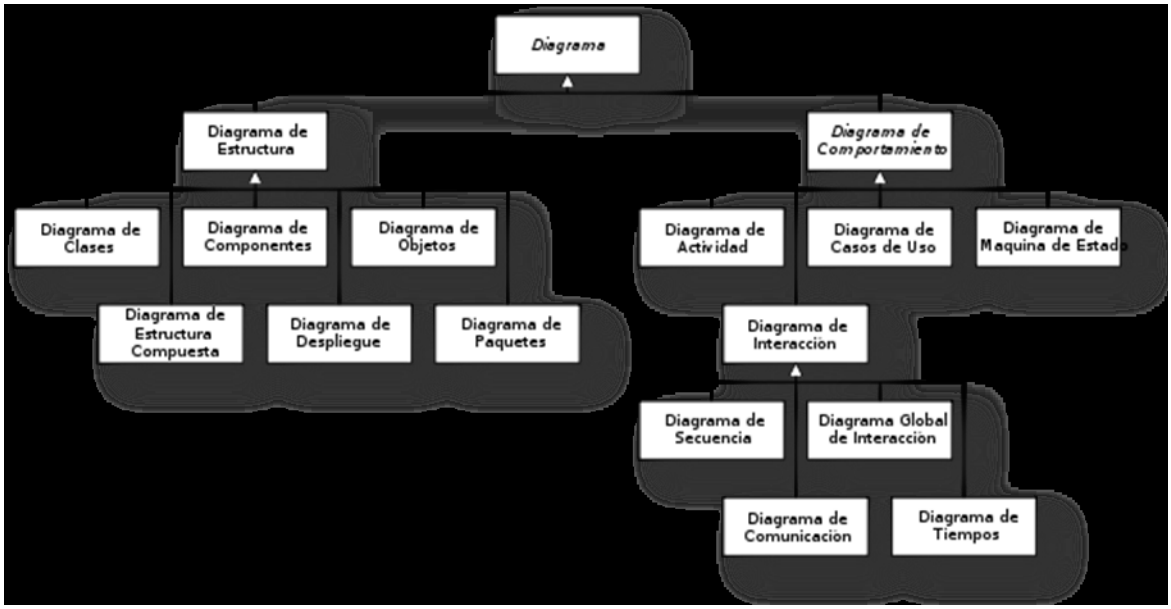

Figura B-2: Jerarquía de los diagramas UML 2.0

Los ***Diagramas de Estructura*** destacan los elementos que deben *existir* en el sistema modelado:

- Diagrama de clases
- Diagrama de componentes
- Diagrama de objetos
- Diagrama de estructura compuesta
- Diagrama de despliegue
- Diagrama de paquetes

Los ***Diagramas de Comportamiento*** destacan lo que debe *suceder* en el sistema modelado:

- Diagrama de actividades
- Diagrama de casos de uso
- Diagrama de estados

Los ***Diagramas de Interacción*** son un subtipo de diagramas de comportamiento, que enfatizan el *flujo de control y de datos* entre los elementos del sistema modelado:

- Diagrama de secuencia
- Diagrama de colaboración
- Diagrama de tiempos
- Diagrama global de interacciones o Diagrama de vista de interacción

A continuación se describirán brevemente los principales diagramas del UML que servirán para entender el modelo de los diferentes patrones de diseño, así como los conceptos que representan.

Diagrama de Clases

El diagrama de clases describe la estructura de un sistema mostrando sus clases, interfaces, atributos y las relaciones entre ellos [BRJ05]. Los diagramas de clases son utilizados durante el proceso de análisis y diseño de los sistemas, donde se crea el diseño conceptual de la información que se manejará en el sistema, y los componentes que se encargarán del funcionamiento y la relación entre uno y otro. Los diagramas de clase son los diagramas que se encuentran comúnmente en el modelado de sistemas orientados a objetos. En la figura B-3 se proporciona un ejemplo de diagrama de clase.

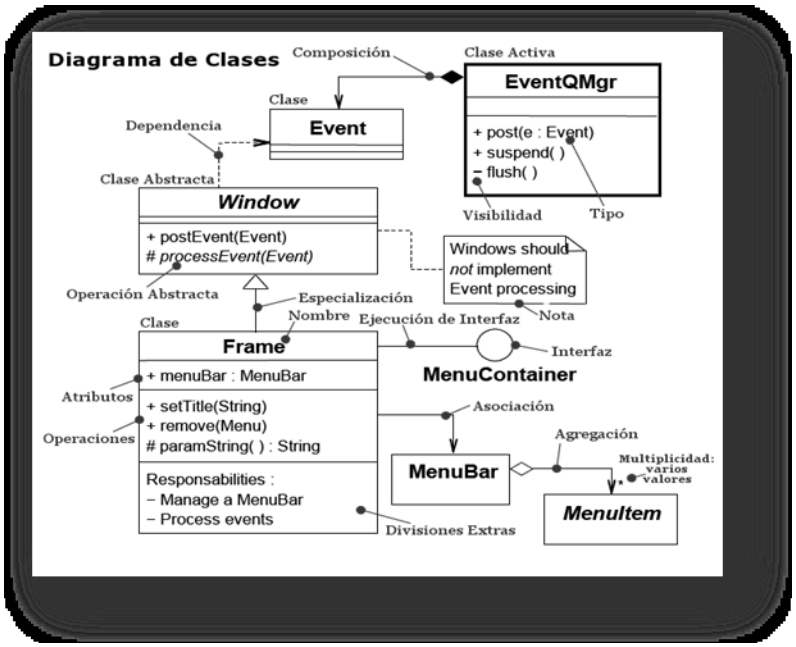

Figura B-3: Diagrama de clases [Lau03]

Diagrama de Objetos

Un objeto es una instancia de clase [Sch98]. El diagrama de objetos muestra un conjunto de objetos y sus relaciones. Se usan los diagramas de objetos para ilustrar estructuras de datos. Los diagramas de objetos utilizan un subconjunto de los elementos de un diagrama de clase. Los diagramas de objetos no muestran la multiplicidad ni los roles, aunque su notación es similar a los diagramas de clase. La figura B-4 ejemplifica la forma en que el UML muestra los objetos. El símbolo es un rectángulo, como en una clase, pero el nombre está subrayado. El nombre de la instancia se encuentra a la izquierda de los dos puntos (:), y el nombre de la clase a la derecha.

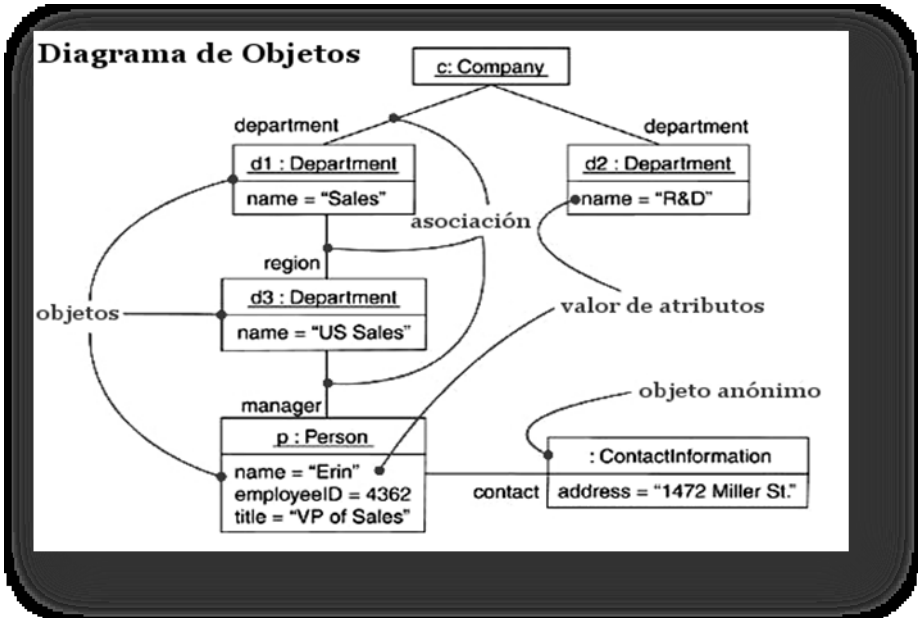

Figura B-4: Diagrama de objetos [BRJ05]

Diagrama de Casos de Uso

Un caso de uso es una descripción de las acciones de un sistema desde el punto de vista del usuario. Para los desarrolladores del sistema, ésta es una técnica de aciertos y errores para obtener los requerimientos del sistema desde el punto de vista del usuario.

En la figura B-5 se describe la funcionalidad de un Sistema Restaurante muy simple. Los casos de uso están representados por elipses y los actores están representados por las figuras humanas. El actor *Cliente* de comidas puede *Probar la comida, Pagar la comida, o Beber vino*. Sólo el actor *Cocinera* puede *Preparar la comida*. El marco define los límites del sistema Restaurante, por ejemplo, los casos de uso se muestran como parte del sistema que está siendo modelado, los actores no.

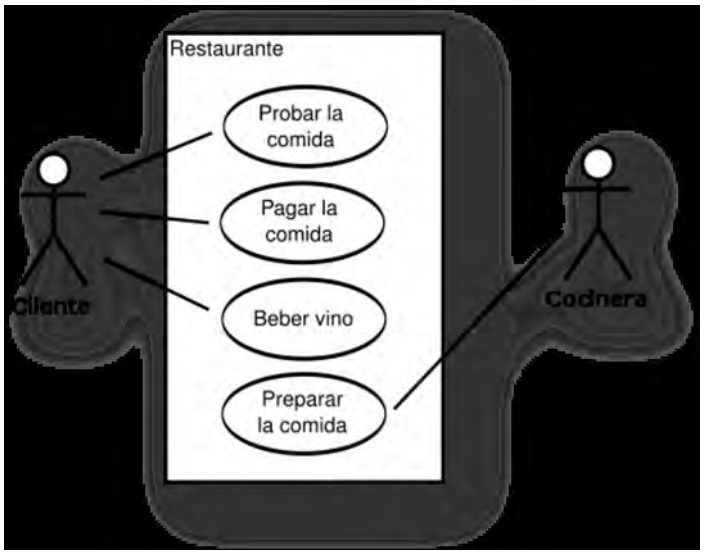

Figura B-5: Diagrama de casos de uso

La figura correspondiente al cliente o cocinera se le conoce como actor y la elipse representa el caso de uso.

Existen varias relaciones entre los casos de uso [FS99] que describen una notación gráfica, las principales relaciones son:

Inclusión (include o use). Las relaciones *uses* ocurren cuando se tiene una porción de comportamiento que es similar en más de un caso de uso y no se quiere copiar la descripción de tal conducta. Para esta relación se utiliza la etiqueta «uses».

Extensión (extend). Se usa la relación *extends* cuando se tiene un caso de uso que es similar a otro, pero que hace un poco más, la etiqueta usada es «extends».

Se recomienda utilizar *extends* cuando se describe una variación de conducta normal. Se emplea *uses* para repetir cuando se trate de uno o varios casos de uso y desee evitar repeticiones.

Diagrama de Estados

El diagrama de estado muestra una máquina de estado [BRJ05], que consiste en estados, transiciones, eventos, y actividades. Se usa el diagrama de estado para ilustrar la vista dinámica de un sistema. Son especialmente importantes para modelar el comportamiento de una interfaz, clase o colaboración. Los diagramas de estado enfatizan el comportamiento ordenado de eventos de un objeto, siendo especialmente útiles en el modelado de sistemas reactivos.

La figura B-6 muestra un ejemplo de diagrama de estados.

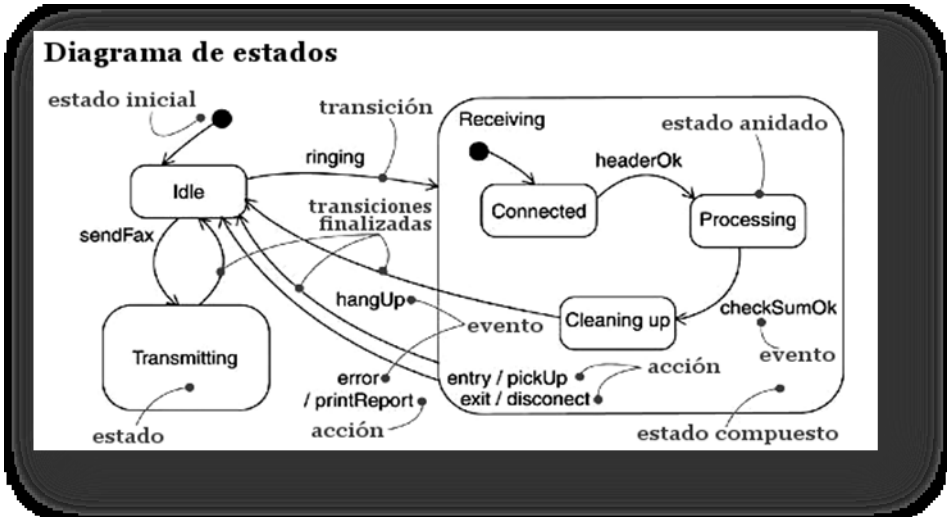

Figura B-6: Diagrama de estados [BRJ05]

Diagrama de Secuencia

Los diagramas de clases y los de objeto representan información estática. No obstante, en un sistema funcional los objetos interactúan entre sí, y tales interacciones suceden con el tiempo. Un diagrama de secuencia es un diagrama de interacción [Sch98] que hace destacar el orden de tiempo de mensajes. Muestra un conjunto de roles y de mensajes enviados y recibidos por las instancias ejemplificadas para cada rol. También se utiliza el diagrama de secuencia para representar la vista dinámica de un sistema. En la figura B-7 se muestra un ejemplo de un diagrama de secuencia.

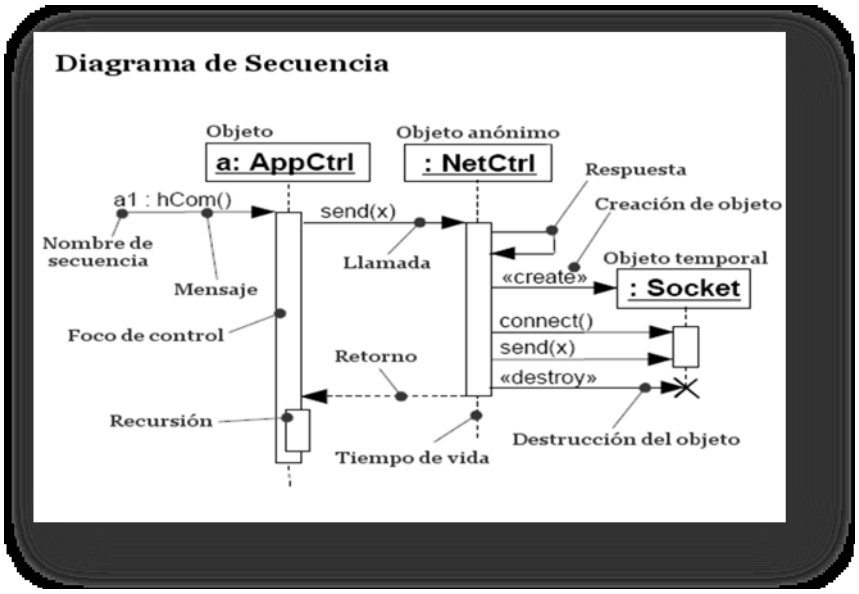

Figura B-7: Diagrama de secuencia [Lau03]

Diagrama de Colaboración

Los elementos de un sistema trabajan en conjunto para cumplir con los objetivos del sistema. Los diagramas de colaboración muestran la forma en que los objetos colaboran entre sí, tal como sucede con un diagrama de secuencia [Sch98]. Muestran los objetos junto con los mensajes que se envían entre ellos. A diferencia de los diagramas de secuencia que destacan la sucesión de las interacciones, los diagramas de colaboración destacan el contexto y organización general de los objetos que interactúan, es decir, el diagrama de secuencias se organiza de acuerdo al tiempo y el de colaboración de acuerdo al espacio. En la figura B-8 se muestra un ejemplo del diagrama de secuencia y de colaboración para entender ésta diferencia.

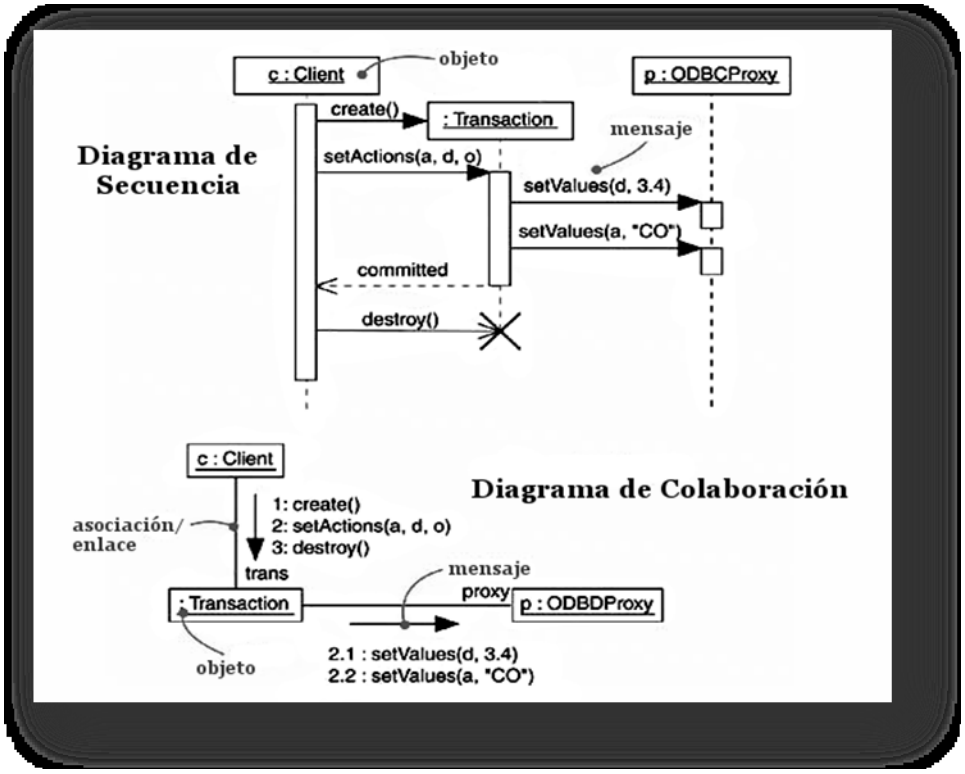

Figura B-8: Diagramas de Interacción: diagrama de secuencia y de colaboración [BRJ05]

Diagrama de Actividades

Los diagramas de actividades son un caso especial de diagramas de estado. Muestran el flujo paso a paso dentro del cómputo. La actividad muestra el conjunto de acciones, el flujo secuencial o de ramificaciones de acción a acción, y los valores que se producen o que son consumidas por las acciones [BRJ05]. Los diagramas de actividad se centran en el flujo de control desde adentro de la ejecución de un comportamiento. La figura B-9 muestra un ejemplo de diagrama de actividades.

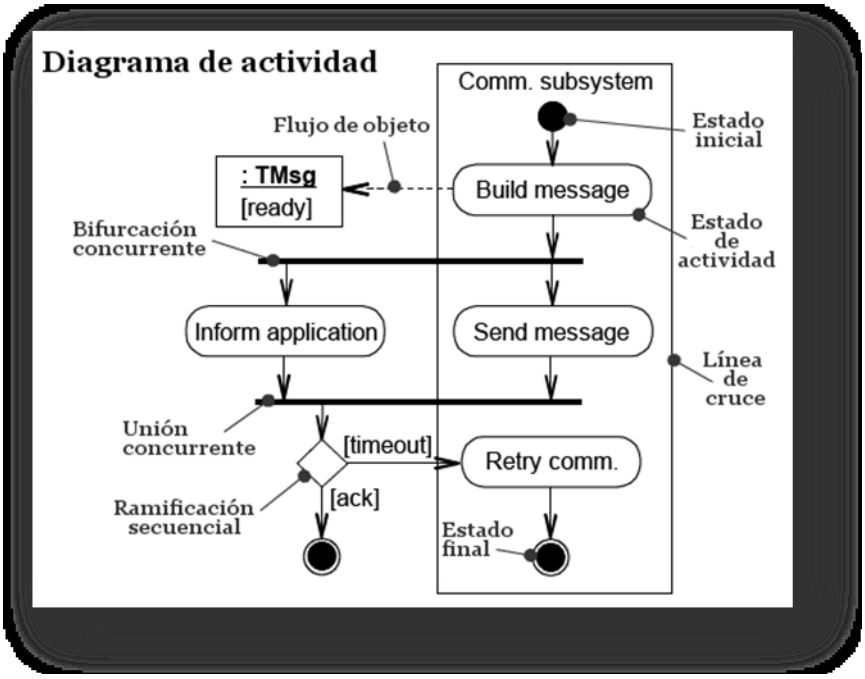

Figura B-9: Diagrama de actividades [Lau03]

Capítulo 2: Patrones de Creación

2.1 Abstract Factory (Fábrica Abstracta)

También conocido como: Kit, Toolkit.

Descripción

En términos simples, una fábrica abstracta es una clase que proporciona una interfaz para producir una familia de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas. En el lenguaje de programación Java, puede aplicarse como una interfaz, o como una clase abstracta.

En el contexto de una fábrica abstracta, existen:

- Series o familias de relación de clases dependientes.
- Un grupo de clases de fábrica concreta que implementa la interfaz proporcionada por la clase de fábrica abstracta. Cada una de estas fábricas controla o proporciona acceso a una familia particular de relación de objetos dependientes e implementa la interfaz de fábrica abstracta en una forma específica a la familia de las clases que controla.

El patrón Abstract Factory es útil cuando un objeto cliente quiere crear una instancia de una serie de relación, y las clases dependientes no tienen que saber específicamente qué clase concreta está siendo instanciada. En ausencia de una fábrica abstracta, la implementación requerida para seleccionar una clase apropiada (en otras palabras, el criterio de selección de clase) necesita estar presente en todas partes al tiempo que una instancia es creada. Una fábrica abstracta ayuda a evitar esa duplicación proporcionando la interfaz necesaria para la creación de esas instancias. Diferentes fábricas concretas implementan esta interfaz. Objetos cliente hacen uso de estas fábricas concretas para crear objetos y, por tanto, no es necesario conocer qué clase concreta está siendo instanciada.

En Java, una fábrica abstracta también puede ser diseñada como una clase abstracta con sus subclases concretas como las fábricas, donde cada fábrica es responsable de crear y proporcionar acceso a los objetos de una familia particular de clases.

Aplicabilidad

Úsese el patrón Abstract Factory cuando:

- Un sistema debe ser independiente de cómo se crean, componen y representan sus productos.
- Un sistema debe ser configurado con una familia de productos de entre varias.
- Una familia de objetos producto relacionados está diseñada para hacer usada conjuntamente, y es necesario hacer cumplir esta restricción.
- Quiere proporcionar una biblioteca de clases de productos y sólo quiere revelar sus interfaces, no sus implementaciones.

Diagrama UML

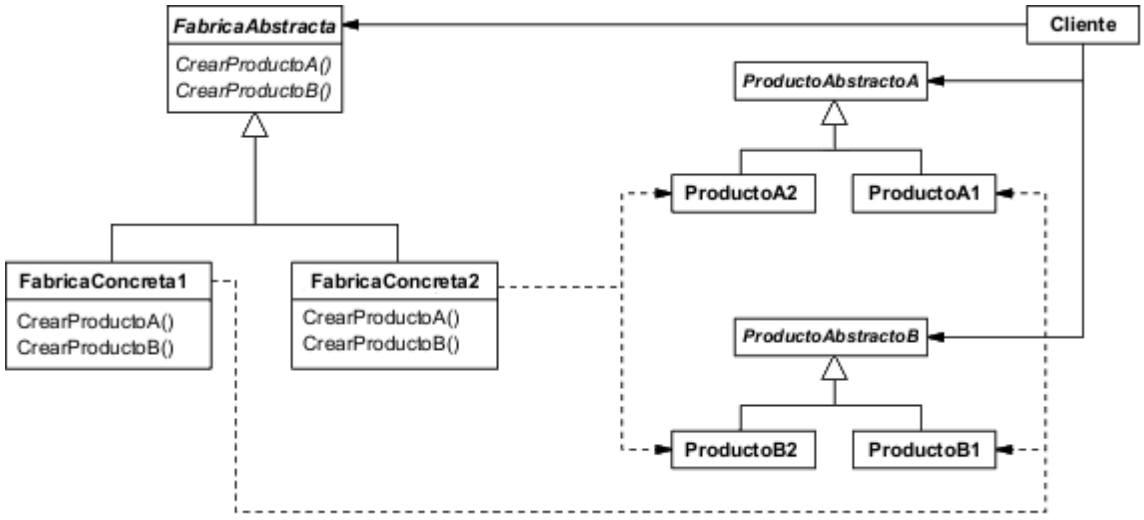

Figura B-10: Diagrama de clase para el patrón Abstract Factory

Participantes

FabricaAbstracta. Declara una interfaz para operaciones que crean objetos producto a partir de objetos abstractos.

FabricaConcreta. Implementa las operaciones para crear objetos producto concretos.

ProductoAbstracto. Declara una interfaz para un tipo de objeto producto.

ProductoConcreto. Define un producto para que sea creado por la fábrica correspondiente. Implementa la interfaz ProductoAbstracto.

Cliente. Sólo usa interfaces declaradas por las clases FabricaAbstracta y ProductoAbstracto.

Colaboraciones

Normalmente sólo se crea una única instancia de una clase FabricaConcreta en tiempo de ejecución. Esta fábrica concreta crea objetos producto que tienen una determinada implementación. Para crear diferentes objetos producto, los clientes deben usar una fábrica concreta diferente.

FabricaAbstracta delega la creación de objetos producto en su clase FabricaConcreta.

Consecuencias

El patrón Abstract Factory tiene las siguientes ventajas e inconvenientes:

Aísla las clases concretas. El patrón Abstract Factory ayuda a controlar las clases de objetos que crea una aplicación. Como una fábrica encapsula la responsabilidad y el proceso de creación de objetos producto, aísla a los clientes de las clases de implementación. Los clientes manipulan las instancias a través de sus interfaces abstractas. Los nombres de las clases producto quedan aislados en la implementación de la fábrica concreta; no aparecen en el código cliente.

Facilita el intercambio de familias de productos. La clase de una fábrica concreta sólo aparece una vez en una aplicación (cuando se crea). Esto facilita cambiar la fábrica concreta que usa una aplicación. Como una fábrica abstracta crea una familia completa de productos, toda la familia de productos cambia de una vez.

Promueve la consistencia entre productos. Cuando se diseñan objetos producto en una familia para trabajar juntos, es importante que una aplicación use objetos de una sola familia a la vez. FabricaAbstracta facilita que se cumpla esta restricción.

Es difícil dar cabida a nuevos tipos de productos. Ampliar las fábricas abstractas par producir nuevos tipos de productos no es fácil. Esto se debe a que la interfaz FabricaAbstracta fija el conjunto de productos que se pueden crear. Permitir nuevos tipos de productos requiere ampliar la interfaz de la fábrica, lo que a su vez implica cambiar la clase FabricaAbstracta y todas sus subclases.

Patrones Relacionados

Las clases Fábrica Abstracta suelen implementarse con métodos de fabricación (patrón Factory Method), pero también se pueden implementar usando prototipos (patrón Prototype).

Una fábrica concreta suele ser un Singleton.

Ejemplo A.1 Código de ejemplo para el patrón Abstract Factory

FabricaAbstracta.java

```
public interface FabricaAbstracta {
     public ProductoAbstractoA crearProductoA();
     public ProductoAbstractoB crearProductoB();
}
```

FabricaConcreta1.java

```
public class FabricaConcreta1 implements FabricaAbstracta{

     @Override
     public ProductoAbstractoA crearProductoA() {
          return new ProductoA1();
     }

     @Override
     public ProductoAbstractoB crearProductoB() {
          return new ProductoB1();
     }

}
```

FabricaConcreta2.java

```
public class FabricaConcreta2 implements FabricaAbstracta{

     @Override
     public ProductoAbstractoA crearProductoA() {
          return new ProductoA2();
     }

     @Override
     public ProductoAbstractoB crearProductoB() {
          return new ProductoB2();
     }

}
```

ProductoAbstractoA.java

```
public abstract class ProductoAbstractoA {
     private String descripcion = "ProductoAbstractoA";

     public String getDescripcion() {
          return descripcion;
     }

     public void setDescripcion(String descripcion) {
          this.descripcion = descripcion;
     }

     public abstract String getMensaje();

}
```

ProductoAbstractoB.java

```
public abstract class ProductoAbstractoB {
     private String descripcion = "ProductoAbstractoB";

     public String getDescripcion() {
          return descripcion;
     }

     public void setDescripcion(String descripcion) {
```

```
        this.descripcion = descripcion;
    }

    public abstract String getMensaje();

}
```

ProductoA1.java

```
public class ProductoA1 extends ProductoAbstractoA{

    @Override
    public String getMensaje() {
        return "Soy un productoConcretoA1 que extiende de "+this.getDescripcion();
    }

}
```

ProductoA2.java

```
public class ProductoA2 extends ProductoAbstractoA{

    @Override
    public String getMensaje() {
        return "Soy un productoConcretoA2 que extiende de "+this.getDescripcion();
    }

}
```

ProductoB1.java

```
public class ProductoB1 extends ProductoAbstractoB{

    @Override
    public String getMensaje() {
        return "Soy un productoConcretoB1 que extiende de "+this.getDescripcion();
    }

}
```

ProductoB2.java

```
public class ProductoB2 extends ProductoAbstractoB{

    @Override
    public String getMensaje() {
        return "Soy un productoConcretoB2 que extiende de "+this.getDescripcion();
    }

}
```

Cliente.java

```
public class Cliente {

    public static void main(String[] args) {

        FabricaAbstracta fabrica1 = new FabricaConcreta1();
        ProductoAbstractoA productoA1 = fabrica1.crearProductoA();
        ProductoAbstractoB productoB1 = fabrica1.crearProductoB();

        FabricaAbstracta fabrica2 = new FabricaConcreta2();
        ProductoAbstractoA productoA2 = fabrica2.crearProductoA();
        ProductoAbstractoB productoB2 = fabrica2.crearProductoB();

        System.out.println(productoA1.getMensaje());
        System.out.println(productoB1.getMensaje());
        System.out.println(productoA2.getMensaje());
        System.out.println(productoB2.getMensaje());
```

```
    }
}
```

Salida en Consola

```
Soy un productoConcretoA1 que extiende de ProductoAbstractoA
Soy un productoConcretoB1 que extiende de ProductoAbstractoB
Soy un productoConcretoA2 que extiende de ProductoAbstractoA
Soy un productoConcretoB2 que extiende de ProductoAbstractoB
```

2.2 Builder (Constructor)

Descripción

Como Patrón de diseño, el patrón builder es usado para permitir la creación de una variedad de objetos complejos desde un objeto fuente (Producto), el objeto fuente se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo a través de un conjunto de llamadas a interfaces comunes de la clase Abstract Builder.

A menudo, el patrón builder construye el patrón Composite, un patrón estructural. Abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.

Aplicabilidad

Úsese el patrón Builder cuando:

- El algoritmo para crear un objeto complejo debiera ser independiente de las partes de que se compone dicho objeto y de cómo se ensamblan.
- El proceso de construcción debe permitir diferentes representaciones del objeto que está siendo construido.

Diagrama UML

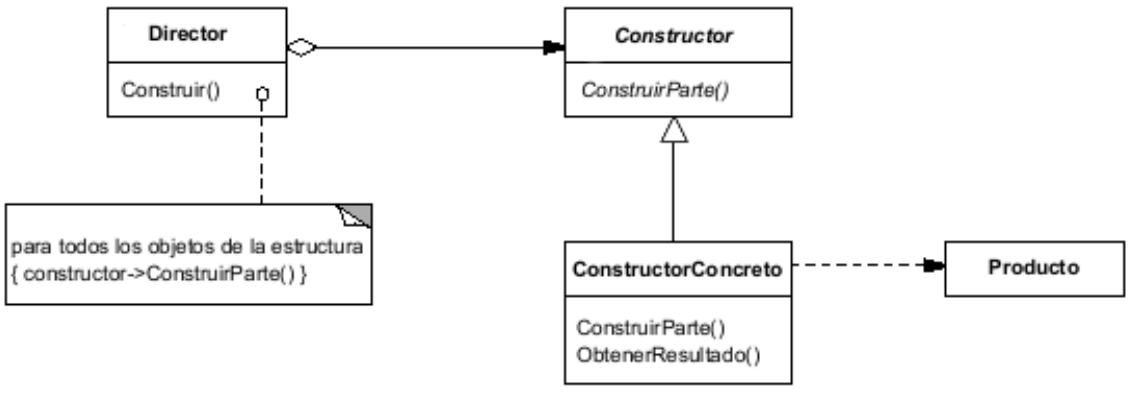

Figura B-11: Diagrama de clase para el patrón Builder

Participantes

Constructor. Especifica una interfaz abstracta para crear las partes de un objeto Producto.

ConstructorConcreto. Implementa la interfaz Constructor para construir y ensamblar las partes del producto. Define la representación a crear. Proporciona una interfaz para devolver el producto.

Colaboraciones

El cliente crea el objeto Director y lo configura con el objeto Constructor deseado.

El Director notifica al constructor cada vez que hay que construir una parte de un producto.

El Constructor maneja las peticiones del director y las añade al producto.

El cliente obtiene el producto del constructor.

El siguiente diagrama de secuencia (figura B-12) ilustra cómo cooperan con un cliente el Constructor y el Director.

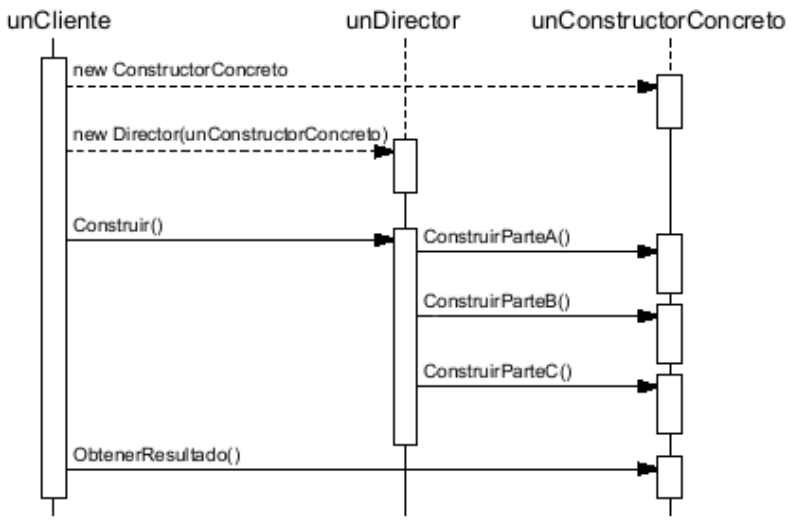

Figura B-12: Diagrama de secuencia para el patrón Builder

Consecuencias

Las principales consecuencias del patrón Builder son:

Permite variar la representación interna de un producto. El objeto Constructor proporciona al director una interfaz abstracta para construir el producto. La interfaz permite que el constructor oculte la representación y la estructura interna del producto. También oculta el modo en que éste es ensamblado. Dado que el producto se construye a través de una interfaz abstracta, todo lo que hay que hacer para cambiar la representación interna del producto es definir un nuevo tipo constructor.

Aísla el código de construcción y representación. El patrón Builder aumenta la modularidad al encapsular cómo se construyen y se representan los objetos complejos. Los clientes no necesitan saber nada de las clases que definen la estructura interna del producto; dichas clases no aparecen en la interfaz del Constructor. Cada ConstructorConcreto contiene todo el código para crear y ensamblar un determinado tipo de producto. El código sólo se escribe una vez; después, los diferentes Directores pueden reutilizarlo para construir variantes del Producto a partir del mismo conjunto de partes.

Proporciona un control más fino sobre el proceso de construcción. A diferencia de los patrones de creación que construyen los productos de una vez, el patrón Builder construye el producto paso a paso, bajo el control del director. El director sólo obtiene el producto del constructor una vez que ése está terminado. Por lo tanto, la interfaz Constructor refleja el proceso de construcción del producto más que otros patrones de creación. Esto da un control más fino sobre el proceso de construcción y, por lo tanto, sobre la estructura interna del producto resultante.

Patrones Relacionados

El patrón Abstract Factory se parece a un Builder en que también puede construir objetos complejos. La principal diferencia es que el patrón Builder se centra en construir un objeto complejo paso a paso. El Abstract Factory hace hincapié en familias de objetos producto (simples o complejos). El Builder devuelve el producto como paso final, mientras el Abstract Factory lo devuelve inmediatamente.

Muchas veces lo que construye el constructor es un Composite.

Ejemplo A.2 Código de ejemplo para el patrón Builder

Constructor.java

```
public abstract class Constructor {
     public abstract void construirParteA();
     public abstract void construirParteB();
     public abstract void construirParteC();
}
```

ConstructorConcreto.java

```
public class ConstructorConcreto extends Constructor{
     private Producto producto = new Producto();

     @Override
     public void construirParteA() {
          producto.AgregarParte("parte A");
     }
     @Override
     public void construirParteB() {
          producto.AgregarParte("parte B");
     }
     @Override
     public void construirParteC() {
          producto.AgregarParte("parte C");
     }

     public Producto obtenerResultado() {
          return producto;
     }

}
```

Director.java

```
public class Director {
     private Constructor constructor;

     public Constructor getConstructor() {
          return constructor;
     }

     public void setConstructor(Constructor constructor) {
          this.constructor = constructor;
     }

     public void construir(){
          constructor.construirParteA();
          constructor.construirParteB();
          constructor.construirParteC();
     }

}
```

Producto.java

```
import java.util.ArrayList;

public class Producto {
     ArrayList <String> partes = new ArrayList <String>();

    public void AgregarParte(String parte){
      partes.add(parte);
    }

    public void MostrarPartes(){
      for(String parte : partes)
        System.out.println("Se construye la " + parte);
```

```
    }

}
```

Cliente.java

```
public class Cliente {
     public static void main(String[] args) {
          Constructor constructorConcreto = new ConstructorConcreto();
          Director director = new Director();
          director.setConstructor(constructorConcreto);
          director.construir();
          Producto producto = ((ConstructorConcreto)constructorConcreto).obtenerResultado();
          producto.MostrarPartes();
     }
}
```

Salida en Consola

```
Se construye la parte A
Se construye la parte B
Se construye la parte C
```

2.3 Factory Method (Método de Fabricación)

También conocido como: Virtual Constructor (Constructor Virtual)

Descripción

Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. En general, todas las subclases en una jerarquía de clases heredan los métodos implementados por la clase padre. Una subclase puede sobrescribir la implementación de la clase padre para ofrecer otro tipo de funcionalidad para el mismo método. Cuando un objeto de aplicación es consciente de la funcionalidad exacta que necesita, puede instanciar directamente la jerarquía de clase que ofrece la funcionalidad requerida.

Aplicabilidad

Úsese el patrón Factory Method cuando:

- Una clase no puede prever la clase de objetos que debe crear.
- Una clase quiere que sean sus subclases quienes especifiquen los objetos que ésta crea.
- Las clases delegan la responsabilidad en una de entre varias clases auxiliares, y se requiere conocer en qué subclase de auxiliar concreta se delega.

Diagrama UML

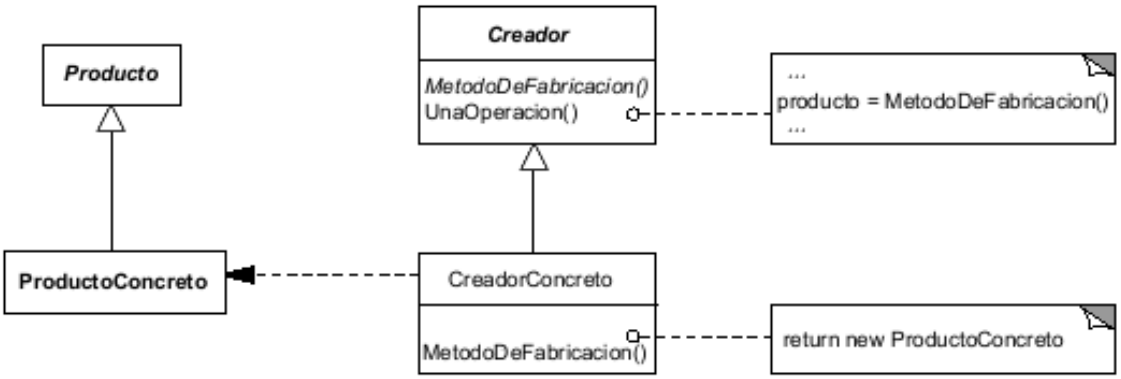

Figura B-13: Diagrama de clase para el patrón Factory Method

Participantes

Producto. Define la interfaz de los objetos que crea el método de fabricación.

ProductoConcreto. Implementa la interfaz Producto.

Creador. Declara el método de fabricación, el cual devuelve un objeto de tipo Producto. También puede definir una implementación predeterminada del método de fabricación que devuelva un objeto ProductoConreto. Puede llamar al método de fabricación para crear un objeto Producto.

CreadorConcreto. Redefine el método de fabricación para devolver una instancia de un ProductoConcreto.

Colaboraciones

El Creador se apoya en sus subclases para definir el método de fabricación de manera que éste devuelva una instancia del ProductoConcreto apropiado.

Consecuencias

Los métodos de fabricación eliminan la necesidad de ligar clases específicas de la aplicación a nuestro código. El código sólo trata con la interfaz Producto; además, puede funcionar con cualquier clase ProductoConcreto definida por el usuario.

Un inconveniente potencial de los métodos de fabricación es que los clientes pueden tener que heredar de la clase Creador simplemente para crear un determinado objeto ProductoConcreto. La herencia está bien definida cuando el cliente tiene que heredar de todos modos de la clase Creador, pero si no es así se estaría introduciendo una nueva vía de futuros cambios.

Dos consecuencias más del patrón Factory Method son:

Proporciona enganches para las subclases. Crear objetos dentro de una clase con un método de fabricación es siempre más flexible que hacerlo directamente. El Factory Method les da a las subclases un punto de enganche para proveer una versión extendida de un objeto.

Conecta jerarquías de clases paralelas. Las jerarquías paralelas se producen cuando una clase delega alguna de sus responsabilidades a una clase separada. Pensemos en figuras gráficas que pueden manipularse interactivamente; es decir, que pueden alargarse, moverse y girarse usando el ratón. Implementar estas interacciones no siempre es fácil. Muchas veces requiere almacenar y actualizar información que guarda el estado de la manipulación en un momento dado. Este estado es necesario sólo mientras dura la manipulación; por tanto, no necesita ser almacenado en el objeto figura. Con estas restricciones es mejor usar un objeto Manipulador separado que implemente la interacción y mantenga cualquier estado específico de la manipulación que sea necesario. Figuras diferentes usarían distintas subclases de Manipulador para manejar interacciones.

Patrones Relacionados

El patrón Abstract Factory suele implementarse con métodos de fabricación. Los métodos de fabricación generalmente son llamados desde el interior de Métodos Plantilla.

El Prototype no necesita heredar de Creador. Sin embargo, suelen requerir una operación Inicializar de la clase Producto. El Creador usa Inicializar para inicializar el objeto. El patrón Factory Method no requiere dicha operación.

Ejemplo A.3 Código de ejemplo para el patrón Factory Method

Creador.java

```
public abstract class Creador {
     public Producto unaOperacion(){
          return metodoDeFabricacion();
     }

     protected abstract Producto metodoDeFabricacion();
}
```

CreadorConcreto.java

```
public class CreadorConcreto extends Creador{

     @Override
     protected Producto metodoDeFabricacion() {
          return new ProductoConcreto();
     }

}
```

Producto.java

```
public interface Producto {
     public String operacionProducto();
}
```

ProductoConcreto.java

```
public class ProductoConcreto implements Producto{

     @Override
     public String operacionProducto() {
          return "Esta es una operacion de un producto concreto";
     }

}
```

Cliente.java

```
public class Cliente {
     public static void main(String[] args) {
          Creador creador = new CreadorConcreto();
          Producto producto = creador.unaOperacion();
          System.out.println("Realizo operacion: "+producto.operacionProducto());
     }
}
```

Salida en Consola

```
Realizo operacion: Esta es una operacion de un producto concreto
```

2.4 Prototype (Prototipo)

Descripción

Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando dicho prototipo. Tiene como finalidad crear nuevos objetos duplicándolos, clonando una instancia creada previamente.

Aplicabilidad

Se usa el patrón Prototype cuando un sistema deba ser independiente de cómo se crean, se componen y se representan sus productos; y:

Cuando las clases a instanciar sean especificadas en tiempo de ejecución (por ejemplo, mediante carga dinámica); *o*

Para evitar construir una jerarquía de clases de fábricas paralela a la jerarquía de clases de los productos, *o*

Cuando las instancias de una clase puedan tener uno de entre sólo unos pocos estados diferentes. Puede ser más adecuado tener un número equivalente de prototipos y clonarlos, en vez de crear manualmente instancias de la clase cada vez con el estado apropiado.

Diagrama UML

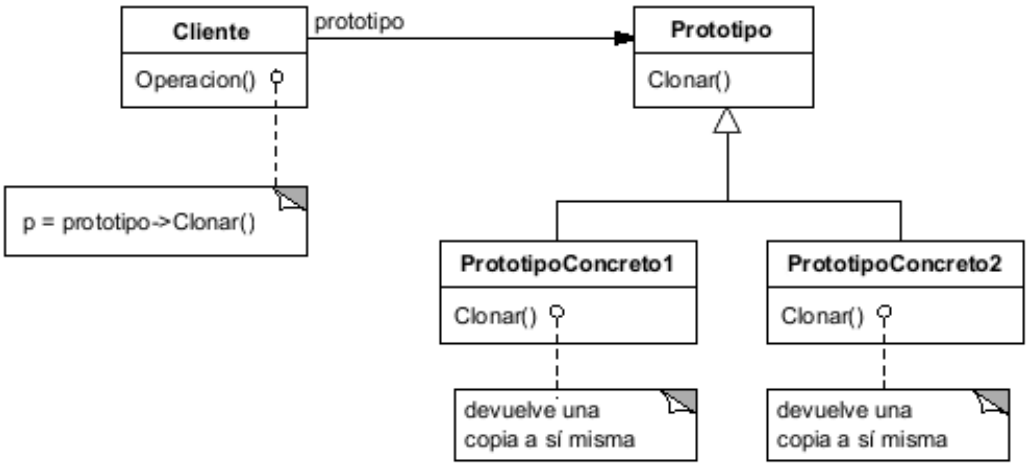

Figura B-14: Diagrama de clase para el patrón Prototype

Participantes

Prototipo. Declara una interfaz para clonarse.

PrototipoConcreto. Implementa una operación para clonarse.

Cliente. Crea un nuevo objeto pidiéndole a un prototipo que se clone.

Colaboraciones

Un cliente le pide a un prototipo que se clone.

Consecuencias

Muchas de la consecuencias del Prototype son las mismas que las de los patrones Abstract Factory y Builder, ya que oculta al cliente las clases producto concretas, reduciendo así el número de nombres que conocen los clientes. Además, estos patrones permiten que un cliente use clases específicas de la aplicación sin cambios.

A continuación se enumeran algunos beneficios adicionales del patrón Prototype:

Añadir y eliminar productos en tiempo de ejecución. Permite incorporar a un sistema una nueva clase concreta de producto simplemente registrando una instancia prototípica con el cliente. Esto es algo más flexible que otros patrones de creación, ya que un cliente puede instalar y eliminar prototipos en tiempo de ejecución.

Especificar nuevos objetos modificando valores. Los sistemas altamente dinámicos permiten definir comportamiento nuevo mediante la composición de objetos. Se pueden definir nuevos tipos de objetos creando instancias de clases existentes y registrando esas instancias como prototipos de los objetos cliente. Un cliente puede exhibir comportamiento nuevo delegando responsabilidad en su prototipo. Este tipo de diseño permite que los usuarios definan nuevas "clases" sin programación. De hecho, clonar un prototipo es parecido a crear una instancia de una clase. El patrón Prototype puede reducir en gran medida el número de clases necesarias en el sistema.

Especificar nuevos objetos variando la estructura. Muchas aplicaciones construyen objetos a partir e partes y subpartes. Los editores de diseño de circuitos, por ejemplo, construyen circuitos a partir de subcircuitos. A menudo, por comodidad, dichas aplicaciones permiten crear instancias de estructuras complejas definidas por el usuario, por ejemplo, para usar un determinado circuito una y otra vez. El patrón Prototype también permite eso. Simplemente añadimos ese subcircuito como prototipo a la paleta de circuitos disponibles. Siempre y cuando el objeto circuito compuesto

implemente Clonar como una copia profunda, los circuitos con varias estructuras también pueden ser prototipos.

Reduce la herencia. El patrón Factory Method suele producir una jerarquía de clases Creador que es paralela a la jerarquía de clases de productos. El patrón Prototype permite clonar un prototipo en vez de decirle a un método de fabricación que cree un nuevo objeto. Por lo tanto, no es en absoluto necesaria una jerarquía de clases Creador.

Configurar dinámicamente una aplicación con clases. Algunos entornos de tiempo de ejecución permiten cargar clases en una aplicación dinámicamente. El patrón Prototype es la clase para explotar dichas facilidades. Una aplicación que quiere crear instancias de una clase cargada dinámicamente no podrá hacer referencia al constructor de ésta estáticamente. En vez de eso, el entorno en tiempo de ejecución crea automáticamente una instancia de cada clase cada vez que es cargada, y la registra con un gestor de prototipos.

El principal inconveniente del patrón Prototype es que cada subclase de Prototipo debe implementar la operación Clonar, lo cual puede ser difícil. Por ejemplo, añadir Clonar es difícil cuando las clases ya existen. Igualmente, puede ser difícil implementar Clonar cuando sus interioridades incluyen objetos que no pueden copiarse o que tienen referencias circulares.

Patrones Relacionados

Prototype y Abstract Factory son patrones rivales en algunos aspectos, como se analiza al final de este capítulo. No obstante, también pueden usarse juntos. Una fábrica abstracta puede almacenar un conjunto de prototipos a partir de los cuales puede clonar y devolver objetos producto.

Los diseños que hacen un uso intensivo de los patrones Composite y Decorador suelen beneficiarse también de Prototype.

Ejemplo A.4 Código de ejemplo para el patrón Prototype

Prototipo.java

```
public interface Prototipo {
     public Object clonar();
     public String getNombre();
}
```

PrototipoConcreto1.java

```
public class PrototipoConcreto1 implements Prototipo{
     private String nombre;

     public PrototipoConcreto1(String nombre){
          this.nombre=nombre;
     }
```

```
     public String getNombre(){
          return nombre;
     }
     @Override
     public Object clonar() {
          return new PrototipoConcreto1(this.nombre);
     }
}
```

PrototipoConcreto2.java

```
public class PrototipoConcreto2 implements Prototipo{
     private String nombre;

     public PrototipoConcreto2(String nombre){
          this.nombre=nombre;
     }

     public String getNombre(){
          return nombre;
     }

     @Override
     public Object clonar() {
          return new PrototipoConcreto2(this.nombre);
     }

}
```

Cliente.java

```
public class Cliente {

     public static void main(String[] args) {
          PrototipoConcreto1 prototipo1 =
               new PrototipoConcreto1("Me llamo prototipo1");
          PrototipoConcreto2 prototipo2 =
               new PrototipoConcreto2("Me llamo prototipo2");

          operacion(prototipo1);
          operacion(prototipo2);
     }

     public static void operacion(Prototipo prototipo){
          Prototipo clonPrototipo = (Prototipo)prototipo.clonar();
          System.out.println("Soy un clon de '"+ clonPrototipo.getNombre()+"'");
     }
}
```

Salida en Consola

```
Soy un clon de 'Me llamo prototipo1'
Soy un clon de 'Me llamo prototipo2'
```

2.5 Singleton (Instancia Única)

Descripción

Hay veces que se necesita tener una y sólo una instancia de una clase determinada durante el tiempo de vida de una aplicación. El patrón Singleton se encarga y garantiza que una clase sólo tenga una instancia, y además proporciona un punto de acceso global a ella. Por ejemplo, podemos necesitar un único objeto para nuestra conexión a la base de datos en una aplicación.

Aplicabilidad

Se usa el patrón Singleton cuando:

- Deba existir exactamente una instancia de una clase, y ésta debe ser accesible a los clientes desde un punto de acceso conocido.
- La única instancia debería ser extensible mediante herencia, y los clientes deberían ser capaces de usar una instancia extendida sin modificar su código.

Diagrama UML

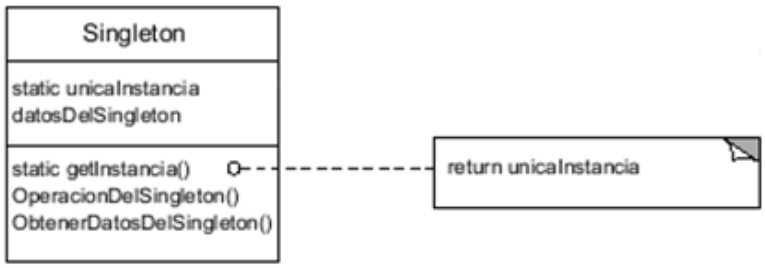

Figura B-15: Diagrama de clase para el patrón Singleton

Participantes

Singleton. Define una operación que permite que los clientes accedan a su única instancia. El método "getInstancia()" puede ser responsable de crear su única instancia referenciada por la variable "unicaInstancia".

Colaboraciones

Los clientes acceden a la instancia de un Singleton exclusivamente a través de la operación Instancia de éste.

Consecuencias

El patrón Singleton proporciona varios beneficios:

Acceso controlado a la única instancia. Puesto que la clase Singleton encapsula su única instancia, puede tener un control estricto sobre cómo y cuándo acceden a ella los clientes.

Espacio de nombre reducido. El patrón Singleton es una mejora sobre las variables globales. Evita contaminar el espacio de nombre con variables globales que almacenen las instancias.

Permite el refinamiento de operaciones y la representación. Se puede crear una subclase de la clase Singleton, y es fácil configurar una aplicación con una instancia de esta clase extendida. Se puede configurar la aplicación con una instancia de la clase necesaria en tiempo de ejecución.

Permite un número variable de instancias. El patrón hace que sea fácil cambiar de opinión y permitir más de una instancia de la clase Singleton. Además, se puede usar el mismo enfoque para controlar el número de instancias que usa la aplicación. Sólo se necesitaría cambiar la operación que otorga acceso a la instancia del Singleton.

Más flexible que las operaciones de la clase. Otra forma de empaquetar la funcionalidad de un Singleton es usando métodos de clase. Pero ambas técnicas de estos lenguajes dificultan cambiar un diseño para permitir más de una instancia de una clase.

Patrones Relacionados

Hay muchos patrones que pueden implementarse usando el patrón Singleton. Véanse el patrón Abstract Factory, Builder y Prototype.

Ejemplo A.5 Código de ejemplo para el patrón Singleton

Singleton.java

```
public class Singleton {
     private static Singleton UNICA_INSTANCIA = null;
     private String datosSingleton;

     private Singleton() {}

     public String getDatosSingleton() {
          return datosSingleton;
     }

     public void setDatosSingleton(String datosSingleton) {
          this.datosSingleton = datosSingleton;
     }

     private synchronized static void creaInstancia() {
          if (UNICA_INSTANCIA == null) {
```

```
                UNICA_INSTANCIA = new Singleton();
            }
        }

        @Override
        public Object clone() throws CloneNotSupportedException {
            throw new CloneNotSupportedException("Este objeto es Singleton"
                    + ", NO se puede clonar");
        }

        public static Singleton getInstancia() {
            if (UNICA_INSTANCIA == null){
                creaInstancia();
            }
            return UNICA_INSTANCIA;
        }
}
```

Cliente.java

```
public class Cliente {

        public static void main(String[] args) {
            Singleton instancia1 = Singleton.getInstancia();
            instancia1.setDatosSingleton("Datos iniciales de instancia1");

            Singleton instancia2 = Singleton.getInstancia();
            Singleton instancia3 = Singleton.getInstancia();

            System.out.println("datos instancia1: "+instancia1.getDatosSingleton());
            System.out.println("datos instancia2: "+instancia2.getDatosSingleton());
            System.out.println("datos instancia3: "+instancia3.getDatosSingleton());

            instancia3.setDatosSingleton("Nuevos datos seteados a instancia3");

            System.out.println("datos instancia1: "+instancia1.getDatosSingleton());
            System.out.println("datos instancia2: "+instancia2.getDatosSingleton());
            System.out.println("datos instancia3: "+instancia3.getDatosSingleton());
        }

}
```

Salida en Consola

```
datos instancia1: Datos iniciales de instancia1
datos instancia2: Datos iniciales de instancia1
datos instancia3: Datos iniciales de instancia1
datos instancia1: Nuevos datos seteados a instancia3
datos instancia2: Nuevos datos seteados a instancia3
datos instancia3: Nuevos datos seteados a instancia3
```

Capítulo 3: Patrones Estructurales

3.1 Adapter (Adaptador)

También conocido como: Wrapper (Envoltorio)

Descripción

El patrón Adapter se utiliza para transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda.

Convierte la interfaz de una clase en otra interfaz que el cliente espera. Adapter permite a las clases trabajar juntas, lo que de otra manera no podrían hacerlo debido a sus interfaces incompatibles.

Aplicabilidad

Debería usarse el patrón Adapter cuando:

- Se quiere usar una clase existente y su interfaz no concuerda con la que necesita.
- Se quiere crear una clase reutilizable que coopere con clases no relacionadas o que no han sido previstas, es decir, clases que no tienen por qué tener interfaces compatibles.
- (*Solamente en el caso de un adaptador de objetos*) es necesario usar varias subclases existentes, pero no resulta práctico adaptar su interfaz heredando de cada una de ellas. Un adaptador de objetos puede adaptar su interfaz heredando de cada una de ellas. Un adaptador de objetos puede adaptar la interfaz de su clase padre.

Diagrama UML

Un adaptador de clases usa la herencia múltiple para adaptar una interfaz a otra:

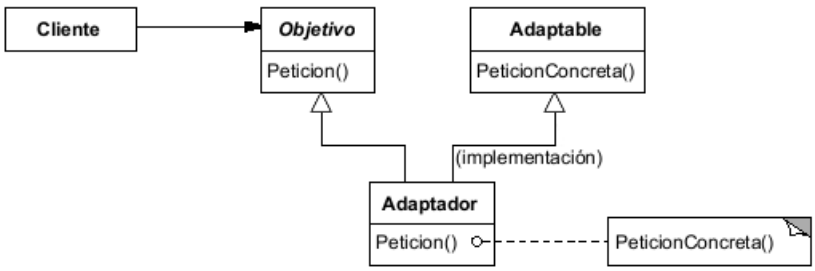

Figura B-16: Diagrama de clase para el patrón Adapter (herencia multiple)

Un adaptador de objetos se basa en la composición de objetos:

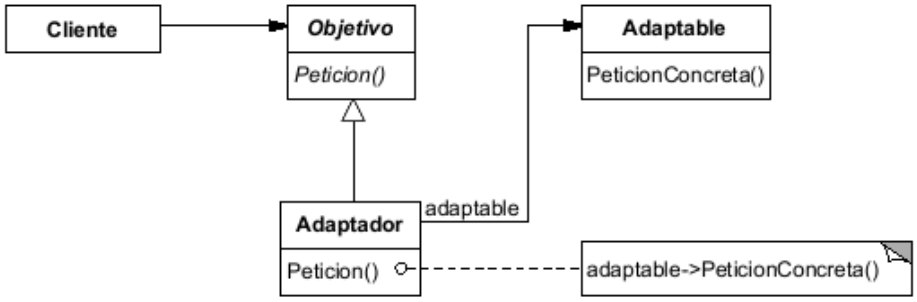

Figura B-17: Diagrama de clase para el patrón Adapter (composición objetos)

Participantes

Objetivo. Define la interfaz específica del dominio que usa el Cliente.

Cliente. Colabora con objetos que se ajustan a la interfaz Objetivo.

Adaptable. Define una interfaz existente que necesita ser adaptada.

Adaptador. Adapta la interfaz de Adaptable a la interfaz Objetivo.

Colaboraciones

Los clientes llaman a operaciones de una instancia de Adaptador. A su vez, el adaptador llama a operaciones de Adaptable, que son las que satisfacen la petición.

Consecuencias

Los adaptadores de clases y objetos tienen diferentes ventajas e inconvenientes. Un adaptador de clases:

Adapta una clase Adaptable a Objetivo, pero se refiere únicamente a una clase Adaptable concreta. Por tanto, un adaptador de clases no servirá cuando lo que se quiere es adaptar una clase y todas sus subclases.

Permite que Adaptador redefina parte del comportamiento de Adaptable, por ser Adaptador una subclase de Adaptable.

Introduce un solo objeto, y no se necesita ningún puntero de indirección adicional para obtener el objeto adaptado.

Por su parte, un adaptador de objetos:

- Permite que un mismo Adaptador funcione con muchos Adaptables, es decir, con el Adaptable en sí y todas sus subclases, en caso de que las tenga. El Adaptador también puede añadir funcionalidad a todos los Adaptables a la vez.
- Hace que sea más difícil redefinir el comportamiento de Adaptable. Se necesitará crear una subclase de Adaptable y hacer que el Adaptador se refiera a la subclase en vez de a la clase Adaptable en sí.

Patrones Relacionados

El patrón Bridge tiene una estructura similar a un adaptador de objetos, pero con un propósito diferente: está pensado para separar una interfaz de su implementación, de manera que ambos puedan cambiar fácilmente y de forma independiente uno del otro, mientras que un adaptador está pensado para cambiar la interfaz de un objeto existente.

El patrón Decorador decora otro objeto sin cambiar su interfaz. Un decorador es por tanto más transparente a la aplicación que un adaptador. Como resultado, el patrón Decorador permite la composición recursiva, lo que no es posible con adaptadores puros.

El patrón Proxy define un representante o sustituto de otro objeto sin cambiar su interfaz.

Ejemplo A.6 Código de ejemplo para el patrón Adapter

Adaptable.java

```
public class Adaptable {

     public String peticionConcreta(){
          return "Esta es una peticion concreta de la clase Adaptable";
     }

}
```

Adaptador.java

```
public class Adaptador implements Objetivo{

     @Override
     public String peticion() {
          Adaptable adaptable = new Adaptable();
          return adaptable.peticionConcreta();
     }

}
```

Objetivo.java

```
public interface Objetivo {
     public String peticion();
}
```

Cliente.java

```
public class Cliente {

     public static void main(String[] args) {
          Objetivo objetivo = new Adaptador();
          System.out.println("Realizo peticion: "+objetivo.peticion());
     }

}
```

Salida en Consola

```
Realizo peticion: Esta es una peticion concreta de la clase Adaptable
```

3.2 Bridge (Puente)

También conocido como: Handle/Body (Manejador/Cuerpo)

Descripción

El patrón Bridge promueve la separación de la interfaz de una abstracción de su implementación, de modo que ambas puedan variar de forma independiente. En general, el término abstracción se refiere al proceso de identificar el conjunto de atributos y el comportamiento de un objeto que es específico para un uso particular. Este punto de vista específico de un objeto puede ser concebido como un objeto separado omitiendo atributos irrelevantes y comportamiento. El objeto resultante en sí puede ser contemplado como una abstracción. Téngase en cuenta que un objeto determinado puede tener más de una abstracción asociada, cada uno con un uso distinto.

Aplicabilidad

Úsese el patrón Bridge cuando:

- Se quiere evitar un enlace permanente entre una abstracción y su implementación. Por ejemplo, cuando debe seleccionarse o cambiarse la implementación en tiempo de ejecución.
- Tanto las abstracciones como sus implementaciones deberían ser extensibles mediante subclases. En este caso, el patrón Bridge permite combinar las diferentes abstracciones y sus implementaciones, y extenderlas independientemente.
- Los cambios en la implementación de una abstracción no deberían tener impacto en los clientes; es decir, su código no tendría que ser recompilado.
- Se quiere compartir una implementación entre varios objetos (tal vez usando un contador de referencias) y este hecho deba permanecer oculto al cliente.

Diagrama UML

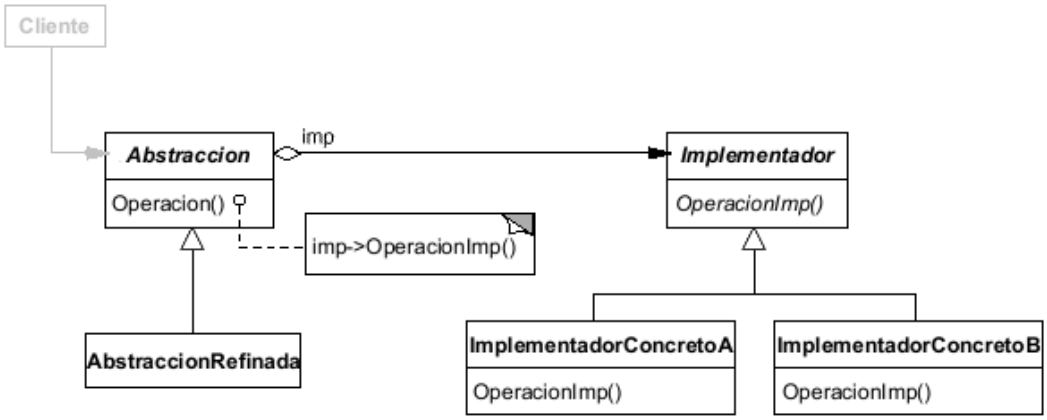

Figura B-18: Diagrama de clase para el patrón Bridge

Participantes

Abstraccion. Define la interfaz de la abstracción. Mantiene una referencia a un objeto de tipo Implementador.

AbstraccionRefinada. Extiende la interfaz por Abstracción.

Implementador. Define la interfaz de las clases de implementación. Esta interfaz no tiene por qué corresponderse exactamente con la de Abstracción; de hecho, ambas interfaces pueden ser muy distintas. Normalmente la interfaz Implementador sólo proporciona operaciones primitivas, y Abstracción define operaciones de más alto nivel basadas en dichas primitivas.

ImplementadorConcreto. Es la clase (ImplementadorConcretoA, ImplementadorConcretoB) encargada de realizar una implementación concreta de la interfaz declarada por "Implementador".

Colaboraciones

Abstracción redirige las peticiones del cliente a su objeto Implementador.

Consecuencias

El patrón Bridge tiene las siguientes consecuencias:

Desacopla la interfaz y la implementación. No une permanentemente una implementación a una interfaz, sino que la implementación puede configurarse en tiempo de ejecución. Incluso es posible que un objeto cambie su implementación en tiempo de ejecución.

Mejora la extensibilidad. Se pueden extender las jerarquías de Abstracción y de Implementación de forma independiente.

Oculta detalles de implementación a los clientes. Se puede aislar a los clientes de los detalles de implementación, como el compartimiento de objetos implementadores y el correspondiente mecanismo de conteo de referencias (si es que hay alguno).

Patrones Relacionados

El patrón Abstract Factory puede crear y configurar un Bridge.

El patrón Adapter está orientado a conseguir que trabajen juntas clases que no están relacionadas. Normalmente se aplica a sistemas que ya han sido diseñados. El patrón Bridge, por otro lado, se usa al comenzar un diseño para permitir que abstracciones e implementaciones varíen independientemente unas de otras.

Ejemplo A.7 Código de ejemplo para el patrón Bridge

Abstraccion.java

```
public interface Abstraccion {
     public String operacion();
}
```

AbstraccionRefinada.java

```
public class AbstraccionRefinada implements Abstraccion{
     Implementador implementador;

     public AbstraccionRefinada(Implementador implementador) {
          this.implementador = implementador;
     }

     @Override
     public String operacion() {
          return implementador.operacionImpl();
     }
}
```

Implementador.java

```
public interface Implementador {
     public String operacionImpl();
}
```

ImplementadorConcretoA.java

```
public class ImplementadorConcretoA implements Implementador{

     @Override
     public String operacionImpl() {
          return "Esta es la implementacion concreta A";
     }

}
```

ImplementadorConcretoB.java

```
public class ImplementadorConcretoB implements Implementador{

     @Override
     public String operacionImpl() {
```

```
            return "Esta es la implementacion concreta B";
      }

}
```

Cliente.java

```
public class Cliente {

      public static void main(String[] args) {

            Implementador implementadorA = new ImplementadorConcretoA();
            Implementador implementadorB = new ImplementadorConcretoB();

            Abstraccion abstraccion = new AbstraccionRefinada(implementadorA);
            System.out.println("Con implementadorA: "+abstraccion.operacion());

            abstraccion = new AbstraccionRefinada(implementadorB);
            System.out.println("Con implementadorB: "+abstraccion.operacion());

      }

}
```

Salida en Consola

```
Con implementadorA: Esta es la implementacion concreta A
Con implementadorB: Esta es la implementacion concreta B
```

3.3 Composite (Compuesto)

Descripción

Cada componente u objeto puede ser clasificado en una de dos categorías: (1) Componentes Individuales o (2) Componentes Compuestos, es decir, que se componen de componentes individuales o componentes compuestos. El patrón Composite es útil en el diseño de una interfaz común que permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos. En otras palabras el patrón de diseño Composite permite que un objeto cliente trate a un componente individual y a un componente compuesto de forma idéntica.

Aplicabilidad

Se usa el patrón Composite cuando:

- Se quiere representar jerarquías de objetos parte-todo.
- Se desea que los clientes sean capaces de obviar las diferencias entre composiciones de objetos y objetos individuales. Los clientes tratarán a todos los objetos de la estructura compuesta de manera uniforme.

Diagrama UML

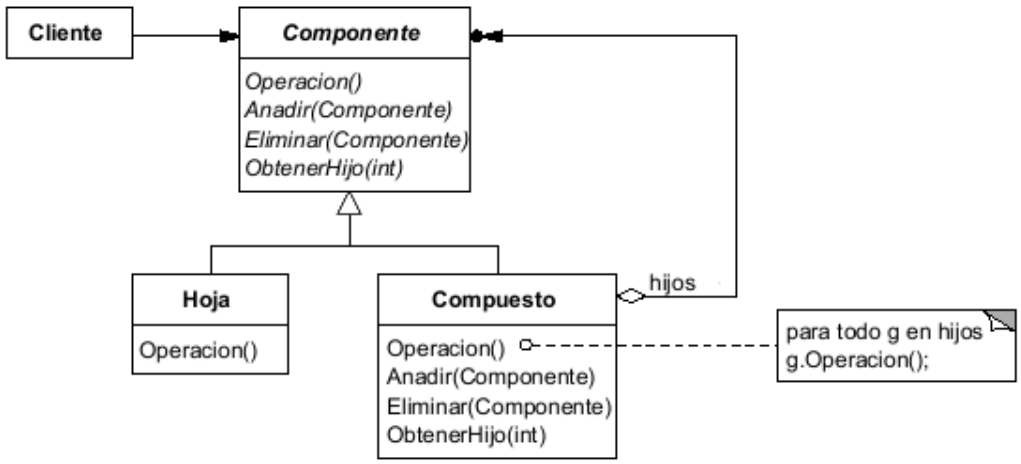

Figura B-19: Diagrama de clase para el patrón Composite

Participantes

Componente. Declara la interfaz de los objetos de la composición. Implementa el comportamiento predeterminado de la interfaz que es común a todas las clases. Declara una

interfaz para acceder a sus componentes hijos y gestionarlos. (Opcional) define una interfaz para acceder al padre de un componente en la estructura recursiva y, si es necesario, la implementa.

Hoja. Representa objetos hoja en composición. Una hoja no tiene hijos. Define el comportamiento de los objetos primitivos de la composición.

Compuesto. Define el comportamiento de los componentes que tienen hijos. Almacena componentes hijos. Implementa las operaciones de la interfaz Componente relacionadas con los hijos.

Cliente. Manipula objetos en la composición a través de la interfaz Componente.

Colaboraciones

Los clientes usan la interfaz de la clase Componente para interactuar con los objetos de la estructura compuesta. Si el recipiente es una Hoja, la petición se trata correctamente. Si es un Compuesto, normalmente redirige las peticiones a sus componentes hijos, posiblemente realizando operaciones adicionales antes o después.

Consecuencias

El patrón Composite:

Define jerarquías de clases formadas por objetos primitivos y compuestos. Los objetos primitivos pueden componerse en otros objetos más complejos, que a su vez pueden ser compuestos, y así de manera recurrente. Allí donde el código espere un objeto primitivo, también podrá recibir un objeto compuesto.

Simplifica el cliente. Los clientes pueden tratar uniformemente a las estructuras compuestas y a los objetos individuales. Los clientes normalmente no conocen (y no les debería importar) si están tratando con una hoja o con un componente compuesto. Esto simplifica el código del cliente, puesto que evita tener que escribir funciones con instrucciones *if* anidadas en las clases que definen la composición.

Facilita añadir nuevos tipos componentes. Si se definen nuevas subclases Compuesto u Hoja, éstas funcionarán automáticamente con las estructuras y el código cliente existente. No hay que cambiar los clientes para las nuevas clases Componente.

Pueden hacer que un diseño sea demasiado general. La desventaja de facilitar añadir nuevos componentes es que hace más difícil restringir los componentes de un compuesto. A veces se requiere que un compuesto sólo tenga ciertos componentes. Con el patrón Composite, no se

puede confiar en el sistema de tipos para que haga cumplir estas restricciones por nosotros. En vez de eso, tendrán que usar comprobaciones en tiempo de ejecución.

Patrones Relacionados

Muchas veces se usa el enlace al componente padre para implementar el patrón Chain of Responsability.

El patrón Decorator suele usarse junto con el Composite. Cuando se usan juntos decoradores y compuestos, normalmente ambos tendrán una clase padre común. Por tanto, los decoradores tendrán que admitir la interfaz Componente con operaciones como Anadir, Eliminar y ObtenerHijo.

El patrón Flyweight permite compartir componentes, si bien en ese caso éstos ya no pueden referirse a sus padres.

Se puede usar el patrón Iterator para recorrer las estructuras definidas por el patrón Composite.

El patrón Visitor localiza operaciones y comportamiento que de otro modo estaría distribuido en varias clases Compuesto y Hoja.

Ejemplo A.8 Código de ejemplo para el patrón Composite

Componente.java

```
public abstract class Componente {
     private String nombre;

     public Componente(String nombre){
          this.nombre = nombre;
     }

     public String getNombre() {
          return nombre;
     }
     public void setNombre(String nombre) {
          this.nombre = nombre;
     }

     public abstract String operacion();
     public abstract void agregar(Componente hijo);
     public abstract void eliminar(Componente hijo);
     public abstract Componente obtenerHijo(int id);
}
```

Compuesto.java

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Compuesto extends Componente{

     List<Componente> hijos = new ArrayList<Componente>();
```

```
    public Compuesto(String nombre) {
        super(nombre);
    }

    @Override
    public void agregar(Componente hijo) {
        hijos.add(hijo);
    }

    @Override
    public void eliminar(Componente hijo) {
        hijos.remove(hijo);
    }

    @Override
    public Componente obtenerHijo(int id) {
        Componente hijoEncontrado = null;
        if(hijos.size()>=id && id>0){
            hijoEncontrado = hijos.get(id-1);
        }
        return hijoEncontrado;
    }

    @Override
    public String operacion() {
        String respuesta = "\n\nNombre de Compuesto: "+getNombre();
        respuesta += "\nHijos:";
        for (Iterator<Componente> iterator = hijos.iterator(); iterator.hasNext();) {
            Componente hijo = iterator.next();
            respuesta +=hijo.operacion();
        }
        return respuesta;
    }

}
```

Hoja.java

```
public class Hoja extends Componente{

    public Hoja(String nombre) {
        super(nombre);
    }

    @Override
    public void agregar(Componente hijo) {
        System.out.println("No puedo agregar hijos, soy Hoja");
    }

    @Override
    public void eliminar(Componente hijo) {
        System.out.println("No tengo hijos, soy Hoja");
    }

    @Override
    public Componente obtenerHijo(int id) {
        System.out.println("No tengo hijos, soy Hoja");
        return null;
    }

    @Override
    public String operacion() {
        return "\n\tNombre de Hoja: "+getNombre();
    }

}
```

Cliente.java

```
public class Cliente {
```

```
     public static void main(String[] args) {
          Componente hoja1 = new Hoja("hoja1");
          Componente hoja2 = new Hoja("hoja2");
          Componente hoja3 = new Hoja("hoja3");
          Componente hoja4 = new Hoja("hoja4");
          Componente hoja5 = new Hoja("hoja5");
          Componente hoja6 = new Hoja("hoja6");
          Componente hoja7 = new Hoja("hoja7");
          Componente hoja8 = new Hoja("hoja8");
          Componente hoja9 = new Hoja("hoja9");

          Componente rama1 = new Compuesto("rama1");
          Componente rama2 = new Compuesto("rama2");
          Componente rama3 = new Compuesto("rama3");

          Componente raiz = new Compuesto("raiz del arbol");

          raiz.agregar(rama1);
          raiz.agregar(rama2);
          raiz.agregar(rama3);

          rama1.agregar(hoja1);
          rama1.agregar(hoja2);
          rama1.agregar(hoja3);
          rama1.agregar(hoja4);

          rama2.agregar(hoja5);
          rama2.agregar(hoja6);
          rama2.agregar(hoja7);

          rama3.agregar(hoja8);
          rama3.agregar(hoja9);
          rama3.agregar(hoja1);
          rama3.agregar(hoja2);

          System.out.println("1) Estructura del arbol inicial: "+raiz.operacion());

          raiz.eliminar(rama3);
          raiz.eliminar(rama2);
          rama1.eliminar(hoja3);
          rama1.eliminar(hoja4);

          System.out.println("\n2) Despues de eliminar algunos componentes:
"+raiz.operacion());

          System.out.println("\n3) Obtengo el segundo hijo de rama1:
"+rama1.obtenerHijo(2).getNombre());

     }
}
```

Salida en Consola

```
1) Estructura del arbol inicial:

Nombre de Compuesto: raiz del arbol
Hijos:

Nombre de Compuesto: rama1
Hijos:
     Nombre de Hoja: hoja1
     Nombre de Hoja: hoja2
     Nombre de Hoja: hoja3
     Nombre de Hoja: hoja4

Nombre de Compuesto: rama2
Hijos:
     Nombre de Hoja: hoja5
     Nombre de Hoja: hoja6
     Nombre de Hoja: hoja7

Nombre de Compuesto: rama3
```

```
Hijos:
     Nombre de Hoja: hoja8
     Nombre de Hoja: hoja9
     Nombre de Hoja: hoja1
     Nombre de Hoja: hoja2

2) Despues de eliminar algunos componentes:

Nombre de Compuesto: raiz del arbol
Hijos:

Nombre de Compuesto: rama1
Hijos:
     Nombre de Hoja: hoja1
     Nombre de Hoja: hoja2

3) Obtengo el segundo hijo de rama1: hoja2
```

3.4 Decorator (Decorador)

También conocido como: Wrapper (Envoltorio)

Descripción

El patrón Decorator es utilizado para extender la funcionalidad de un objeto de forma dinámica sin tener que cambiar el código de la clase original o el uso de la herencia. Esto se logra mediante la creación de un objeto envoltorio que se refiere como un decorador alrededor de un objeto existente.

Aplicabilidad

Úsese el Decorador:

- Para añadir objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos.
- Para responsabilidades que puedan ser retiradas.
- Cuando la extensión mediante la herencia no es viable. A veces es posible tener un gran número de extensiones independientes, produciéndose una explosión de subclases para permitir todas las combinaciones. O puede ser que una definición de una clase esté oculta o que no esté disponible para ser heredada.

Diagrama UML

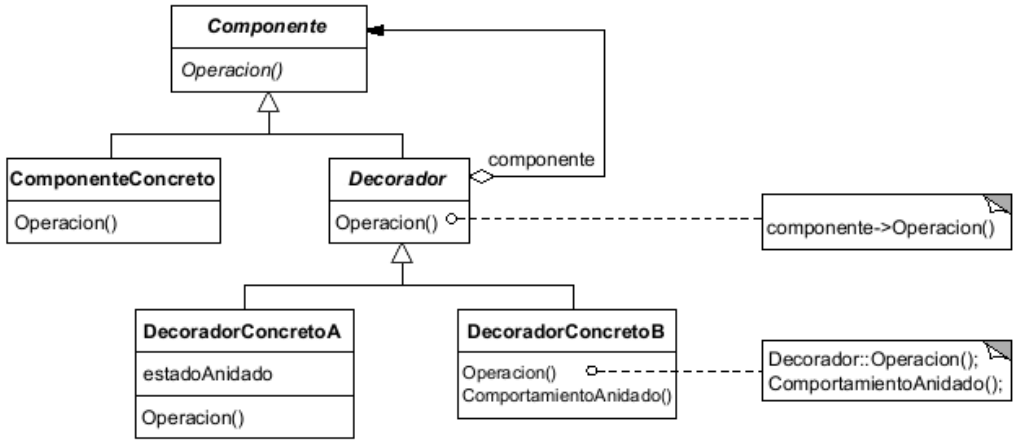

Figura B-20: Diagrama de clase para el patrón Decorator

Participantes

Componente. Define la interfaz para objetos a los que se puede añadir responsabilidades dinámicamente.

ComponenteConcreto. Define un objeto al que se pueden añadir responsabilidades adicionales.

Decorador. Mantiene una referencia a un objeto Componente y define una interfaz que se ajusta a la interfaz del Componente.

DecoradorConcreto. Añade responsabilidades al componente.

Colaboraciones

El Decorador redirige peticiones a su objeto Componente. Opcionalmente puede realizar operaciones adicionales antes y después de reenviar la petición.

Consecuencias

El patrón Decorador tiene al menos dos ventajas y dos inconvenientes fundamentales:

Más flexibilidad que la herencia estática. El patrón Decorador proporciona una manera más flexible de añadir responsabilidades a los objetos que la que podía obtenerse a través de la herencia (múltiple) estática. Con los decoradores se pueden añadir y eliminar responsabilidades en tiempo de ejecución simplemente poniéndolas y quitándolas. Por el contrario, la herencia requiere crear una nueva clase para cada responsabilidad adicional. Esto da lugar a muchas clases diferentes e incrementa la complejidad de un sistema. Por otro lado, proporcionar diferentes clases Decorador para una determinada clase Componente permite mezclar responsabilidades. Los Decoradores también facilitan añadir una propiedad dos veces.

Evita clases cargadas de funciones en la parte superior de la jerarquía. El Decorador ofrece un enfoque para añadir responsabilidades que consiste en pagar sólo aquello que se necesita. En vez de tratar de permitir todas las funcionalidades inimaginables en una clase compleja y adaptable, definimos primero una clase simple y se añade luego funcionalidad incrementalmente con objetos Decorador. La funcionalidad puede obtenerse componiendo partes simples. Como resultado, una aplicación no necesita pagar por características que no usa. También resulta fácil definir nuevos tipos de Decoradores independientemente de las clases de objetos de las que heredan, incluso para extensiones que no hubieran sido previstas. Extender una clase compleja tiende a exponer detalles no relacionados con las responsabilidades que se están añadiendo.

Un decorador y su componente no son idénticos. Un decorador se comporta como revestimiento transparente. Pero desde el punto de vista de la identidad de un objeto, un componente decorado no es idéntico al componente en sí. Por tanto, no deberíamos apoyarnos en la identidad de objetos cuando se estén usando decoradores.

Muchos objetos pequeños. Un diseño que usa el patrón Decorador suele dar como resultado sistemas formados por muchos objetos pequeños muy parecidos. Los objetos sólo se diferencian en la forma en que están interconectados, y no en su clase o en el valor de sus variables. Aunque dichos sistemas son fáciles de adaptar por parte de quienes los comprenden bien, pueden ser difíciles de aprender y de depurar.

Patrones Relacionados

Adapter: un decorador se diferencia de un adaptador en que el decorador sólo cambia las responsabilidades de un objeto, no su interfaz, mientras que un adaptador le da a un objeto una interfaz completamente nueva.

Composite: se puede ver a un decorador como un compuesto degenerado que sólo tiene un componente. No obstante, un decorador añade responsabilidades adicionales (no está pensado para la agregación de objetos).

Strategy: un decorador permite cambiar el exterior de un objeto; una estrategia permite cambiar sus tripas. Son dos formas alternativas de modificar un objeto.

Ejemplo A.9 Código de ejemplo para el patrón Decorator

Componente.java

```
public interface Componente {
     public String operacion();
}
```

ComponenteConcreto.java

```
public class ComponenteConcreto implements Componente{

     @Override
     public String operacion() {
          return "Operacion de ComponenteConcreto. ";
     }

}
```

Decorador.java

```
public abstract class Decorador implements Componente{
     private Componente componente;

     Decorador(Componente componente){
          this.componente = componente;
```

```
    }

    @Override
    public String operacion() {
        return componente.operacion();
    }

}
```

DecoradorConcretoA.java

```
public class DecoradorConcretoA extends Decorador{
    private String estadoAnidado = "Atributo agregado. ";

    DecoradorConcretoA(Componente componente) {
        super(componente);
    }

    public String operacion() {
        return super.operacion() + estadoAnidado;
    }

}
```

DecoradorConcretoB.java

```
public class DecoradorConcretoB extends Decorador{

    DecoradorConcretoB(Componente componente) {
        super(componente);
    }

    public String operacion() {
        return super.operacion() + comportamientoAnidado();
    }

    private String comportamientoAnidado(){
        return "Comportamiento agregado. ";
    }

}
```

Cliente.java

```
public class Cliente {

    public static void main(String[] args) {
        Componente componente = new ComponenteConcreto();
        System.out.println("1) "+componente.operacion());
        componente = new DecoradorConcretoA(componente);
        System.out.println("2) "+componente.operacion());
        componente = new DecoradorConcretoB(componente);
        System.out.println("3) "+componente.operacion());
    }

}
```

Salida en Consola

```
1) Operacion de ComponenteConcreto.
2) Operacion de ComponenteConcreto. Atributo agregado.
3) Operacion de ComponenteConcreto. Atributo agregado. Comportamiento agregado.
```

3.5 Facade (Fachada)

Descripción

El patrón de diseño Facade sirve para proveer una interfaz unificada sencilla que haga de intermediaria entre un cliente y una interfaz o grupo de interfaces más complejas.

Facade puede hacer una biblioteca de software más fácil de usar y entender, ya que facade implementa métodos convenientes para tareas comunes y permite que el código que usa la biblioteca sea más legible, por la misma razón; puede reducir la dependencia de código externo en los trabajos internos de una biblioteca, ya que la mayoría del código lo usa Facade, permitiendo así más flexibilidad en el desarrollo de sistemas; y puede envolver una colección mal diseñada de APIs con un solo API bien diseñado.

Aplicabilidad

Se usa el patrón Facade cuando:

- Se quiere proporcionar una interfaz simple para un subsistema complejo. Los subsistemas suelen volverse más complicados a medida que van evolucionando. La mayoría de los patrones, cuando se aplican, dan como resultado más clases y más pequeñas. Esto hace que un subsistema sea más reutilizable y fácil de personalizar, pero eso también lo hace más difícil de usar para aquellos clientes que no necesitan personalizarlo. Una fachada puede proporcionar, por omisión una vista simple del subsistema que resulta adecuada para la mayoría de los clientes. Sólo aquellos clientes que necesitan más personalización necesitarán ir más allá de la fachada.
- Existen muchas dependencias entre los clientes y las clases que implementan una abstracción. Se introduce una fachada para desacoplar el subsistema de sus clientes y de otros subsistemas, promoviendo así la independencia entre subsistemas y la portabilidad.
- Se quieren dividir en capas nuestros subsistemas. Se usa una fachada para definir un punto de entrada en cada nivel del subsistema. Si éstos son dependientes, se pueden simplificar las dependencias entre ellos haciendo que se comuniquen entre sí únicamente a través de sus fachadas.

Diagrama UML

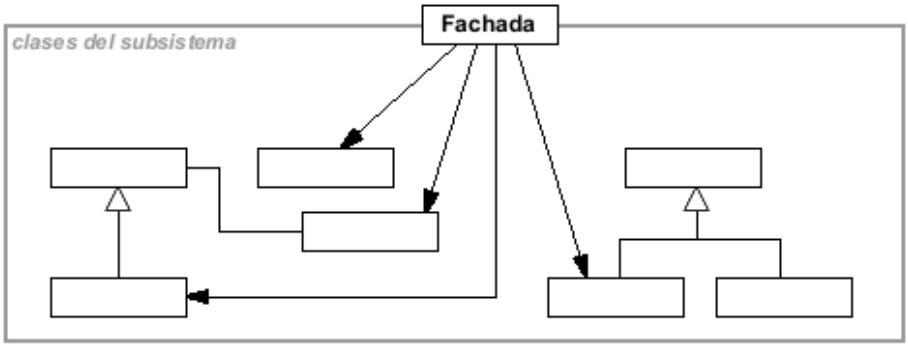

Figura B-21: Diagrama de clase para el patrón Facade

Participantes

Fachada. Sabe qué clases del subsistema son las responsables ante una petición. Delega las peticiones de los clientes en los objetos apropiados del subsistema.

Clases del subsistema. Implementan la funcionalidad del subsistema. Realizan las labores encomendadas por el objeto Fachada. No conocen a la fachada; es decir, no tienen referencias de ella.

Colaboraciones

Los clientes se comunican con el subsistema enviando peticiones al objeto Fachada, el cual las reenvía a los objetos apropiados del subsistema. Aunque son los objetos del subsistema los que realizan el trabajo real, la fachada puede tener que hacer algo de trabajo para pasar de su interfaz a las del subsistema.

Los clientes que usan la fachada no tienen que acceder directamente a los objetos del subsistema.

Consecuencias

El patrón Facade proporciona las siguientes ventajas:

Oculta a los clientes los componentes del subsistema, reduciendo así el número de objetos con los que tratan los clientes y haciendo que el subsistema sea más fácil de usar.

Promueve un débil acoplamiento entre el subsistema y sus clientes. Muchas veces los componentes de un subsistema están fuertemente acoplados. Un acoplamiento débil permite modificar los componentes del subsistema sin que sus clientes se vean afectados. Las fachadas ayudan a estructurar en capas un sistema y las dependencias entre los objetos. También pueden eliminar

dependencias complejas o circulares. Esto puede ser una consistencia importante cuando el cliente y el subsistema se implementan por separado.

No impide que las aplicaciones usen las clases del subsistema en caso de que sea necesario. De este modo se puede elegir entre facilidad de uso y generalidad.

Patrones Relacionados

El patrón Abstract Factory puede usarse para proporcionar una interfaz para crear el subsistema de objetos de forma independiente a otros subsistemas. Las fábricas abstractas también pueden ser una alternativa a las fachadas para ocultar clases específicas de la plataforma.

El patrón Mediator es parecido al Facade en el sentido de que abstrae funcionalidad a partir de ciertas clases existentes. Sin embargo, el propósito del Mediador es abstraer cualquier comunicación entre objetos similares, a menudo centralizando la funcionalidad que no pertenece a ninguno de ellos. Los colegas de un mediador sólo se preocupan de comunicarse con él y no entre ellos directamente. Por el contrario, una fachada simplemente abstrae una interfaz para los objetos del subsistema, haciéndolos más fácil de usar; no define nueva funcionalidad, y las clases del subsistema no saben de su existencia.

Normalmente sólo necesita un objeto Fachada. Por tanto, éstos suelen implementarse con Singletons.

Ejemplo A.10 Código de ejemplo para el patrón Facade

El siguiente ejemplo la clase Computadora actúa como fachada y las clases Cpu, DiscoDuro y Memoria representan las clases del subsistema. La clase Cliente hace uso de la fachada (Computadora).

Computadora.java

```
/* Fachada */
public class Computadora {
     private static final String DIRECCION_ARRANQUE = "382FBD652E91A";
     private static final String SECTOR_ARRANQUE = "34234";
     private static final String TAMANO_SECTOR = "32Kb";
     private Cpu cpu;
     private Memoria memoria;
     private DiscoDuro disco;

     public Computadora(){
          this.cpu = new Cpu();
          this.memoria = new Memoria();
          this.disco = new DiscoDuro();
     }

     public void prender(){
          cpu.enciendeVentilador();
          memoria.cargar(DIRECCION_ARRANQUE, disco.read(SECTOR_ARRANQUE, TAMANO_SECTOR));
        cpu.jump(DIRECCION_ARRANQUE);
```

```
        cpu.executar();

    }
}
```

Cpu.java

```
public class Cpu {

    public void enciendeVentilador() {
        System.out.println("Encendiendo el ventilador");
    }

    public void jump(String direccionArranque) {
        System.out.println("CPU jump. ");
    }

    public void executar() {
        System.out.println("CPU executando comandos de arranque. ");
    }

}
```

DiscoDuro.java

```
public class DiscoDuro {

    public String read(String sectorArranque, String tamanoSector) {
        return "Iniciado sector de arranque "+sectorArranque+ ", con tamaño "
            +tamanoSector+ " en el disco duro. ";
    }

}
```

Memoria.java

```
public class Memoria {

    public void cargar(String direccionArranque, String read) {
        System.out.println(read+"\nCargando direccion de arranque "
                +direccionArranque+" en Memoria.");
    }

}
```

Cliente.java

```
public class Cliente {

    public static void main(String[] args) {
        Computadora computadora = new Computadora();
        computadora.prender();
    }

}
```

Salida en Consola

```
Encendiendo el ventilador
Iniciado sector de arranque 34234, con tamaño 32Kb en el disco duro.
Cargando direccion de arranque 382FBD652E91A en Memoria.
CPU jump.
CPU executando comandos de arranque.
```

3.6 Flyweight (Peso Ligero)

Descripción

Cada objeto puede ser visto como parte de uno o ambos de los siguientes dos conjuntos de información:

Información intrínseca - La información intrínseca de un objeto es independiente del contexto del objeto. Eso significa que la información intrínseca es la información común que permanece constante entre las diferentes instancias de una clase determinada. Por ejemplo, la información de cierta compañía para una tarjeta de visita es la misma para todos los empleados.

Información extrínseca - La información extrínseca de un objeto depende y varía con el contexto del objeto. Eso significa que la información extrínseca es única para cada instancia de una clase determinada. Por ejemplo, el nombre del empleado y el título son extrínsecas en una tarjeta de visita porque que esa información es única para cada empleado.

El patrón de Flyweight (peso mosca) sugiere separar todos los datos intrínsecos comunes en un objeto independiente siendo éste un objeto peso mosca. El grupo de objetos que están siendo creados pueden compartir el objeto peso mosca, ya que representa su estado intrínseco. Esto elimina la necesidad de almacenar la misma invariante de la información intrínseca de cada objeto, para que se almacene sólo una vez en la forma de un objeto de peso mosca único.

Como resultado, la aplicación cliente puede realizar un ahorro considerable en términos de uso de memoria y tiempo.

Aplicabilidad

La efectividad del patrón Flyweight depende enormemente de cómo y dónde se use. Debería aplicarse el patrón cuando se cumpla *todo* lo siguiente:

- Una aplicación utiliza un gran número de objetos.
- Los costes de almacenamiento son elevados debido a la gran cantidad de objetos.
- La mayor parte del estado del objeto puede hacerse extrínseco.
- Muchos grupos y objetos pueden reemplazarse por relativamente pocos objetos compartidos, una vez que se ha eliminado el estado extrínseco.
- La aplicación no depende de la identidad de un objeto. Puesto que los objetos peso ligero pueden ser compartidos, las comprobaciones de identidad devolverán verdadero para objetos conceptualmente distintos.

Diagrama UML

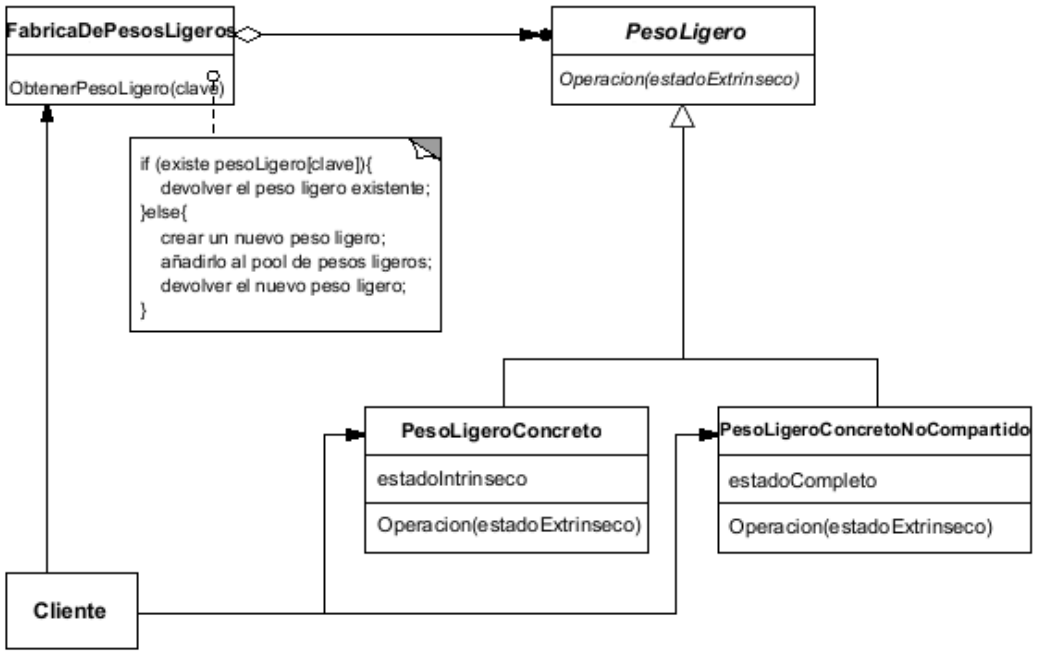

Figura B-22: Diagrama de clase para el patrón Flyweight

Participantes

PesoLigero. Declara una interfaz a través de la cual los pesos ligeros pueden recibir un estado extrínseco y actuar sobre él.

PesoLigeroConcreto. Implementa la interfaz PesoLigero y permite almacenar el estado intrínseco, en caso de que lo haya. Un objeto PesoLigeroConcreto debe poder ser compartido, por lo que cualquier estado que almacene debe ser intrínseco, esto es, debe ser independiente del contexto del objeto PesoLigeroConcreto.

PesoLigeroConcretoNoCompartido. No todas las subclases de PesoLigero necesitan ser compartidas. La interfaz PesoLigero permite el comportamiento, no fuerza a él. Los objetos PesoLigeroConcretoNoCompartido suelen tener objetos PesoLigeroConcreto como hijos en algún nivel de la estructura de objetos.

FabricaDePesosLigeros. Crea y controla objetos pesos ligeros. Garantiza que los pesos ligeros se compartan de una manera adecuada. Cuando un cliente solicita un peso ligero, el objeto FabricaDePesosLigeros proporciona una instancia concreta o crea uno nuevo, en caso de que no exista ninguno.

Cliente. Mantiene una referencia a los pesos ligeros. Calcula o guarda el estado extrínseco de los pesos ligeros.

Colaboraciones

El estado que un peso ligero necesita para funcionar debe ser caracterizado como intrínseco o extrínseco. El estado intrínseco se guarda en el objeto PesoLigeroConcreto, mientras que el estado extrínseco lo guardan o lo calculan objetos Cliente. Los clientes pasan este estado al peso ligero cuando invocan sus operaciones.

Los clientes no deberían crear instancias de PesoLigeroConcreto directamente, sino que deben obtener los objetos PesoLigeroConcreto sólo a partir del objeto FabricaDePesosLigeros para garantizar que se puedan compartir adecuadamente.

Consecuencias

Los pesos ligeros pueden introducir costes en tiempo de ejecución asociados con la transferencia, búsqueda y cálculo del estado extrínseco, especialmente si éste se almacenó en primer lugar como estado intrínseco. En cualquier caso, dichos costes se ven compensados por el ahorro de espacio de almacenamiento, que se incrementa a medida que se comparten más objetos.

El ahorro de almacenamiento está en función de varios factores:

- La reducción en el número total de instancias lograda mediante el compartimiento
- La cantidad de estado intrínseco por objeto
- Si el estado extrínseco se calcula o se almacena.

Cuantos más objetos peso ligero se compartan, mayor será el ahorro de almacenamiento. Este ahorro aumentará a medida que se comparta más cantidad de estado. El mayor ahorro tendrá lugar cuando los objetos tengan gran cantidad de estado, tanto intrínseco como extrínseco, y cuando el estado extrínseco pueda calcularse en vez de tener que ser guardado. De esa manera se ahorra espacio de almacenamiento de dos formas: el comportamiento reduce el coste del estado intrínseco, y se cambia estado extrínseco por tiempo de cálculo.

Patrones Relacionados

El patrón Flyweight suele combinarse con el patrón Composite para implementar una estructura lógica jerárquica en términos de un grafo dirigido acíclico con nodos hojas compartidas.

Suele ser mejor implementar los objetos State y Strategy como pesos ligeros.

Ejemplo A.11 Código de ejemplo para el patrón Flyweight

FabricaDePesosLigeros.java

```
import java.util.HashMap;

public class FabricaDePesosLigeros {
     HashMap<String, PesoLigeroConcreto> pesosLigeros =
                              new HashMap<String, PesoLigeroConcreto>();

     public PesoLigeroConcreto obtenerPesoLigero(String clave){
          if(pesosLigeros.containsKey(clave)){
               return (PesoLigeroConcreto) pesosLigeros.get(clave);
          }
          PesoLigeroConcreto pesoLigeroConcreto = new PesoLigeroConcreto();
          pesosLigeros.put(clave, pesoLigeroConcreto);
          return pesoLigeroConcreto;
     }
}
```

PesoLigero.java

```
public interface PesoLigero {
     public String operacion(String estadoExtrinseco);
}
```

PesoLigeroConcreto.java

```
public class PesoLigeroConcreto implements PesoLigero{
     String estadoIntrinseco;

     @Override
     public String operacion(String estadoExtrinseco) {
          return "* Estado intrinseco: "+estadoIntrinseco
                    +", estado extrinseco: "+estadoExtrinseco;
     }

     public String getEstadoIntrinseco() {
          return estadoIntrinseco;
     }

     public void setEstadoIntrinseco(String estadoIntrinseco) {
          this.estadoIntrinseco = estadoIntrinseco;
     }

}
```

PesoLigeroConcretoNoCompartido.java

```
public class PesoLigeroConcretoNoCompartido implements PesoLigero{
     private String estadoCompleto;

     public PesoLigeroConcretoNoCompartido(String estadoCompleto){
          this.estadoCompleto = estadoCompleto;
     }

     public String getEstadoCompleto() {
          return estadoCompleto;
     }

     @Override
     public String operacion(String estadoExtrinseco) {
          return "> Hola "+estadoExtrinseco+"!, "+estadoCompleto;
     }

}
```

Cliente.java

```
public class Cliente {

     public static void main(String[] args) {

          FabricaDePesosLigeros fabrica = new FabricaDePesosLigeros();

          PesoLigeroConcreto credencial = fabrica.obtenerPesoLigero("credencialUnam");
          credencial.setEstadoIntrinseco("Titulo credencial UNAM");

          credencial = fabrica.obtenerPesoLigero("credencialFESAcatlan");
          credencial.setEstadoIntrinseco("Titulo credencial FES Acatlan");


          PesoLigeroConcreto credencialFes =
fabrica.obtenerPesoLigero("credencialFESAcatlan");
          PesoLigeroConcreto credencialUnam = fabrica.obtenerPesoLigero("credencialUnam");

          System.out.println("Credenciales con clave 'credencialFESAcatlan', comparten el
mismo titulo:");
          System.out.println(credencialFes.operacion("Juan Diaz Lopez"));
          System.out.println(credencialFes.operacion("Julieta Garcia Perez"));
          System.out.println(credencialFes.operacion("Pablo Fernandez Sandoval"));

          System.out.println("\nCredenciales con clave 'credencialUnam', comparten el mismo
titulo:");
          System.out.println(credencialUnam.operacion("Adolfo Bautista Rojas"));
          System.out.println(credencialUnam.operacion("Carolina Herrera Gonzales"));

          System.out.println("\nPeso Ligero No Compartido que permite el mismo
comportamiento:");
          System.out.println(new
PesoLigeroConcretoNoCompartido("felicidades.").operacion("Roberto"));
     }

}
```

Salida en Consola

```
Credenciales con clave 'credencialFESAcatlan', comparten el mismo titulo:
* Estado intrinseco: Titulo credencial FES Acatlan, estado extrinseco: Juan Diaz Lopez
* Estado intrinseco: Titulo credencial FES Acatlan, estado extrinseco: Julieta Garcia Perez
* Estado intrinseco: Titulo credencial FES Acatlan, estado extrinseco: Pablo Fernandez
Sandoval

Credenciales con clave 'credencialUnam', comparten el mismo titulo:
* Estado intrinseco: Titulo credencial UNAM, estado extrinseco: Adolfo Bautista Rojas
* Estado intrinseco: Titulo credencial UNAM, estado extrinseco: Carolina Herrera Gonzales

Peso Ligero No Compartido que permite el mismo comportamiento:
> Hola Roberto!, felicidades.
```

3.7 Proxy (Apoderado)

También conocido como: Surrogate (Sustituto)

Descripción

Proporciona un representante o sustituto de otro objeto para controlar el acceso a éste. Se utiliza para acceder a un objeto, permitiendo controlar su acceso.

Aplicabilidad

Este patrón es aplicable cada vez que hay necesidad de una referencia a un objeto más versátil o sofisticado que un simple puntero. Éstas son varias situaciones comunes en las que es aplicable el patrón Proxy:

- Un proxy remoto proporciona un representante local de un objeto situado en otro espacio de direcciones.
- Un proxy virtual crea objetos costosos por encargo.
- Un proxy de protección controla el acceso al objeto original. Los proxies de protección son útiles cuando los objetos debieran tener diferentes permisos de acceso.
- Una referencia inteligente es un sustituto de un simple puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto.

Diagrama UML

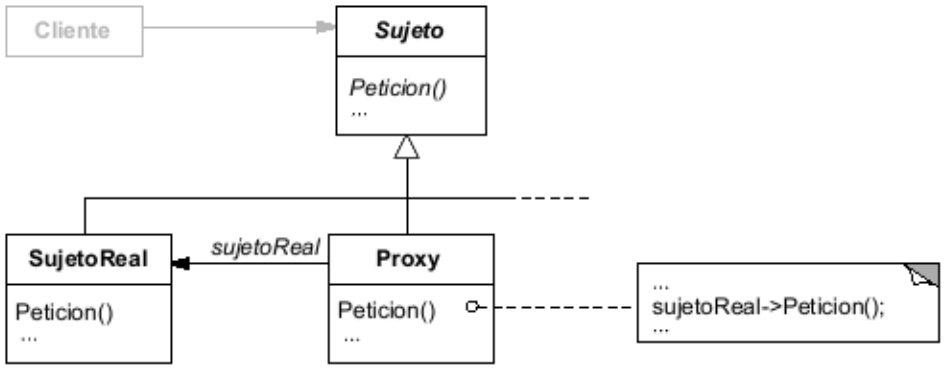

Figura B-23: Diagrama de clase para el patrón Proxy

Participantes

Proxy

Mantiene una referencia que permite al proxy acceder al objeto real. El proxy puede referirse a un Sujeto en caso de que las interfaces del SujetoReal y Sujeto sean la misma.

Proporciona una interfaz idéntica a la de Sujeto, de manera que un proxy pueda ser sustituido por el sujeto real.

Controla el acceso al sujeto real, y puede ser responsable de su creación y borrado.

Otras responsabilidades dependen del tipo proxy:

Los ***proxies remotos*** son responsables de codificar una petición y sus argumentos para enviar la petición codificada al sujeto real que se encuentra en un espacio de direcciones diferentes.

Los ***proxies virtuales*** pueden guardar información adicional sobre el sujeto real, por lo que pueden retardar el acceso al mismo.

Los ***proxies de protección*** comprueban que el llamador tenga los permisos de acceso necesarios para realizar una petición.

Sujeto. Define la interfaz común para el SujetoReal y el Proxy, de modo que pueda usarse un Proxy en cualquier sitio en el que se espere un SujetoReal.

SujetoReal. Define el objeto real representado.

Colaboraciones

El Proxy redirige peticiones al SujetoReal cuando sea necesario, dependiendo de tipo de proxy.

Consecuencias

El patrón Proxy introduce un nivel de indirección al acceder a un objeto. Esta indirección adicional tiene muchos posibles usos, dependiendo del tipo proxy:

Un proxy remoto puede ocultar el hecho de que un objeto reside en un espacio de direcciones diferente.

Un proxy virtual puede llevar a cabo optimizaciones tales como crear un objeto por encargo.

Tanto los proxies de protección, como las referencias inteligentes, permiten realizar tareas de mantenimiento adicionales cuando se accede a un objeto.

Hay otra optimización que el patrón Proxy puede ocultar al cliente. Se le conoce como copia-de-escritura, y está relacionada con la creación por encargo.

Patrones Relacionados

Adapter: un adaptador proporciona una interfaz diferente para el objeto que adapta. Por el contrario, un proxy tiene la misma interfaz que su sujeto. No obstante, un proxy utilizado para protección de acceso podría rechazar una operación que el sujeto sí realiza, de modo que su interfaz puede ser realmente un subconjunto de la del sujeto.

Decorador: si bien los decoradores pueden tener una implementación parecida a los proxies tienen un propósito diferente. Un decorador añade una o más responsabilidades a un objeto, mientras que un proxy controla el acceso a un objeto.

Los proxies difieren en el grado de similitud entre su implementación y la de un decorador. Un proxy de protección podría implementarse exactamente como un decorador. Por otro lado, un proxy remoto no contendrá una referencia directa a su sujeto real sino sólo una referencia indirecta, como "un ID de máquina y la dirección local en dicha máquina". Un proxy virtual empezará teniendo una referencia indirecta como un nombre de archivo, pero podrá al final obtener y utilizar una referencia directa.

Ejemplo A.12 Código de ejemplo para el patrón Proxy

Proxy.java

```
public class Proxy implements Sujeto{
     Sujeto sujetoReal = new SujetoReal();
     Integer numeroSaludos = 0;

     @Override
     public String saludar(String nombre) {
          numeroSaludos++;
          return sujetoReal.saludar(nombre);
     }

     public Integer getNumeroSaludos() {
          return numeroSaludos;
     }

}
```

Sujeto.java

```
public interface Sujeto {
     public String saludar(String nombre);
}
```

SujetoReal.java

```
public class SujetoReal implements Sujeto{

     @Override
     public String saludar(String nombre) {
          return "Hola "+nombre+"! Soy el Sujeto Real, no el Proxy.";
     }

}
```

Cliente.java

```
public class Cliente {

     public static void main(String[] args) {
          Sujeto sujetoProxy = new Proxy();

          System.out.println(sujetoProxy.saludar("Julio"));
          System.out.println(sujetoProxy.saludar("Jessica"));
          System.out.println(sujetoProxy.saludar("Pedro"));

          System.out.println("\nNumero de saludos realizados: "
                    +((Proxy)sujetoProxy).getNumeroSaludos());
     }

}
```

Salida en Consola

```
Hola Julio! Soy el Sujeto Real, no el Proxy.
Hola Jessica! Soy el Sujeto Real, no el Proxy.
Hola Pedro! Soy el Sujeto Real, no el Proxy.

Numero de saludos realizados: 3
```

Capítulo 4: Patrones de Comportamiento

4.1 Chain of Responsibility (Cadena de Responsabilidad)

Descripción

Evita acoplar el emisor de una petición a su receptor, dando a más de un objeto la posibilidad de responder a la petición. Encadena los objetos receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto.

Aplicabilidad

Úsese el patrón Chain of Responsibility cuando:

- Hay más de un objeto que pueden manejar una petición, y el manejador no se conoce a *priori*, sino que debería determinarse automáticamente.
- Se requiere enviar una petición a un objeto entre varios sin especificar explícitamente el receptor.
- El conjunto de objetos que pueden tratar una petición debería ser especificado dinámicamente.

Diagrama UML

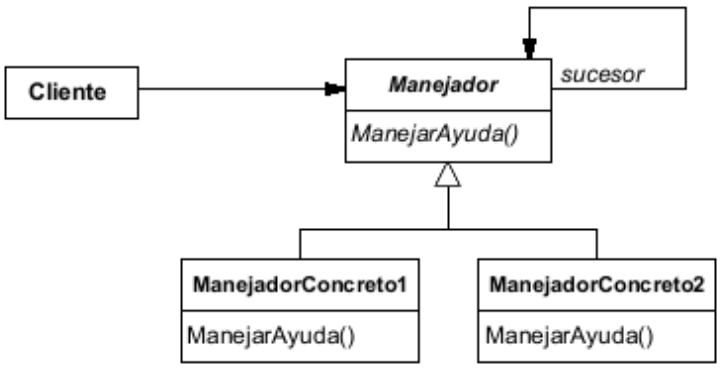

Figura B-24: Diagrama de clase para el patrón Chain of Responsability

Participantes

Manejador. Define una interfaz para tratar las peticiones. (Opcional) implementa el enlace al sucesor.

Manejador Concreto. Trata las peticiones de las que es responsable. Puede acceder a su sucesor. Si el ManejadorConcreto puede manejar la petición, lo hace; en caso contrario lo reenvía a su sucesor.

Cliente. Inicializa la petición a un objeto ManejadorConcreto de la cadena.

Colaboraciones

Cuando un cliente envía una petición, ésta se propaga a través de la cadena hasta que un objeto ManejadorConcreto se hace responsable de procesarla.

Consecuencias

Este patrón tiene las siguientes ventajas e inconvenientes:

Reduce el acoplamiento. El patrón libera a un objeto de tener que saber qué otro objeto maneja una petición. Un objeto sólo tiene que saber que una petición será manejada "de forma apropiada". Ni el receptor ni el emisor se conocen explícitamente entre ellos, y un objeto de la cadena tampoco tiene que conocer la estructura de ésta.

Añade flexibilidad para asignar responsabilidades a objetos. La Cadena de Responsabilidad ofrece una flexibilidad añadida para repartir responsabilidades entre objetos. Se pueden añadir o cambiar responsabilidades para tratar una petición modificando la cadena de tiempo de ejecución. Esto se puede combinar con la herencia para especializar los manejadores estáticamente.

No se garantiza la recepción. Dado que las peticiones no tienen un receptor explícito, no hay *garantía* de que sean manejadas (la petición puede alcanzar el final de la cadena sin haber sido procesada). Una petición también puede quedar sin tratar cuando la cadena no está configurada correctamente.

Patrones Relacionados

Este patrón se suele aplicar conjuntamente con el patrón Composite. En él, los padres de los componentes pueden actuar como sucesores.

Ejemplo A.13 Código de ejemplo para el patrón Chain of Responsability

Ayuda.java

```
public class Ayuda {
     private int cantidadAyudaNecesitada;
     private String nombre;

     public Ayuda(String nombre, int cantidadAyuda){
          this.nombre = nombre;
          this.cantidadAyudaNecesitada = cantidadAyuda;
     }

     public int getCantidadAyudaNecesitada() {
          return cantidadAyudaNecesitada;
     }

     public String getNombre() {
          return nombre;
     }
}
```

Manejador.java

```
public abstract class Manejador {
     protected Manejador sucesor;

     public void setSucesor(Manejador sucesor) {
          this.sucesor = sucesor;
     }

     public abstract void manejarAyuda(Ayuda ayudaRequerida);
}
```

ManejadorConcreto1.java

```
public class ManejadorConcreto1 extends Manejador{
     private final int MAXIMA_CANTIDAD_AYUDA = 100;

     @Override
     public void manejarAyuda(Ayuda ayudaRequerida) {
          if(ayudaRequerida.getCantidadAyudaNecesitada() > MAXIMA_CANTIDAD_AYUDA){
               sucesor.manejarAyuda(ayudaRequerida);
          }else{
               System.out.println("ManejadorConcreto1, maneja la ayuda de
"+ayudaRequerida.getNombre());
          }
     }
}
```

ManejadorConcreto2.java

```
public class ManejadorConcreto2 extends Manejador{
     private final int MAXIMA_CANTIDAD_AYUDA = 200;

     @Override
     public void manejarAyuda(Ayuda ayudaRequerida) {
          if(ayudaRequerida.getCantidadAyudaNecesitada() > MAXIMA_CANTIDAD_AYUDA){
               sucesor.manejarAyuda(ayudaRequerida);
          }else{
               System.out.println("ManejadorConcreto2, maneja la ayuda de
"+ayudaRequerida.getNombre());
          }
     }
}
```

ManejadorConcreto3.java

```
public class ManejadorConcreto3 extends Manejador{

     @Override
     public void manejarAyuda(Ayuda ayudaRequerida) {
          System.out.println("ManejadorConcreto3, maneja la ayuda de
"+ayudaRequerida.getNombre());
     }
}
```

Cliente.java

```
public class Cliente {

     public static void main(String[] args) {
          Manejador manejador1 = new ManejadorConcreto1();
          Manejador manejador2 = new ManejadorConcreto2();
          Manejador manejador3 = new ManejadorConcreto3();

          manejador1.setSucesor(manejador2);
          manejador2.setSucesor(manejador3);

          manejador1.manejarAyuda(new Ayuda("tornado", 65));
          manejador1.manejarAyuda(new Ayuda("heridos", 199));
          manejador1.manejarAyuda(new Ayuda("terremoto", 2000));
          manejador1.manejarAyuda(new Ayuda("maremoto", 101));
          manejador1.manejarAyuda(new Ayuda("otro desastre natural", 123));
     }

}
```

Salida en Consola

```
ManejadorConcreto1, maneja la ayuda de tornado
ManejadorConcreto2, maneja la ayuda de heridos
ManejadorConcreto3, maneja la ayuda de terremoto
ManejadorConcreto2, maneja la ayuda de maremoto
ManejadorConcreto2, maneja la ayuda de otro desastre natural
```

4.2 Command (Orden)

También conocido como: Action (Acción), Transaction (Transacción)

Descripción

Este patrón permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación, ni el receptor real de la misma. Para ello se encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de las peticiones, y poder deshacer las operaciones.

Aplicabilidad

Úsese el patrón Command cuando se quiera:

- Parametrizar objetos con una acción a realizar, como ocurría con los objetos ElementoDeMenu anteriores. En un lenguaje de procedimiento se puede expresar dicha parametrización con una función ***callback***, es decir con una función que está registrada en algún sitio para que sea llamada más tarde. Los objetos Orden son un sustituto orientado a objetos para las funciones callback.
- Especificar, poner en cola y ejecutar peticiones en diferentes instantes de tiempo. Un objeto Orden puede tener un tiempo de vida independiente de la petición original. Si se puede representar el receptor de una petición en una forma independiente del espacio de direcciones, entonces se puede transferir un objeto orden con la petición a un proceso diferente y llevar a cabo la petición allí.
- Permitir deshacer. La operación Ejecutar de Orden puede guardar en la propia orden el estado que anule sus efectos. Debe añadirse a la interfaz Orden una operación Deshacer que anule los efectos de una llamada anterior a Ejecutar. Las órdenes ejecutadas se guardan en una lista que hace las veces de historial. Se puede lograr niveles ilimitados de deshacer y repetir recorriendo dicha lista hacia atrás y hacia delante llamando respectivamente a Deshacer y Ejecutar.
- Permitir registrar los cambios de manera que se puedan volver a aplicar en caso de una caída del sistema. Aumentando la interfaz Orden con operaciones para cargar y guardar se puede mantener un registro persistente de los cambios. Recuperarse de una caída implica volver a cargar desde el disco las órdenes guardadas y volver a ejecutarlas con la operación Ejecutar.

- Estructurar un sistema alrededor de operaciones de alto nivel construidas sobre operaciones básicas. Dicha estructura es común en los sistemas de información que permiten **transacciones.**

Diagrama UML

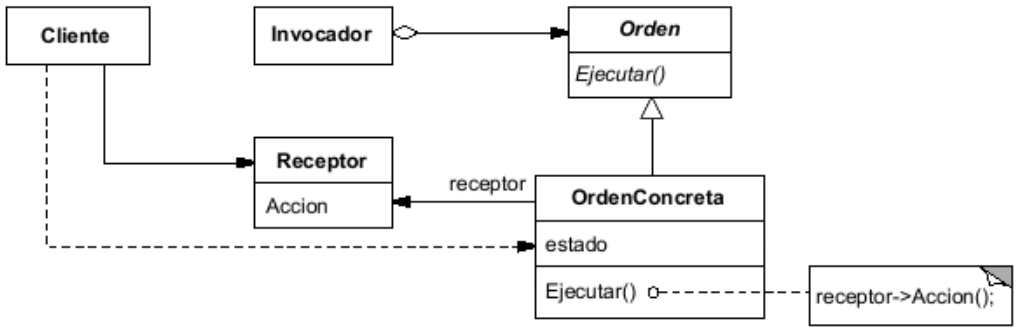

Figura B-25: Diagrama de clase para el patrón Command

Participantes

Orden. Declara una interfaz para ejecutar una operación.

Orden concreta. Define un enlace entre un objeto Receptor y una acción. Implementa Ejecutar invocando la correspondiente operación y operaciones del Receptor.

Cliente. Crea un objeto OrdenConcreta y establece un receptor.

Invocador. Le pide a la orden que ejecute la petición.

Receptor. Sabe cómo llevar a cabo las operaciones asociadas a una petición. Cualquier clase puede hacer actuar como Receptor.

Colaboraciones

El cliente crea un objeto OrdenConcreta y especifica su receptor.

Un objeto Invocador almacena el objeto OrdenConcreta.

El invocador envía una petición llamando a Ejecutar sobre la orden. Cuando las órdenes se pueden deshacer, OrdenConcreta guarda el estado para deshacer la orden antes de llamar a Ejecutar.

El objeto OrdenConcreta invoca operaciones de su receptor para llevar a cabo la petición.

El siguiente diagrama (figura B-26) muestra las interacciones entre estos objetos, ilustrando cómo Orden desacopla el invocador del receptor (y de la petición que éste lleva a cabo).

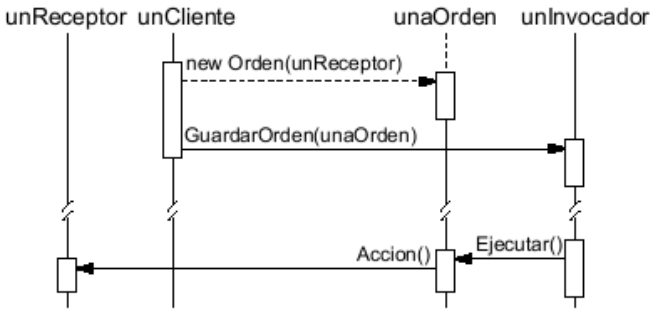

Figura B-26: Diagrama de secuencia para el patrón Command

Consecuencias

El patrón Command tiene las siguientes consecuencias:

Orden desacopla el objeto que invoca la operación de aquél que sabe cómo realizarla.

Las órdenes son objetos de primera clase. Pueden ser manipulados y extendidos como cualquier objeto.

Se pueden ensamblar órdenes en una orden compuesta. En general las órdenes compuestas son una instancia del patrón Composite.

Es fácil añadir nuevas órdenes, ya que no hay que cambiar las clases existentes.

Patrones Relacionados

Un Memento puede mantener el estado que necesitan las órdenes para anular sus efectos.

Una orden que debe ser copiada antes de ser guardada en el historial funciona como un Prototipo.

Ejemplo A.14 Código de ejemplo para el patrón Command

Invocador.java

```
public class Invocador {
     private Orden correr;
     private Orden caminar;

     public Invocador(Orden correr, Orden caminar){
          this.correr = correr;
          this.caminar = caminar;
     }

     public void correr(){
          correr.ejecutar();
     }

     public void caminar(){
          caminar.ejecutar();
     }
}
```

Orden.java

```
public interface Orden {
     public void ejecutar();
}
```

OrdenConcreta1.java

```
public class OrdenConcreta1 implements Orden{
     private Receptor receptor;

     public OrdenConcreta1(Receptor receptor){
          this.receptor = receptor;
     }

     @Override
     public void ejecutar() {
          receptor.accionCaminar();
     }
}
```

OrdenConcreta2.java

```
public class OrdenConcreta1 implements Orden{
     private Receptor receptor;

     public OrdenConcreta1(Receptor receptor){
          this.receptor = receptor;
     }

     @Override
     public void ejecutar() {
          receptor.accionCaminar();
     }
}
```

Receptor.java

```
public class Receptor {

     public void accionCaminar() {
          System.out.println("El receptor esta Caminando");
     }
```

```
     public void accionCorrer() {
          System.out.println("El receptor esta Corriendo");
     }
}
```

Cliente.java

```
public class Cliente {

     public static void main(String[] args) {
          Receptor receptor = new Receptor();
          Orden caminar = new OrdenConcreta1(receptor);
          Orden correr = new OrdenConcreta2(receptor);
          Invocador invocador = new Invocador(correr, caminar);

          invocador.caminar();
          invocador.correr();
     }
}
```

Salida en Consola

```
El receptor esta Caminando
El receptor esta Corriendo
```

4.3 Interpreter (Intérprete)

Descripción

El patrón Interpreter define un lenguaje, y una representación de su gramática junto con un intérprete que usa dicha representación para interpretar sentencias del lenguaje.

Se usa para definir un lenguaje para representar expresiones regulares que representen cadenas a buscar dentro de otras cadenas. Además, en general, para definir un lenguaje que permita representar las distintas instancias de una familia de problemas.

Aplicabilidad

Se usa el patrón Interpreter cuando hay un lenguaje que interpretar. El patrón Interpreter funciona mejor cuando:

- La gramática es simple. Para gramáticas complejas, la jerarquía de clases de la gramática se vuelve grande e inmanejable. Herramientas como los generadores analizadores sintácticos constituyen una alternativa mejor en estos casos. Éstas pueden interpretar expresiones sin necesidad de construir árboles sintácticos abstractos, lo que puede ahorrar espacio y, posiblemente, tiempo.
- La eficiencia no es una preocupación crítica. Los intérpretes más eficientes normalmente no se implementan interpretando árboles de análisis sintáctico directamente, sino que primero los traducen a algún otro formato. Por ejemplo, las expresiones regulares suelen transformarse en máquinas de estados. Pero incluso en ese caso, el *traductor* puede implementarse con el patrón Interpreter, de modo que éste sigue siendo aplicable.

Diagrama UML

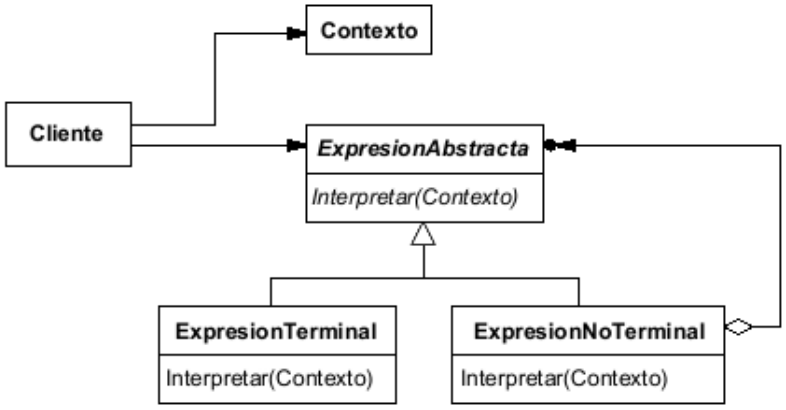

Figura B-27: Diagrama de clase para el patrón Interpreter

Participantes

ExpresionAbstracta

Declara una operación abstracta Interpretar que es común a todos los nodos del árbol de sintaxis abstracto.

ExpresionTerminal

Implementa una operación Interpretar asociada con los símbolos terminales de la gramática.

Se necesita una instancia de esta clase para cada símbolo terminal de una sentencia.

ExpresionNoTerminal

Por cada regla de la gramática $R::=R_1R_2\ldots R_n$ debe haber una de estas clases.

Mantiene variables de instancia de tipo ExpresionAbstracta para cada uno de los símbolos de R_1 a R_n.

Implementa una operación Interpretar para los símbolos no terminales de la gramática. Interpretar normalmente se llama a sí misma recursivamente sobre las variables que representan de R_1 a R_n.

Contexto

Contiene información que es global al intérprete.

Cliente

Construye (o recibe) un árbol sintáctico abstracto que representa una determinada sentencia del lenguaje definido por la gramática. Este árbol sintáctico abstracto está formado por instancias de las clases ExpresionNoTerminal y ExpresionTerminal.

Invoca a la operación Interpretar.

Colaboraciones

El cliente construye (o recibe) la sentencia como un árbol sintáctico abstracto formado por instancias de ExpresionNoTerminal y ExpresionTerminal. A continuación el cliente inicializa el contexto e invoca a la operación Interpretar.

Cada nodo ExpresionNoTerminal define Interpretar en términos de Interpretar de cada subexpresión. La operación Interpretar de cada ExpresionTerminal define el caso base de la recursión.

Las operaciones Interpretar de cada nodo usan el contexto para almacenar y acceder al estado del intérprete.

Consecuencias

El patrón Interpreter tiene las siguientes ventajas e inconvenientes:

Es fácil cambiar y ampliar la gramática. Puesto que el patrón usa clases para representar las reglas de la gramática, se puede usar la herencia para cambiar o extender ésta. Se puede modificar incrementalmente las expresiones existentes, y se pueden definir otras nuevas como variaciones de las antiguas.

También resulta fácil implementar la gramática. Las clases que definen los nodos del árbol sintáctico abstracto tienen implementaciones similares.

Las gramáticas complejas son difíciles de mantener. El patrón Interpreter define al menos una clase para cada regla de la gramática. De ahí que las gramáticas que contienen muchas reglas pueden ser difíciles de controlar y mantener. Se pueden aplicar otros patrones de diseño para mitigar el problema. Pero cuando la gramática es muy compleja son más adecuadas otras técnicas como los generadores de analizadores sintácticos o de compiladores.

Añadir nuevos modos de interpretar expresiones. El patrón Interpreter facilita evaluar una expresión o realizar una comprobación de tipos en ella definiendo una nueva operación en las clases de las

expresiones. Si se va a seguir añadiendo nuevos modos de interpretar una expresión, se debiese considerar la utilización del patrón Visitor para evitar cambiar las clases de la gramática.

Patrones Relacionados

Composite: el árbol sintáctico abstracto es una instancia del patrón Composite.

El patrón Flyweight muestra cómo compartir símbolos terminales dentro del árbol sintáctico abstracto.

Iterator: el intérprete puede usar un Iterator para recorrer la estructura.

Puede usarse el patrón Visitor para mantener el comportamiento de cada nodo del árbol sintáctico abstracto en una clase.

Ejemplo A.15 Código de ejemplo para el patrón Interpreter

ExpresionAbstracta.java

```
import java.util.HashMap;

public interface ExpresionAbstracta {
     public int interpretar(HashMap<String,Integer> contexto);
}
```

ExpresionNoTerminal.java

```
import java.util.HashMap;

public class ExpresionNoTerminal implements ExpresionAbstracta{
     private ExpresionAbstracta primerSimbolo;
    private ExpresionAbstracta segundoSimbolo;

    public ExpresionNoTerminal(ExpresionAbstracta primerSimbolo, ExpresionAbstracta
segundoSimbolo) {
     this.primerSimbolo = primerSimbolo;
     this.segundoSimbolo = segundoSimbolo;
    }

    public int interpretar(HashMap<String,Integer> contexto){
        return primerSimbolo.interpretar(contexto) + segundoSimbolo.interpretar(contexto);
    }
}
```

ExpresionTerminal.java

```
import java.util.HashMap;

public class ExpresionTerminal implements ExpresionAbstracta{
     private String nombre;
    public ExpresionTerminal(String nombre)       { this.nombre = nombre; }
    public int interpretar(HashMap<String,Integer> contexto)  {
        if(null==contexto.get(nombre)) return 0;
        return contexto.get(nombre).intValue();
    }
}
```

Cliente.java

```
import java.util.HashMap;
import java.util.Stack;

public class Cliente {

     public static void main(String[] args) {

         HashMap<String,Integer> contexto = new HashMap<String,Integer>();
         contexto.put("-", 5);
         contexto.put(".", 1);

         System.out.println("Interpretador de numeros Mayas," +
               "\n para los simbolos \"-\" = 5 y \".\" = 1 \n");

         String expresion = "--....";
         ExpresionAbstracta arbolSintactico = contruyeArbolSintactico(expresion);
         int resultado = arbolSintactico.interpretar(contexto);
         System.out.println("* La expresion maya: \""+expresion+"\", equivale a "+resultado);

     }

     private static ExpresionAbstracta contruyeArbolSintactico(String Expresion) {
         Stack<ExpresionAbstracta> ExpresionStack = new Stack<ExpresionAbstracta>();
         for (String token : Expresion.split("")) {
          if(!ExpresionStack.empty()){
               ExpresionAbstracta subExpresion =
                    new ExpresionNoTerminal(ExpresionStack.pop(), new
ExpresionTerminal(token));
                ExpresionStack.push( subExpresion );
           }else{
               ExpresionStack.push( new ExpresionTerminal(token) );
           }
         }
         return ExpresionStack.pop();
     }

}
```

Salida en Consola

```
Interpretador de numeros Mayas,
 para los simbolos "-" = 5 y "." = 1

* La expresion maya: "--....", equivale a 14
```

4.4 Iterator (Iterador)

También conocido como: Cursor

Descripción

El patrón Iterator permite a un objeto cliente acceder al contenido de un contenedor de una manera secuencial, sin tener ningún conocimiento acerca de la representación interna de su contenido.

El término contenedor, utilizado anteriormente, se puede definir como una colección de datos u objetos. Los objetos dentro del contenedor pueden a su vez ser colecciones, haciendo también así una colección de colecciones. El patrón Iterator permite a un objeto cliente recorrer esta colección de objetos, sin tener que exponer su representación interna.

Para lograr esto, el patrón Iterator sugiere que un objeto contenedor tiene que estar diseñado para proporcionar una interfaz pública en la forma de un objeto Iterador para que diferentes objetos cliente accedan a su contenido. Un objeto Iterator contiene métodos públicos que permiten al objeto cliente navegar por la lista de los objetos dentro del contenedor.

Aplicabilidad

Se usa el patrón Iterator:

- Para acceder al contenido de un objeto agregado sin exponer su representación interna.
- Para permitir varios recorridos sobre objetos agregados.
- Para proporcionar una interfaz uniforme para recorrer diferentes estructuras agregadas (es decir, para permitir la iteración polimórfica).

Diagrama UML

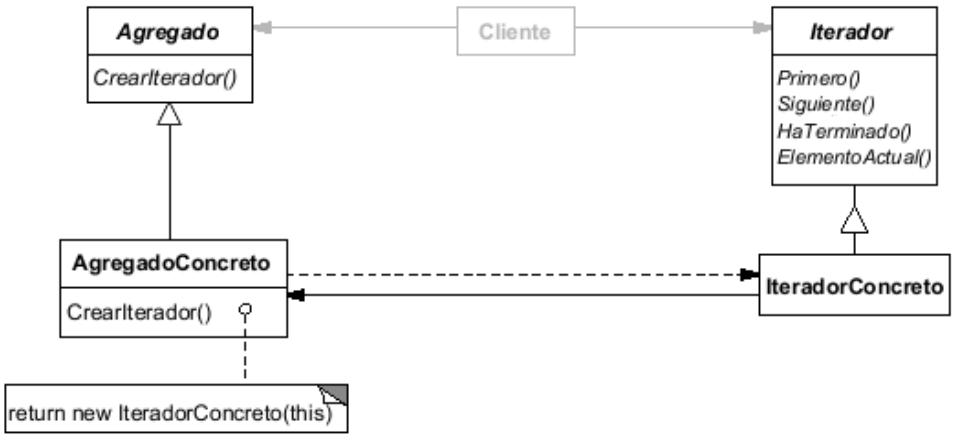

Figura B-28: Diagrama de clase para el patrón Iterator

Participantes

Iterador. Define una interfaz para recorrer los elementos y acceder a ellos.

IteradorConcreto. Implementa la interfaz Iterador. Mantiene la posición actual en el recorrido del agregado.

Agregado. Define una interfaz para crear un objeto Iterador.

AgregadoConcreto. Implementa la interfaz de creación de Iterador para devolver una instancia del IteradorConcreto apropiado.

Colaboraciones

Un IteradorConcreto sabe cuál es el objeto actual del agregado y puede calcular el objeto siguiente en el recorrido.

Consecuencias

El patrón Iterador tiene tres consecuencias importantes:

Permite variaciones en el recorrido de un agregado. Los agregados complejos pueden recorrerse de muchas formas. Por ejemplo, la generación de código y comprobación de tipos implican recorrer árboles de análisis sintáctico. La generación de código puede recorrer dicho árbol de análisis sintáctico en-orden o en pre-orden. Los iteradores facilitan cambiar el algoritmo de recorrido: basta con sustituir la instancia de iterador por otra diferente. También se pueden definir subclases de Iterator para permitir nuevos recorridos.

Los iteradores simplifican la interfaz Agregado. La interfaz de recorrido de Iterador elimina la necesidad de una interfaz parecida en Agregado, simplificando así la interfaz del agregado.

Se puede hacer más de un recorrido a la vez sobre un agregado. Un iterador mantiene su propio estado del recorrido. Por tanto, es posible estar realizando más de un recorrido al mismo tiempo.

Patrones Relacionados

Composite: los iteradores suelen aplicarse a estructuras recursivas como los compuestos.

Factory Method: los iteradores polimórficos se basan en métodos de fabricación para crear instancias de las subclases apropiadas de Iterator.

El patrón Memento suele usarse conjuntamente con el patrón Iterator. Un iterador puede usar un memento para representar el estado de una iteración. El iterador almacena el memento internamente.

Ejemplo A.16 Código de ejemplo para el patrón Iterator

Agregado.java

```
public interface Agregado {
     Iterador crearIterador();
}
```

AgregadoConcreto.java

```
public class AgregadoConcreto implements Agregado{
     private String [] elementos;

     @Override
     public Iterador crearIterador() {
          return new IteradorConcreto(elementos);
     }

     public void agregarElemento(String elemento){
          if(elementos == null){
               elementos = new String [1];
               elementos[0]=elemento;
          }else{
               int tamano = elementos.length;
               String [] nuevosElementos = new String [tamano+1];
               nuevosElementos[tamano]=elemento;
               for(int i=0; i<elementos.length; i++){
                    nuevosElementos[i]= elementos[i];
               }
               elementos = nuevosElementos;
          }
     }
}
```

Iterador.java

```
public interface Iterador {

    String primero();
    String siguiente();
    boolean haTerminado();
```

```
    String elementoActual();
}
```

IteradorConcreto.java

```
public class IteradorConcreto implements Iterador{
     private String [] elementos;
     private int elementosRecorridos = 0;

     public IteradorConcreto(String [] elementos){
          this.elementos = elementos;
     }

     @Override
     public String elementoActual() {
          return elementos[elementosRecorridos-1];
     }

     @Override
     public boolean haTerminado() {
          return (elementos.length>elementosRecorridos)?false:true;
     }

     @Override
     public String primero() {
          return elementos[0];
     }

     @Override
     public String siguiente() {
          elementosRecorridos++;
          return elementos[elementosRecorridos-1];
     }

}
```

Cliente.java

```
public class Cliente {
     public static void main(String[] args) {
          AgregadoConcreto agregado = new AgregadoConcreto();
          agregado.agregarElemento("primero");
          agregado.agregarElemento("segundo");
          agregado.agregarElemento("tercero");
          agregado.agregarElemento("cuarto");
          agregado.agregarElemento("quinto");

          Iterador iterador = agregado.crearIterador();

          while(!iterador.haTerminado()){
               System.out.println("iterando: "+iterador.siguiente());
          }
          System.out.println("Elemento primero: "+iterador.primero());
          System.out.println("Elemento actual: "+iterador.elementoActual());

     }
}
```

Salida en Consola

```
iterando: primero
iterando: segundo
iterando: tercero
iterando: cuarto
iterando: quinto
Elemento primero: primero
Elemento actual: quinto
```

4.5 Mediator (Mediador)

Descripción

Define un objeto que encapsula cómo interactúan una serie de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.

Cuando muchos objetos interactúan con otros objetos, se puede formar una estructura muy compleja, con objetos con muchas conexiones con otros objetos. En un caso extremo cada objeto puede conocer a todos los demás objetos. Para evitar esto el patrón Mediator encapsula el comportamiento de todo un conjunto de objetos en un solo objeto.

Aplicabilidad

Úsese el patrón Mediador cuando:

- Un conjunto de objetos se comunican de forma bien definida, pero compleja. Las interdependencias resultantes no están estructuradas y son difíciles de comprender.
- Es difícil reutilizar un objeto, ya que éste se refiere a otros muchos objetos, con los que se comunica.
- Un comportamiento que está distribuido entre varias clases debería poder ser adaptado sin necesidad de una gran cantidad de subclases.

Diagrama UML

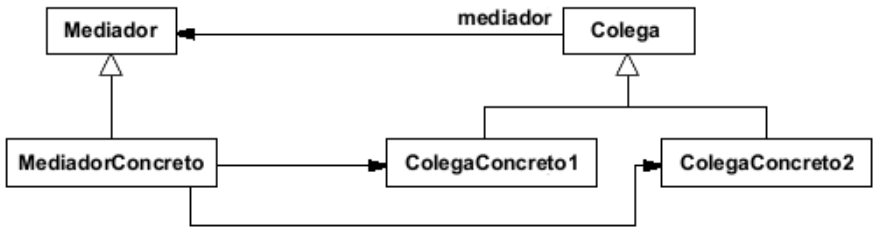

Figura B-29: Diagrama de clase para el patrón Mediator

Participantes

Mediador. Define una interfaz para comunicarse con sus objetos Colega.

MediadorConcreto. Implementa el comportamiento cooperativo coordinando objetos Colega. Conoce a sus Colegas.

Clases Colega. Cada clase Colega conoce a su objeto Mediador. Cada Colega se comunica con su mediador cada vez que, de no existir éste, se hubiera comunicado con otro Colega.

Colaboraciones

Los colegas envían y reciben peticiones a través de un Mediador. El mediador implementa el comportamiento cooperativo encaminando estas peticiones a los Colegas apropiados.

Consecuencias

El patrón Mediador tiene las siguientes ventajas e inconvenientes:

Reduce la herencia. Un mediador localiza el comportamiento que de otra manera estaría distribuido en varios objetos. Para cambiar este comportamiento sólo es necesario crear una subclase del Mediador; las clases Colega pueden ser reutilizadas tal cual.

Desacopla a los Colegas. Un mediador promueve un bajo acoplamiento entre Colegas, las clases Colega pueden usarse y modificarse de forma independiente.

Simplifica los protocolos de los objetos. Un mediador sustituye interacciones muchos-a-muchos por interacciones uno-a-muchos entre el mediador y sus Colegas. Las relaciones uno-a-muchos son más fáciles de comprender, mantener y extender.

Abstrae cómo cooperan los objetos. Hacer de la mediación un concepto independiente y encapsularla en un objeto permite centrarse en cómo interactúan los objetos en vez de un su comportamiento individual. Eso ayuda a clarificar cómo interactúan los objetos en vez de en su comportamiento individual. Eso ayuda a clarificar cómo interactúan los objetos de un sistema.

Centraliza el control. El patrón Mediador cambia complejidad de interacción por complejidad en el mediador. Dado que encapsula protocolos, puede hacerse más complejo que cualquier Colega individual, esto puede hacer del mediador un monolito difícil de mantener.

Patrones Relacionados

El patrón Facade difiere del Mediador en que abstrae un subsistema de objetos para proporcionar una interfaz más conveniente. Su protocolo es unidireccional; es decir, los objetos Fachada hacen peticiones a las clases del subsistema pero no a la inversa. Por el contrario, el patrón Mediador permite un comportamiento cooperativo que no es proporcionado por los objetos Colegas y el protocolo es multidireccional.

Los Colegas pueden comunicarse con el mediador usando el patrón Observer.

Ejemplo A.17 Código de ejemplo para el patrón Mediator

Colega.java

```
public abstract class Colega {
     protected Mediador mediador;

     public Colega(Mediador mediador){
          this.mediador = mediador;
     }
}
```

ColegaConcreto1.java

```
public class ColegaConcreto1 extends Colega{

     public ColegaConcreto1(Mediador mediador) {
          super(mediador);
     }

     public void enviar(String mensaje){
          mediador.enviar(mensaje, this);
     }

     public void notificar(String mensaje){
          System.out.println("ColegaConcreto1 recibe mensaje: " + mensaje);
     }

}
```

ColegaConcreto2.java

```
public class ColegaConcreto2 extends Colega{

     public ColegaConcreto2(Mediador mediador) {
          super(mediador);
     }

     public void enviar(String mensaje){
          mediador.enviar(mensaje, this);
     }

     public void notificar(String mensaje){
          System.out.println("ColegaConcreto2 recibe mensaje: " + mensaje);
     }

}
```

Mediador.java

```
public interface Mediador {
     public void enviar(String mensaje, Colega colega);
}
```

MediadorConcreto.java

```
public class MediadorConcreto implements Mediador{
     private ColegaConcreto1 colega1;
     private ColegaConcreto2 colega2;

     @Override
     public void enviar(String mensaje, Colega colega) {
          if(colega.equals(colega1)){
               colega2.notificar(mensaje);
          }else{
               colega1.notificar(mensaje);
          }
     }
```

```
     public void setColega1(ColegaConcreto1 colega1) {
          this.colega1 = colega1;
     }

     public void setColega2(ColegaConcreto2 colega2) {
          this.colega2 = colega2;
     }

}
```

Cliente.java

```
public class Cliente {
     public static void main(String[] args) {
          MediadorConcreto mediador = new MediadorConcreto();

          ColegaConcreto1 colega1 = new ColegaConcreto1(mediador);
          ColegaConcreto2 colega2 = new ColegaConcreto2(mediador);

          mediador.setColega1(colega1);
          mediador.setColega2(colega2);

          colega1.enviar("Hola como estas?");
          colega2.enviar("Muy bien, y tu?");
          colega1.enviar("Feliz");
     }
}
```

Salida en Consola

```
ColegaConcreto2 recibe mensaje: Hola como estas?
ColegaConcreto1 recibe mensaje: Muy bien, y tu?
ColegaConcreto2 recibe mensaje: Feliz
```

4.6 Memento (Recuerdo)

También conocido como: Token

Descripción

A veces es necesario guardar el estado interno de un objeto. Esto se necesita cuando se implementan casillas de verificación o mecanismos tipo operación deshacer que permitan al usuario anular operaciones temporales y recuperarse de los errores. Debe guardarse información del estado en algún sitio para que los objetos puedan volver a su estado anterior. Los objetos encapsulan normalmente parte de su estado, o todo, haciéndolo inaccesible a otros objetos e imposible de guardar externamente. Exponer este estado violaría la encapsulación, comprometiendo así la fiabilidad y extensibilidad de la implementación.

El patrón Memento se encarga de representar y externar el estado interno de un objeto sin violar la encapsulación, de forma que el objeto puede volver al estado anterior más tarde.

Aplicabilidad

Úsese el patrón Memento cuando se cumpla todo lo siguiente:

- Hay que guardar una instantánea del estado de un objeto (o de parte de éste) para que pueda volver posteriormente a ese estado.
- Una interfaz directa para obtener el estado exponga detalles de implementación y rompa la encapsulación del objeto.

Diagrama UML

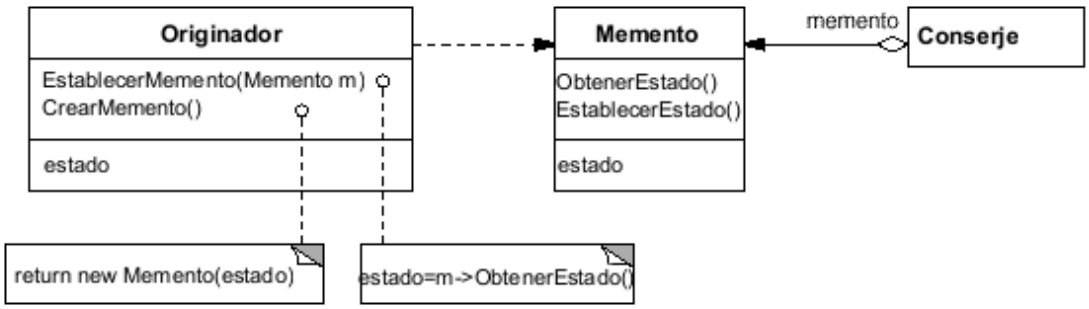

Figura B-30: Diagrama de clase para el patrón Memento

Participantes

Memento

Guarda el estado interno del objeto Creador. El memento puede guardar tanta información del estado interno del creador como sea necesario a discreción del creador.

Protege frente a accesos de otros objetos que no sean el creador. Los mementos tienen realmente dos interfaces. El Conserje ve una interfaz *reducida* del Memento (sólo puede pasar el memento a otros objetos). El Creador, por el contrario, ve una interfaz amplia, que le permite acceder a todos los datos necesarios para volver a su estado anterior. Idealmente, sólo el creador que produjo el memento estaría autorizado a acceder al estado interno de éste.

Creador (Originador)

Crea un memento que contiene una instantánea de su estado interno actual.

Usa el memento para volver a su estado anterior.

Conserje

Es responsable de guardar en lugar seguro el memento.

Nunca examina los contenidos del memento, ni opera sobre ellos.

Colaboraciones

Un conserje solicita un memento a un creador (originador), lo almacena durante un tiempo y se lo devuelve a su creador, tal y como se ilustra en el siguiente diagrama de secuencia (figura B-31):

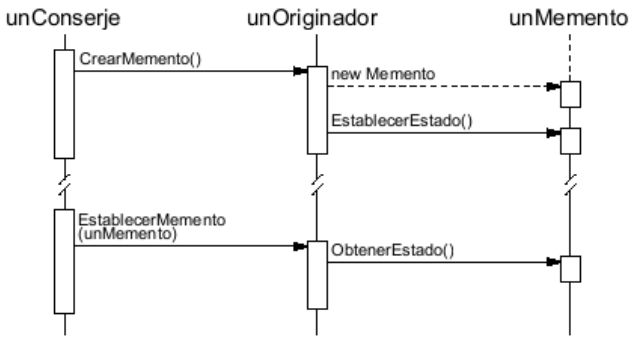

Figura B-31: Diagrama de secuencia para el patrón Memento

A veces el conserje no devolverá el memento a su creador, ya que el creador podría no necesitar nunca volver a un estado anterior.

Los mementos son pasivos. Sólo el creador que creó el memento asignará o recuperará su estado.

Consecuencias

El patrón Memento tiene varias consecuencias:

Preservación de los límites de la encapsulación. El memento evita exponer información que sólo debería ser gestionada por un creador, pero que sin embargo debe ser guardada fuera del creador. El patrón oculta a otros objetos las interioridades, potencialmente complejas, del Creador, preservando así los límites de la encapsulación.

Simplifica al Creador. En otros diseños que persiguen conservar la encapsulación, el Creador mantiene las versiones de su estado interno que han sido solicitadas por los clientes. Eso asigna toda la responsabilidad de gestión del almacenamiento al Creador. Que sean los clientes quienes gestionen el estado que solicita simplificar al Creador y evita que los clientes tengan que notificar a los creadores cuándo ha acabado.

El uso de mementos puede ser costoso. Los mementos podrían producir un coste considerable si el Creador debe copiar grandes cantidades de información para aguardarlas en el memento o si los clientes crean y devuelven mementos a su creador con mucha frecuencia. A menos que encapsular y restablecer el estado del Creador sea poco costoso, el patrón podría no ser apropiado.

Definición de interfaces reducidas y amplias. En algunos lenguajes puede ser difícil garantizar que sólo el creador acceda al estado del memento.

Costes ocultos en el cuidado de los mementos. Un conserje es responsable de borrar los mementos que custodia. Sin embargo, el conserje no sabe cuánto han estado en un memento. De ahí que un conserje que debería ser ligero pueda provocar grandes costes de almacenamiento cuando debe guardar mementos.

Patrones Relacionados

Command: las órdenes pueden usar mementos para guardar el estado de las operaciones que pueden deshacerse.

Iterator: puede usar mementos para la iteración, tal y como se acaba de describir.

Ejemplo A.18 Código de ejemplo para el patrón Memento

Conserje.java

```
import java.util.ArrayList;

public class Conserje {
     private ArrayList<Memento> estadosGuardados = new ArrayList<Memento>();

     public void guardaMemento(Memento memento) { estadosGuardados.add(memento); }

     public Memento getUltimoMemento() {
          if(estadosGuardados.size()>0){
               int indice = estadosGuardados.size()-1;
               Memento mementoActual = estadosGuardados.get(indice);
               estadosGuardados.remove(indice);
               return mementoActual;
          }
          return null;
     }
}
```

Memento.java

```
public class Memento {
     private String estado;

     public Memento(String estado){
          this.estado = estado;
     }
     public void establecerEstado(String estado){
          this.estado = estado;
     }
     public String obtenerEstado() {
          return estado;
     }
}
```

Originador.java

```
public class Originador {
     private String estado;

     public void setEstado(String estado) {
          System.out.println("Originador: Estableciendo estado a "+estado);
          this.estado = estado;
     }

     public Memento crearMemento() {
          System.out.println("Originador: crea y guarda nuevo Memento con estado "+estado);
          return new Memento(estado);
     }

     public void establecerMemento(Memento memento) {
          estado = memento.obtenerEstado();
          System.out.println("Originador: Reestableciendo estado, ahora el estado es:
"+estado);
     }
}
```

Cliente.java

```
public class Cliente {
     public static void main(String[] args) {
          Conserje conserje = new Conserje();
          Originador originador = new Originador();
```

```
        originador.setEstado("Estado1");
        originador.setEstado("Estado2");
        conserje.guardaMemento( originador.crearMemento() );
        originador.setEstado("Estado3");
        conserje.guardaMemento( originador.crearMemento() );
        originador.setEstado("Estado4");
        originador.establecerMemento( conserje.getUltimoMemento() );
        originador.establecerMemento( conserje.getUltimoMemento() );
    }
}
```

Salida en Consola

```
Originador: Estableciendo estado a Estado1
Originador: Estableciendo estado a Estado2
Originador: crea y guarda nuevo Memento con estado Estado2
Originador: Estableciendo estado a Estado3
Originador: crea y guarda nuevo Memento con estado Estado3
Originador: Estableciendo estado a Estado4
Originador: Reestableciendo estado, ahora el estado es: Estado3
Originador: Reestableciendo estado, ahora el estado es: Estado2
```

4.7 Observer (Observador)

También conocido como: Spider, Dependents (Dependientes), Publish-Subscribe (Publicar-Suscribir)

Descripción

El patrón Observer especifica una dependencia del tipo uno-a-muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, el observador se encarga de notificar y actualizar automáticamente este cambio a todos los objetos que dependen de él.

El objetivo de este patrón es desacoplar la clase de los objetos clientes del objeto, aumentando la modularidad del lenguaje, así como evitar bucles de actualización.

Aplicabilidad

Úsese el patrón Observer en cualquiera de las situaciones siguientes:

- Cuando una abstracción tiene dos aspectos y uno depende del otro. Encapsular estos aspectos en objetos separados permite modificarlos y reutilizarlos de forma independiente.
- Cuando un cambio en un objeto requiere cambiar otros, y no sabemos cuántos objetos necesitan cambiarse.
- Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quiénes son dichos objetos. En otras palabras, cuando no se desea que estos objetos estén fuertemente acoplados.

Diagrama UML

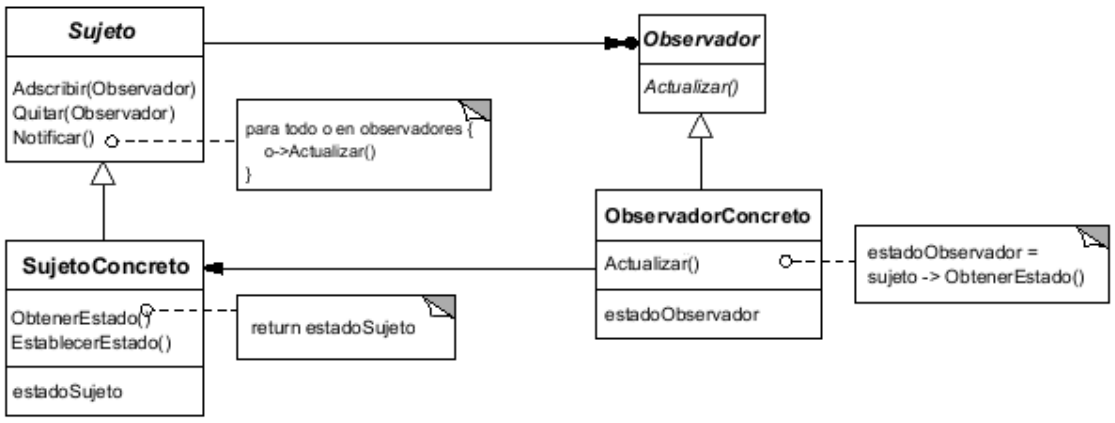

Figura B-32: Diagrama de clase para el patrón Observer

Participantes

Sujeto

Conoce a sus observadores. Un sujeto puede ser observado por cualquier número de objetos Observador.

Proporciona una interfaz para asignar y quitar objetos Observador.

Observador

Define una interfaz para actualizar los objetos que deben ser notificados ante cambios en un sujeto.

SujetoConcreto

Almacena el estado de interés para los objetos ObservadorConcreto.

Envía una notificación a sus observadores cuando cambia su estado.

ObservadorConcreto

Mantiene una referencia a un objeto SujetoConcreto.

Guarda un estado que debería ser consistente con el del sujeto.

Implementa la interfaz de actualización del Observador para mantener su estado consistente con el del sujeto.

Colaboraciones

SujetoConcreto notifica a sus observadores cada vez que se produce un cambio que pudiera hacer que el estado de éstos fuera inconsistente con el suyo.

Después de ser informado de un cambio en el sujeto concreto, un objeto ObservadorConcreto puede pedirle al sujeto más información. ObservadorConcreto usa esta información para sincronizar su estado con el del sujeto.

El siguiente diagrama de secuencia (figura B-33) muestra las colaboraciones entre un sujeto y dos observadores:

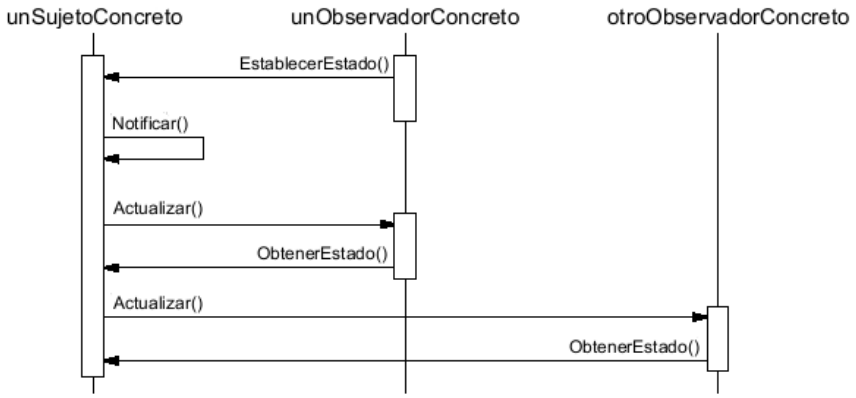

Figura B-33: Diagrama de secuencia para el patrón Observer

Consecuencias

El patrón Observador permite modificar los sujetos observadores de forma independiente. Es posible reutilizar objetos sin reutilizar sus observadores, y viceversa. Esto permite añadir observadores sin modificar el sujeto u otros observadores.

Patrones Relacionados

Mediador: encapsulando semánticas de actualizaciones complejas, puede actuar como mediador entre sujetos y observadores.

Singleton: también se puede usar el patrón Singleton para que sea único y globalmente accesible.

Ejemplo A.19 Código de ejemplo para el patrón Observer

Observador.java

```
public interface Observador {
     public void actualizar();
     public String getNombreObservador();
}
```

ObservadorConcreto.java

```
public class ObservadorConcreto implements Observador {
     private String estadoObservador;
     private SujetoConcreto sujeto;
     private String nombreObservador;

     public ObservadorConcreto(SujetoConcreto sujeto, String nombre){
          this.sujeto = sujeto;
          this.nombreObservador = nombre;
     }
     @Override
     public void actualizar() {
          estadoObservador = sujeto.obtenerEstado();
          System.out.println("El estado del observador '"+nombreObservador+"' es:
"+estadoObservador);
     }
     @Override
```

```
     public String getNombreObservador() {
          return nombreObservador;
     }

}
```

Sujeto.java

```
import java.util.ArrayList;
import java.util.List;

public abstract class Sujeto {
     private List<Observador> observadores = new ArrayList<Observador>();

     public void adscribir(Observador observador){
          observadores.add(observador);
     }

     public void quitar(Observador observador){
          System.out.println("Elimino observador: "+observador.getNombreObservador());
          observadores.remove(observador);
     }

     public void notificar(){
          System.out.println("> Notificando:");
          for (Observador observador : observadores) {
               observador.actualizar();
          }
     }
}
```

SujetoConcreto.java

```
public class SujetoConcreto extends Sujeto{
     private String estadoSujeto;

     public String obtenerEstado(){
          return estadoSujeto;
     }

     public void establecerEstado(String estado){
          estadoSujeto = estado;
          this.notificar();
     }

}
```

Cliente.java

```
public class Cliente {
     public static void main(String[] args) {
          SujetoConcreto sujeto = new SujetoConcreto();
          ObservadorConcreto observador1 = new ObservadorConcreto(sujeto, "observador1");
          ObservadorConcreto observador2 = new ObservadorConcreto(sujeto, "observador2");

          sujeto.adscribir(observador1);
          sujeto.adscribir(observador2);

          sujeto.establecerEstado("Estado1");
          sujeto.establecerEstado("Estado2");
          sujeto.quitar(observador2);
          sujeto.establecerEstado("Estado3");
     }
}
```

Salida en Consola

```
> Notificando:
El estado del observador 'observador1' es: Estado1
El estado del observador 'observador2' es: Estado1
> Notificando:
El estado del observador 'observador1' es: Estado2
El estado del observador 'observador2' es: Estado2
Elimino observador: observador2
> Notificando:
El estado del observador 'observador1' es: Estado3
```

4.8 State (Estado)

También conocido como: Objects for States (Estados como Objetos)

Descripción

El estado de un objeto puede ser definido como su estado exacto en cualquier punto dado de tiempo, dependiendo de los valores de sus propiedades o atributos. El conjunto de los métodos implementados por una clase constituyen el comportamiento de sus instancias. Siempre que hay un cambio en los valores de sus atributos, se dice que el estado de un objeto ha cambiado.

Un simple ejemplo de esto sería el caso de un usuario seleccionando un estilo de fuente o color específicos en un editor HTML. Cuando un usuario selecciona un estilo de letra o color diferente, las propiedades del objeto editor cambian. Esto puede ser considerado como un cambio en su estado interno.

El patrón State es útil porque permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.

Aplicabilidad

Úsese el patrón State en cualquiera de los siguientes dos casos:

- El comportamiento de un objeto depende de su estado, y debe cambiar en tiempo de ejecución dependiendo de ese estado.
- Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. Este estado se suele representar por una o más constantes enumeradas. Muchas veces son varias las operaciones que contienen esta misma estructura condicional. El patrón State pone cada rama de la condición en una clase aparte. Esto permite tratar al estado del objeto como un objeto de pleno derecho que puede variar independientemente de otros objetos.

Diagrama UML

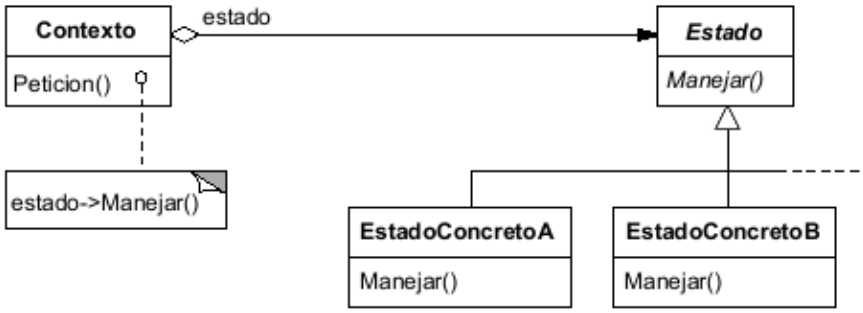

Figura B-34: Diagrama de clase para el patrón State

Participantes

Contexto. Define la interfaz de interés para los clientes y mantiene una instancia de una subclase de EstadoConcreto que define el estado actual.

Estado. Define una interfaz para encapsular el comportamiento asociado con un determinado estado del Contexto.

Subclases EstadoConcreto. Cada subclase implementa un comportamiento asociado con un estado del Contexto.

Colaboraciones

Contexto delega las peticiones que dependen del estado en el objeto EstadoConcreto actual.

Un contexto puede pasarse a sí mismo como parámetro para que el objeto Estado maneje la petición. Esto permite al objeto Estado acceder al contexto si fuera necesario.

Contexto es la interfaz principal para los clientes. Los clientes pueden configurar un contexto con objetos Estado. Una vez que está configurado el contexto, sus clientes ya no tienen que tratar con los objetos Estado directamente.

Cualquiera de las subclases de Contexto o de EstadoConcreto pueden decidir qué estado sigue a otro y bajo qué circunstancias.

Consecuencias

El patrón State tiene las siguientes consecuencias:

Localiza el comportamiento dependiente del estado y divide dicho comportamiento en diferentes estados. El patrón State sitúa en un objeto todo el comportamiento asociado con un determinado estado. Como todo el código dependiente del estado reside en una subclase de Estado, pueden añadirse fácilmente nuevos estados y transiciones definiendo nuevas subclases. Una alternativa es usar valores de datos para definir los estados internos y hacer que las operaciones de Contexto comprueben dichos datos explícitamente. Pero en ese caso se tendrían sentencias condicionales repartidas por toda la implementación del Contexto. Añadir un nuevo estado podría requerir cambiar varias operaciones, complicando el mantenimiento. El patrón State evita este problema, pero puede introducir otro, al distribuir el comportamiento para los diferentes estados en varias subclases de Estado. Esto incrementa el número de clases y es menos compacto que una única clase. Pero dicha distribución es realmente buena si hay muchos estados, que de otro modo necesitarían grandes sentencias condicionales. Al igual que ocurre con los procedimientos largos, hay que tratar de evitar las grandes sentencias condicionales. Son monolíticas y tienden a hacer el código menos explícito, lo que a su vez las hace difíciles de modificar y extender. El patrón State ofrece un modo mejor de estructurar el código dependiente del estado. La lógica que determina las transiciones entre estados no reside en sentencias *if o switch* monolíticas, sino que se reparte entre las subclases de Estado. Al encapsular cada transición y acción en una clase se está elevando la idea de un estado de ejecución a objetos de estado en toda regla. Esto impone una estructura al código y hace que su intención sea más clara.

Hace explícitas las transiciones entre estados. Cuando un objeto define su estado actual únicamente en términos de valores de datos internos, sus transiciones entre estados carecen de una representación explícita; sólo aparecen como asignaciones a determinadas variables. Introducir objetos separados para los diferentes estados hace que las transiciones sean más explícitas. Además, los objetos Estado pueden proteger al Contexto frente a estados internos inconsistentes, ya que las transiciones entre estados son atómicas desde una perspectiva del Contexto (tienen lugar cambiando una variable, no varias).

Los objetos Estado pueden compartirse. En caso de que los objetos Estado no tengan variables (es decir, si el estado que representa está totalmente representado por su tipo) entonces varios

contextos pueden compartir un mismo objeto Estado. Cuando se comparten los estados de este modo, son en esencia pesos ligeros que no tienen estado intrínseco, sino sólo comportamiento.

Patrones Relacionados

El patrón Flyweight explica cuándo y cómo compartir objetos Estado.

Los objetos Estado muchas veces son Singletons.

Ejemplo A.20 Código de ejemplo para el patrón State

Contexto.java

```
public class Contexto {
     private Estado estado;

     public Contexto(Estado estado){
          this.estado = estado;
     }

     public void peticion(){
          estado.manejar(this);
     }

     public void setEstado(Estado estado) {
          this.estado = estado;
     }
}
```

Estado.java

```
public interface Estado {
     public void manejar(Contexto contexto);
     public String getNombreEstado();
}
```

EstadoConcretoA.java

```
public class EstadoConcretoA implements Estado{
     private String nombreEstado = "On";

     @Override
     public void manejar(Contexto contexto) {
          System.out.println("realizamos una tarea especifica del EstadoConcretoA");
          contexto.setEstado(new EstadoConcretoB());
     }

     @Override
     public String getNombreEstado() {
          return nombreEstado;
     }

}
```

EstadoConcretoB.java

```
public class EstadoConcretoB implements Estado{
     private String nombreEstado = "Off";

     @Override
     public void manejar(Contexto contexto) {
          System.out.println("realizamos una tarea especifica del EstadoConcretoB");
          contexto.setEstado(new EstadoConcretoA());
     }
```

```
    @Override
    public String getNombreEstado() {
        return nombreEstado;
    }

}
```

Cliente.java

```
public class Cliente {
    public static void main(String[] args) {
        Contexto contexto = new Contexto(new EstadoConcretoA());

        contexto.peticion();
        contexto.peticion();
        contexto.peticion();
    }
}
```

Salida en Consola

```
realizamos una tarea especifica del EstadoConcretoA
realizamos una tarea especifica del EstadoConcretoB
realizamos una tarea especifica del EstadoConcretoA
```

4.9 Strategy (Estrategia)

También conocido como: Policy (Política)

Descripción

El patrón Strategy permite mantener un conjunto de algoritmos de los que el objeto cliente puede elegir aquel que le conviene e intercambiarlo según sus necesidades.

Los distintos algoritmos se encapsulan y el cliente trabaja contra un objeto contexto. Como se ha dicho, el cliente puede elegir el algoritmo que prefiera de entre los disponibles o puede ser el mismo objeto contexto el que elija el más apropiado para cada situación.

Cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es candidato a utilizar el patrón Strategy. Puede haber cualquier número de estrategias y cualquiera de ellas podrá ser intercambiada por otra en cualquier momento, incluso en tiempo de ejecución.

Aplicabilidad

Úsese el patrón Strategy cuando:

- Muchas clases relacionadas difieren sólo en su comportamiento. Las estrategias permiten configurar una clase con un determinado comportamiento de entre muchos posibles.
- Se necesitan distintas variantes de un algoritmo. Por ejemplo, se podrían definir algoritmos que reflejasen distintas soluciones de compromiso entre tiempo y espacio. Pueden usarse estrategias cuando estas variantes se implementan como una jerarquía de clases de algoritmos.
- Un algoritmo usa datos que los clientes no deberían conocer. Se usa el patrón Strategy para evitar exponer estructuras de datos complejas y dependientes del algoritmo.
- Una clase define muchos comportamientos, y éstos se representan como múltiples sentencias condicionales en sus operaciones. En vez de tener muchos condicionales, uno puede mover las ramas de éstos a su propia clase de Estrategia.

Diagrama UML

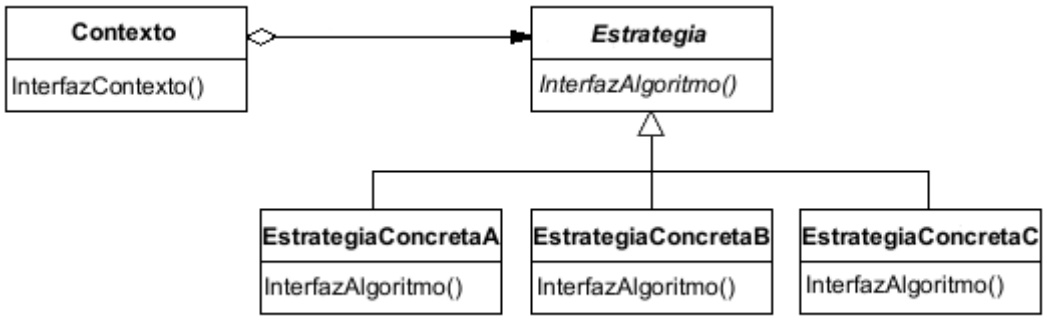

Figura B-35: Diagrama de clase para el patrón Strategy

Participantes

Estrategia

Declara una interfaz común a todos los algoritmos permitidos. El Contexto usa esta interfaz para llamar al algoritmo definido por una EstrategiaConcreta.

EstrategiaConcreta

Implementa el algoritmo usando la interfaz Estrategia.

Contexto

Se configura con un objeto EstrategiaConcreta.

Mantiene una referencia a un objeto Estrategia.

Puede definir una interfaz que permita a la Estrategia acceder a sus datos.

Colaboraciones

Estrategia y Contexto interactúan para implementar el algoritmo elegido. Un contexto puede pasar a la estrategia todos los datos requeridos por el algoritmo cada vez que se llama a éste. Otra alternativa es que el contexto se pase a sí mismo como argumento de las operaciones de Estrategia. Eso permite a la estrategia hacer llamadas al contexto cuando sea necesario.

Un contexto redirige peticiones de los clientes a su estrategia. Los clientes normalmente crean un objeto EstrategiaConcreta, al cual pasan el contexto; por tanto, los clientes interactúan exclusivamente con el contexto. Suele haber una familia de clases EstrategiaConcreta a elegir por el cliente.

Consecuencias

El patrón Strategy presenta las siguientes ventajas e inconvenientes:

Familias de algoritmos relacionados. Las jerarquías de clases Estrategia definen una familia de algoritmos o comportamientos para ser reutilizados por los contextos. La herencia puede ayudar a sacar factor común de la funcionalidad de estos algoritmos.

Una alternativa a la herencia. La herencia ofrece otra forma de permitir una variedad de algoritmos o comportamientos. Pero esto liga el comportamiento al Contexto, mezclando la implementación de algoritmos con la del Contexto, lo que hace que éste sea más difícil de comprender, mantener y extender. Y no se puede modificar el algoritmo dinámicamente. Se terminará teniendo muchas clases relacionadas, cuya única diferencia es el algoritmo o comportamiento que utilizan. Encapsular el algoritmo en clases Estrategia separadas no permite variar el algoritmo independientemente de su contexto, haciéndolo más difícil de cambiar, comprender y extender.

Las estrategias eliminan las sentencias condicionales. El patrón Strategy ofrece una alternativa a las sentencias condicionales para seleccionar el comportamiento deseado. Cuando se juntan muchos comportamientos en una clase es difícil no usar sentencias condicionales para seleccionar el comportamiento correcto. Encapsular el comportamiento en clases Estrategia separadas elimina estas sentencias condicionales.

Una elección de implementación. Las estrategias pueden proporcionar distintas implementaciones del mismo comportamiento. El cliente puede elegir entre estrategias con diferentes soluciones de compromiso entre tiempo y espacio.

Los clientes deben conocer las diferentes Estrategias. El patrón tiene el inconveniente potencial de que un cliente debe comprender cómo difieren las Estrategias antes de seleccionar la adecuada. Los clientes pueden estar expuestos a cuestiones de implementación. Por tanto, el patrón Strategy debería usarse sólo cuando la variación de comportamiento sea relevante a los clientes.

Costes de comunicación entre Estrategias y Contexto. La interfaz de Estrategia es compartida por todas las clases EstrategiaConcreta, ya sea el algoritmo que implementa sea trivial o complejo. Por tanto, es probable que algunos objetos EstrategiaConcreta no usen toda la información que reciben a través de dicha interfaz; las estrategias concretas simples pueden incluso no utilizar nada de dicha información. Eso significa que habrá veces en las que el contexto crea e inicializa parámetros que nunca se usan. Si esto puede ser un problema, se necesitará un acoplamiento más fuerte entre Estrategia y Contexto.

Mayor número de objetos. Las estrategias aumentan el número de objetos de una aplicación. A veces se puede reducir este coste implementando las estrategias como objetos sin estado que puedan ser compartidos por el contexto. El contexto mantiene cualquier estado residual, pasándolo en cada petición al objeto Estrategia. Las estrategias compartidas no deberían mantener el estado entre invocaciones. El patrón Flyweight describe este enfoque en más detalle.

Patrones Relacionados

Flyweight: los objetos Estrategia suelen ser buenos pesos ligeros.

Ejemplo A.21 Código de ejemplo para el patrón Strategy

Contexto.java

```
public class Contexto {
     private Estrategia estrategia;

     public Contexto(Estrategia estrategia){
          this.estrategia = estrategia;
     }

     public void interfazContexto(){
          estrategia.interazAlgoritmo();
     }
}
```

Estrategia.java

```
public interface Estrategia {
     public void interazAlgoritmo();
}
```

EstrategiaConcretaA.java

```
public class EstrategiaConcretaA implements Estrategia{

     @Override
     public void interazAlgoritmo() {
          System.out.println("realizamos algoritmo de EstrategiaConcretaA");
     }

}
```

EstrategiaConcretaB.java

```
public class EstrategiaConcretaB implements Estrategia{

     @Override
     public void interazAlgoritmo() {
          System.out.println("realizamos algoritmo de EstrategiaConcretaB");
     }

}
```

EstrategiaConcretaC.java

```
public class EstrategiaConcretaC implements Estrategia{

     @Override
     public void interazAlgoritmo() {
          System.out.println("realizamos algoritmo de EstrategiaConcretaC");
```

```
    }

}
```

Cliente.java

```
public class Cliente {
    public static void main(String[] args) {
        Contexto contexto;

        contexto = new Contexto(new EstrategiaConcretaA());
        contexto.interfazContexto();

        contexto = new Contexto(new EstrategiaConcretaB());
        contexto.interfazContexto();

        contexto = new Contexto(new EstrategiaConcretaC());
        contexto.interfazContexto();
    }
}
```

Salida en Consola

```
realizamos algoritmo de EstrategiaConcretaA
realizamos algoritmo de EstrategiaConcretaB
realizamos algoritmo de EstrategiaConcretaC
```

4.10 Template Method (Método Plantilla)

Descripción

Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura. El patrón Template Method define una estructura de herencia en la cual la superclase sirve de plantilla de los métodos en las subclases. Una de las ventajas de este método es que evita la repetición de código, por consiguiente la aparición de errores.

Aplicabilidad

El patrón Template Method debería usarse:

- Para implementar las partes de un algoritmo que no cambian y dejar que sean las subclases quienes implementen el comportamiento que puede variar.
- Cuando el comportamiento repetido de varias subclases debería factorizarse y ser localizado en una clase común para evitar el código duplicado. Ésta es una buena idea de "refactorizar para generalizar", tal como la describen Opdyke y Johnson [OJ93]. En primer lugar se identifican las diferencias en el código existente y a continuación se separan dichas diferencias en nuevas operaciones. Por último, sustituimos el código que cambia por un método que llama a una de estas nuevas operaciones.
- Para controlar las extensiones de las subclases. Se puede definir un método plantilla que llame a operaciones "de enganche" en determinados puntos, permitiendo así las extensiones sólo en esos puntos.

Diagrama UML

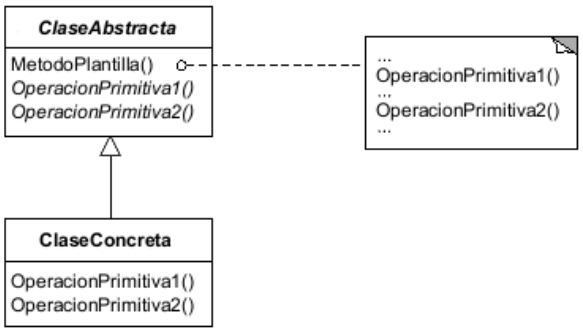

Figura B-36: Diagrama de clase para el patrón Template Method

Participantes

ClaseAbstracta

Define operaciones primitivas abstractas que son definidas por las subclases para implementar los pasos de un algoritmo.

Implementa un método plantilla que define el esqueleto de un algoritmo. El método plantilla llama a las operaciones primitivas así como a operaciones definidas en ClaseAbstracta o a las de otros objetos.

ClaseConcreta

Implementa las operaciones primitivas para realizar los pasos del algoritmo específicos de las subclases.

Colaboraciones

ClaseConcreta se basa en ClaseAbstracta para implementar los pasos de algoritmos que no cambian.

Consecuencias

Los métodos plantilla son una técnica fundamental de reutilización del código. Son particularmente importantes en las bibliotecas de clases, ya que son el modo de factorizar y extraer el comportamiento común de las clases de la biblioteca.

Los métodos plantilla llevan a una estructura de control invertido que a veces se denomina "el principio de Hollywood", es decir, "No nos llame, nosotros le llamaremos". Esto se refiere a cómo una clase padre llama a las operaciones de una subclase y no al revés.

Los métodos plantilla llaman a los siguientes tipos de operaciones:

- Operaciones concretas (ya sea de la ClaseConcreta o de las clases cliente)
- Operaciones concretas de Clase Abstracta (es decir, operaciones que suelen ser útiles para las subclases)
- Operaciones primitivas (es decir, operaciones abstractas)
- Métodos de fabricación
- Operaciones enganche, que proporcionan el comportamiento predeterminado que puede ser modificado por las subclases si es necesario. Una operación de enganche normalmente no hace nada por omisión.

Es importante que los métodos plantilla especifiquen que operaciones son enganches (que puedan ser redefinidas) y cuales son operaciones abstractas (que deben ser redefinidas). Para reutilizar una clase abstracta apropiadamente, los escritores de las subclases deben saber que operaciones están diseñadas para ser redefinidas.

Patrones Relacionados

Los métodos de Fabricación se llaman muchas veces desde métodos plantilla.

Strategy: los métodos plantilla usan la herencia para modificar una parte de un algoritmo.

Las estrategias usan delegación para variar el algoritmo completo.

Ejemplo A.22 Código de ejemplo para el patrón Template Method

ClaseAbstracta.java

```
public abstract class ClaseAbstracta {

     public void metodoPlantilla(){
          operacionPrimitiva1();
          operacionPrimitiva2();
     }

     public abstract void operacionPrimitiva1();
     public abstract void operacionPrimitiva2();
}
```

ClaseConcreta.java

```
public class ClaseConcreta extends ClaseAbstracta{

     @Override
     public void operacionPrimitiva1() {
          System.out.println("esta es la operacion 1 de la clase concreta");
     }

     @Override
     public void operacionPrimitiva2() {
          System.out.println("esta es la operacion 2 de la clase concreta");
     }

}
```

Cliente.java

```
public class Cliente {
     public static void main(String[] args) {
          ClaseAbstracta plantilla = new ClaseConcreta();
          plantilla.metodoPlantilla();
     }
}
```

Salida en Consola

```
esta es la operacion 1 de la clase concreta
esta es la operacion 2 de la clase concreta
```

4.11 Visitor (Visitante)

Descripción

Representa una operación que se realiza sobre los elementos que conforman la estructura de un objeto. El patrón Visitor permite definir nuevas operaciones sin cambiar las clases de los elementos en los que opera.

Cada método visitante de un visitador concreto puede ser pensado como un método que no es de una sola clase, sino de un par de clases: el visitador concreto y la clase elemento particular.

Aplicabilidad

Se usa el patrón Visitor cuando:

- Una estructura de objetos contiene muchas clases de objetos con diferentes interfaces, y se requiere realizar operaciones sobre los elementos que dependen de su clase concreta.
- Se necesitan realizar muchas operaciones distintas y no relacionadas sobre los objetos de una estructura de objetos, y se desea evitar "contaminar" sus clases con dichas operaciones. El patrón Visitor permite mantener juntas operaciones relacionadas definiéndolas en una clase. Cuando la estructura de objetos es compartida por varias aplicaciones, el patrón Visitor permite poner operaciones sólo en aquellas aplicaciones que las necesiten.
- Las clases que definen la estructura de objetos rara vez cambian, pero muchas veces se necesitan definir nuevas operaciones sobre la estructura. Cambiar las clases de la estructura de objetos requiere redefinir la interfaz para todos los visitantes, lo que es potencialmente costoso. Si las clases de la estructura cambian con frecuencia, probablemente sea mejor definir las operaciones en las propias clases.

Diagrama UML

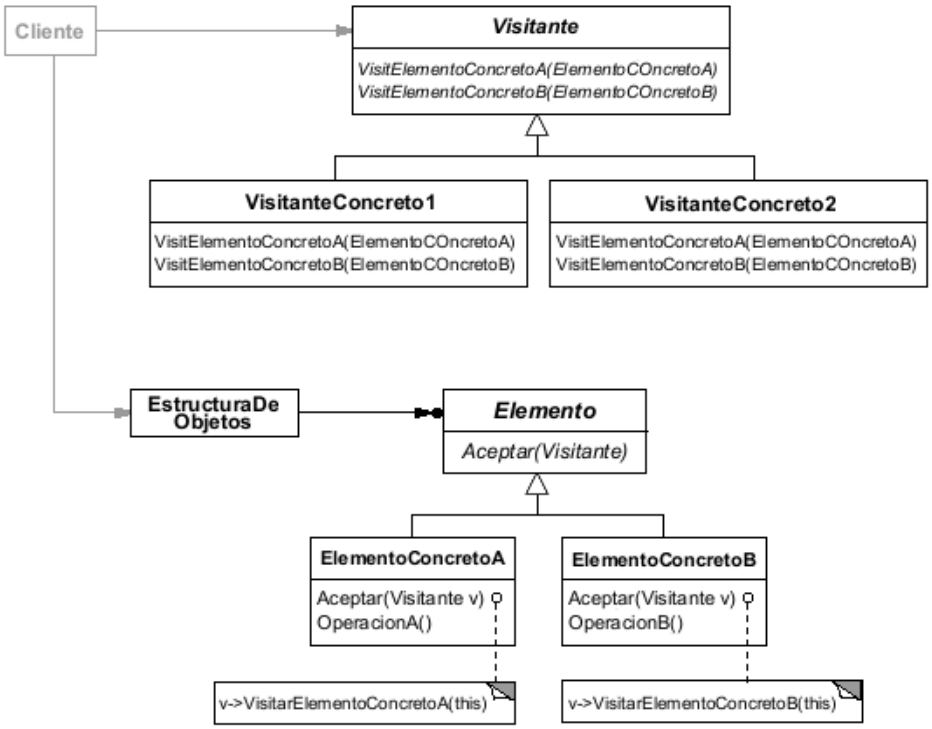

Figura B-37: Diagrama de clase para el patrón Visitor

Participantes

Visitante

Declara una operación Visitar para cada clase de operación ElementoConcreto de la estructura de objetos. El nombre y signatura de la operación identifican a la clase que envía la petición Visitar al visitante. Eso permite al visitante determinar la clase concreta de elemento que está siendo visitada. A continuación el visitante puede acceder al elemento directamente a través de su interfaz particular.

VisitanteConcreto

Implementa cada operación declarada por Visitante. Cada operación implementa un fragmento del algoritmo definido para la clase correspondiente de la estructura. VisitanteConcreto proporciona el contexto para el algoritmo y guarda su estado local. Muchas veces este estado acumula resultados durante el recorrido de la estructura.

Elemento

Define una operación Aceptar que toma un visitante como argumento.

ElementoConcreto

Implementa una operación Aceptar que toma un visitante como argumento.

EstructuraDeObjetos

Puede enumerar sus elementos.

Puede proporcionar una interfaz de alto nivel para permitir al visitante visitar a sus elementos.

Puede ser un compuesto (Composite) o una colección, como una lista o un conjunto.

Colaboraciones

Un cliente que usa el patrón Visitor debe crear un objeto VisitanteConcreto y a continuación recorrer la estructura, visitando cada objeto con el visitante.

Cada vez que se visita a un elemento, éste llama a la operación del Visitante que corresponde con su clase. El elemento se pasa a sí mismo como argumento de la operación para permitir al visitante acceder a su estado, en caso de que sea necesario.

El siguiente diagrama de secuencia (figura B-38) ilustra las colaboraciones entre una estructura de objetos, un visitante y dos elementos:

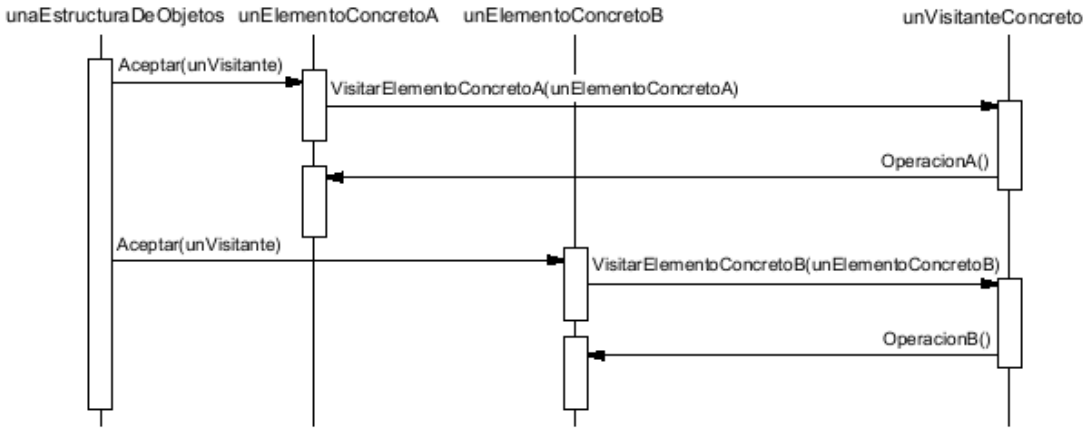

Figura B-38: Diagrama de secuencia para el patrón Visitor

Consecuencias

Algunas de las ventajas e inconvenientes del patrón Visitor son las siguientes:

El visitante facilita añadir nuevas operaciones. Los visitantes facilitan añadir nuevas operaciones que dependen de los componentes de objetos complejos. Se puede definir una nueva operación sobre una estructura simplemente añadiendo un nuevo visitante. Si, por el contrario, se extendiese la funcionalidad sobre muchas clases, habría que cambiar cada clase para definir una nueva operación.

Un visitante agrupa operaciones relacionadas y separa las que no lo están. El comportamiento similar no está esparcido por clases que definen la estructura de objetos; está localizado en un visitante. Las partes de comportamiento no relacionadas se dividen en las propias subclases del visitante. Esto simplifica tanto las clases que definen los elementos como los algoritmos definidos por los visitantes. Cualquier estructura de datos específica de un algoritmo puede estar oculta en el visitante.

Es difícil añadir nuevas clases de ElementoConcreto. El patrón Visitor hace que sea complicado añadir nuevas subclases de Elemento. Cada ElementoConcreto nuevo da lugar a una nueva operación abstracta del Visitante y a su correspondiente implementación en cada clase VisitanteConcreto. A veces se puede proporcionar en Visitante una implementación predeterminada que puede ser heredada por la mayoría de los visitantes concretos, pero esto representa una excepción más que una regla.

Por tanto la cuestión fundamental a considerar a la hora de aplicar el patrón Visitor es si es más probable que cambie el algoritmo aplicado sobre una estructura de objetos o las clases de los objetos que componen la estructura. La jerarquía de clases de Visitante puede ser difícil de mantener cuando se añaden nuevas clases de ElementoConcreto con frecuencia. En tales casos, es probablemente más fácil definir las operaciones en las clases que componen la estructura. Si la jerarquía de clases de Elemento es estable pero se está continuamente añadiendo operaciones o cambiando algoritmos, el patrón Visitor ayudará a controlar dichos cambios.

Visitar varias jerarquías de clases. Un iterador puede visitar a los objetos de una estructura llamando a sus operaciones a medida que los recorre. Pero un iterador no puede trabajar en varias estructuras de objetos con distintos tipos de elementos.

Acumular el estado. Los visitantes pueden acumular estado a medida que van visitando cada elemento de la estructura de objetos. Sin un visitante, este estado se pasaría como argumentos extra a las operaciones que realizan el recorrido, o quizá como variables globales.

Romper la encapsulación. El enfoque del patrón Visitor asume que la interfaz de ElementoConcreto es lo bastante potente como para que los visitantes hagan su trabajo. Como resultado, el patrón suele obligarnos a proporcionar operaciones públicas que accedan al estado interno de un elemento, lo que puede comprometer su encapsulación.

Patrones Relacionados

Composite: los visitantes pueden usarse para aplicar una operación sobre una estructura de objetos definida por el patrón Composite.

Interpreter: se puede aplicar el patrón Visitor para llevar a cabo la interpretación.

Ejemplo A.23 Código de ejemplo para el patrón Visitor

Elemento.java

```
public interface Elemento {

     public void Aceptar(Visitante visitante);

}
```

ElementoConcretoA.java

```
public class ElementoConcretoA implements Elemento{

     @Override
     public void Aceptar(Visitante visitante) {
          visitante.VisitaElementoConcretoA(this);
     }

     public void OperationA(){
          System.out.println("Operacion A");
     }

}
```

ElementoConcretoB.java

```
public class ElementoConcretoB implements Elemento{

     @Override
     public void Aceptar(Visitante visitante) {
          visitante.VisitaElementoConcretoB(this);
     }

     public void OperationB(){
          System.out.println("Operacion B");
     }

}
```

EstructuraDeObjetos.java

```
import java.util.ArrayList;

public class EstructuraDeObjetos {

     private ArrayList<Elemento> elementos = new ArrayList<Elemento>();

     public void Agregar(Elemento elemento){
```

```
            elementos.add(elemento);
      }

      public void Quitar(Elemento elemento){
            elementos.remove(elemento);
      }

      public void Aceptar(Visitante visitante){
            for(Elemento e:elementos){
                  e.Aceptar(visitante);
            }
      }

}
```

Visitante.java

```
public interface Visitante {

      public void VisitaElementoConcretoA(ElementoConcretoA elementoConcretoA);
      public void VisitaElementoConcretoB(ElementoConcretoB elementoConcretoB);

}
```

VisitanteConcreto1.java

```
public class VisitanteConcreto1 implements Visitante{

      @Override
      public void VisitaElementoConcretoA(ElementoConcretoA elementoConcretoA) {
            System.out.println(elementoConcretoA.getClass().getSimpleName()
                        + " visitado por " + this.getClass().getSimpleName() );
      }

      @Override
      public void VisitaElementoConcretoB(ElementoConcretoB elementoConcretoB) {
            System.out.println(elementoConcretoB.getClass().getSimpleName()
                        + " visitado por " + this.getClass().getSimpleName() );
      }

}
```

VisitanteConcreto2.java

```
public class VisitanteConcreto2 implements Visitante{

      @Override
      public void VisitaElementoConcretoA(ElementoConcretoA elementoConcretoA) {
            System.out.println(elementoConcretoA.getClass().getSimpleName()
                        + " visitado por " + this.getClass().getSimpleName() );
      }

      @Override
      public void VisitaElementoConcretoB(ElementoConcretoB elementoConcretoB) {
            System.out.println(elementoConcretoB.getClass().getSimpleName()
                        + " visitado por " + this.getClass().getSimpleName() );
      }
}
```

Cliente.java

```
public class Cliente {
      public static void main(String[] args) {
            EstructuraDeObjetos estructuraElementos = new EstructuraDeObjetos();
            estructuraElementos.Agregar(new ElementoConcretoA());
            estructuraElementos.Agregar(new ElementoConcretoB());

            Visitante v1 = new VisitanteConcreto1();
            Visitante v2 = new VisitanteConcreto2();
```

```
            estructuraElementos.Aceptar(v1);
            estructuraElementos.Aceptar(v2);
        }
}
```

Salida en Consola

```
ElementoConcretoA visitado por VisitanteConcreto1
ElementoConcretoB visitado por VisitanteConcreto1
ElementoConcretoA visitado por VisitanteConcreto2
ElementoConcretoB visitado por VisitanteConcreto2
```

Conclusiones

Se ha comprendido cómo los patrones de diseño para el desarrollo de software orientado a objetos facilitan el manejo de los sistemas abstractos existentes ya que gran parte de los sistemas orientados a objetos usan patrones de diseño.

Como la mayor parte de los sistemas orientados a objetos complejos utilizan uno o varios patrones de diseño. Cuando se comienza a aprender programación orientada a objetos, en ocasiones nos quejamos por la dificultad de seguir el flujo de control o la forma en que se usa la herencia, pero muchas veces se debe a que no se comprenden los patrones que utiliza el sistema.

También se sabe que los patrones de diseño pueden convertirnos en mejores diseñadores. Cuando se trabaja con sistemas grandes, sin darnos cuenta, la mayor parte de las veces aprendemos estos patrones por nosotros mismos, por lo que aprender los patrones de diseño ayuda a un novato a comportarse como un experto.

Se entiende que los patrones de diseño fueron pensados para promover el buen diseño y son especialmente útiles a la hora de pasar de un modelo de análisis a un modelo de implementación. Aunque un diseño flexible y reutilizable contendrá objetos que no están en el modelo de análisis, muchas veces deben ser rediseñados para hacerlos más reutilizables.

El catálogo de patrones que se muestra en este trabajo puede sin lugar a duda afectar de varias formas el modo en que se diseña software orientado a objetos a medida que basamos nuestra experiencia en ellos.

Aprendimos que los patrones de diseño ayudan a determinar cómo reorganizar un diseño. Es conocido que a medida que el código fuente del software crece y se expande, su evolución es gobernada por dos necesidades contradictorias: el software debe satisfacer más requisitos y el software debe ser más reutilizable. Para seguir evolucionando, el software debe ser reorganizado en un proceso conocido como refactorización. Estos patrones de diseño reflejan muchas de las estructuras que resultan de la refactorización y usando éstos patrones de diseño en las fases tempranas del diseño del sistema se previenen posteriores refactorizaciones.

Los mejores diseños usarán muchos patrones de diseño que se mezclan entre sí. Es mi deseo que el presente trabajo cumpla entre otras cosas, el mejorar la forma en que se realiza el diseño orientado a objetos por parte de los desarrolladores de software mediante el uso de los patrones de diseño para producir mejores sistemas.

Anexos

Anexo A. Índice de código fuente

Ejemplo A.1 Código de ejemplo para el patrón Abstract Factory 23
Ejemplo A.2 Código de ejemplo para el patrón Builder 29
Ejemplo A.3 Código de ejemplo para el patrón Factory Method 33
Ejemplo A.4 Código de ejemplo para el patrón Prototype 36
Ejemplo A.5 Código de ejemplo para el patrón Singleton 39
Ejemplo A.6 Código de ejemplo para el patrón Adapter 43
Ejemplo A.7 Código de ejemplo para el patrón Bridge 47
Ejemplo A.8 Código de ejemplo para el patrón Composite 51
Ejemplo A.9 Código de ejemplo para el patrón Decorator 57
Ejemplo A.10 Código de ejemplo para el patrón Facade 61
Ejemplo A.11 Código de ejemplo para el patrón Flyweight 66
Ejemplo A.12 Código de ejemplo para el patrón Proxy 70
Ejemplo A.13 Código de ejemplo para el patrón Chain of Responsability 74
Ejemplo A.14 Código de ejemplo para el patrón Command 79
Ejemplo A.15 Código de ejemplo para el patrón Interpreter 84
Ejemplo A.16 Código de ejemplo para el patrón Iterator 88
Ejemplo A.17 Código de ejemplo para el patrón Mediator 92
Ejemplo A.18 Código de ejemplo para el patrón Memento 97
Ejemplo A.19 Código de ejemplo para el patrón Observer 101
Ejemplo A.20 Código de ejemplo para el patrón State 107
Ejemplo A.21 Código de ejemplo para el patrón Strategy 112
Ejemplo A.22 Código de ejemplo para el patrón Template Method 116
Ejemplo A.23 Código de ejemplo para el patrón Visitor 121

Anexo B. Índice de figuras (Diagramas UML)

Figura B-1: Notación UML 12
Figura B-2: Jerarquía de los diagramas UML 2.0 13
Figura B-3: Diagrama de clases 14
Figura B-4: Diagrama de objetos 15
Figura B-5: Diagrama de casos de uso 16
Figura B-6: Diagrama de estados 17
Figura B-7: Diagrama de secuencia 17
Figura B-8: Diagramas de Interacción: diagrama de secuencia y de colaboración 18
Figura B-9: Diagrama de actividades 19
Figura B-10: Diagrama de clase para el patrón Abstract Factory 21
Figura B-11: Diagrama de clase para el patrón Builder 26
Figura B-12: Diagrama de secuencia para el patrón Builder 27
Figura B-13: Diagrama de clase para el patrón Factory Method 31
Figura B-14: Diagrama de clase para el patrón Prototype 34
Figura B-15: Diagrama de clase para el patrón Singleton 38
Figura B-16: Diagrama de clase para el patrón Adapter (herencia multiple) 41
Figura B-17: Diagrama de clase para el patrón Adapter (composición objetos) 42
Figura B-18: Diagrama de clase para el patrón Bridge 46
Figura B-19: Diagrama de clase para el patrón Composite 49
Figura B-20: Diagrama de clase para el patrón Decorator 55
Figura B-21: Diagrama de clase para el patrón Facade 60
Figura B-22: Diagrama de clase para el patrón Flyweight 64
Figura B-23: Diagrama de clase para el patrón Proxy 68
Figura B-24: Diagrama de clase para el patrón Chain of Responsability 72
Figura B-25: Diagrama de clase para el patrón Command 77
Figura B-26: Diagrama de secuencia para el patrón Command 78
Figura B-27: Diagrama de clase para el patrón Interpreter 82
Figura B-28: Diagrama de clase para el patrón Iterator 87
Figura B-29: Diagrama de clase para el patrón Mediator 90
Figura B-30: Diagrama de clase para el patrón Memento 94
Figura B-31: Diagrama de secuencia para el patrón Memento 95

Figura B-32: Diagrama de clase para el patrón Observer 100
Figura B-33: Diagrama de secuencia para el patrón Observer 101
Figura B-34: Diagrama de clase para el patrón State 105
Figura B-35: Diagrama de clase para el patrón Strategy 110
Figura B-36: Diagrama de clase para el patrón Template Method 114
Figura B-37: Diagrama de clase para el patrón Visitor 118
Figura B-38: Diagrama de secuencia para el patrón Visitor 119

Anexo C. Catálogo de patrones de diseño.

A continuación se muestra un resumen del propósito de cada patrón de diseño explicados a detalle a lo largo del documento.

Nombre	Propósito
Abstract Factory	Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.
Adapter	Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permite que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.
Bridge	Desacopla una abstracción de su implementación, de manera que ambas pueden variar de forma independiente.
Builder	Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.
Chain of Responsability	Evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que ésta sea tratada por algún objeto.
Command	Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer las operaciones.
Composite	Combina objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.
Decorator	Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.
Factory Method	Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.

Flyweight	Usa el comportamiento para permitir un gran número de objetos de grano fino de forma eficiente.
Interpreter	Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar sentencias del lenguaje.
Iterator	Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.
Mediator	Define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.
Memento	Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado más tarde.
Observer	Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifica y se actualizan automáticamente todos los objetos que dependen de él.
Prototype	Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando de este prototipo.
Proxy	Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.
Singleton	Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.
Strategy	Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.
Template Method	Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.
Visitor	Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

Glosario

acoplamiento
El grado en que los componentes de software dependen unos de otros.

acoplamiento abstracto
Dada una clase A que tiene una referencia a una clase abstracta B, la clase A se dice que tiene un acoplamiento abstracto a B. Se llama acoplamiento abstracto porque A se refiere al tipo de un objeto, no a un objeto concreto.

clase
Una clase define la interfaz de un objeto y su implementación. Especifica la representación interna de un objeto y define las operaciones que éste puede llevar a cabo.

clase abstracta
Una clase cuyo principal propósito es definir una interfaz. Una clase abstracta delega parte de su implementación, o toda, en sus subclases. No se pueden crear instancias de una clase abstracta.

clase mezclable
Una clase diseñada para ser combinada con otras por medio de la herencia. Las clases mezclables suelen ser abstractas.

clase amiga
Una clase que tiene los mismos permisos de acceso a las operaciones y datos de la clase que la propia clase.

clase concreta
Una clase que no tiene operaciones abstractas. Se pueden crear instancias de ella.

clase padre
La clase de la que hereda otra clase. Tienen como sinónimos ***superclase***, ***clase base*** y ***clase antecesora***.

composición de objetos
Ensamblar o componer objetos para obtener un comportamiento más complejo.

constructor
Una operación que se invoca automáticamente para inicializar las nuevas instancias.

delegación
Un mecanismo de implementación mediante el cual un objeto redirige o delega una petición a otro objeto. El delegado lleva a cabo la petición en nombre del objeto original.

diagrama de clases
Un diagrama que representa clases, su estructura y operaciones internas, y las relaciones estáticas entre ellas.

diagrama de interacción
Un diagrama que muestra el flujo de peticiones entre objetos.

diagrama de objetos
Un diagrama que representa una determinada estructura de objetos en tiempo de ejecución.

encapsulación
El resultado de ocultar la representación e implementación en un objeto. La representación no es visible y no se puede acceder a ella directamente desde el exterior del objeto. El único modo de acceder a la representación de un objeto y de modificarla es a través de sus operaciones.

enlace dinámico
La asociación en tiempo de ejecución entre una petición a un objeto y una de sus operaciones.

framework
Un conjunto de clases cooperantes que forman un diseño reutilizable para una determinada clase de software. Un framework proporciona una guía arquitectónica para dividir el diseño de clases abstractas y definir sus responsabilidades y colaboraciones. Un desarrollador adapta el framework a una aplicación concreta heredando y componiendo instancias de las clases del framework.

herencia
Una relación que define una entidad en términos de otra. La ***herencia de clases*** define una nueva clase de en términos de una o más clases padre. La nueva clase hereda su interfaz y las implementaciones de sus padres. La nueva clase se dice que es una ***subclase*** o una ***clase derivada***. La herencia de clases combina herencia de interfaces y herencia de implementación. La ***herencia de interfaces*** define una nueva interfaz en términos de una o varias interfaces existentes. La

herencia de implementación define una nueva implementación en términos de una o varias implementaciones existentes.

interfaz

El conjunto de todas las signaturas definidas por las operaciones de un objeto. La interfaz describe el conjunto de peticiones a las que puede responder un objeto.

objeto

Una entidad en tiempo de ejecución que empaqueta datos y los procedimientos que operan sobre esos datos.

objeto agregado

Un objeto que se compone de subobjetos. Los subobjetos se denominan ***partes***, y el agregado es responsable de ellos.

operación

Los datos de un objeto sólo pueden ser manipulados por sus operaciones. Un objeto realiza una operación cuando recibe una petición. En diferentes lenguajes se les denomina ***funciones miembro*** o ***método***.

operación abstracta

Una operación que declara una signatura pero no la implementa.

operación clase

Una operación que pertenece a una clase y no a un objeto individual.

patrón de diseño

Un patrón de diseño enumera, da los motivos y explica sistemáticamente un diseño general que resuelve un problema de diseño recurrente en los sistemas orientados a objetos. Describe el problema, la solución, cuándo aplicar ésta y sus consecuencias. También ofrece trucos de implementación y ejemplos. La solución es una disposición general de clases y objetos que resuelven el problema. Está adaptada e implementada para resolver el problema en un determinado contexto.

petición

Un objeto lleva a cabo una operación cuando recibe la petición correspondiente de otro objeto. Un sinónimo frecuente de petición es ***mensaje***.

polimorfismo
La capacidad de sustituir los objetos que se ajustan a una interfaz por otros en tiempo de ejecución.

protocolo
Extiende el concepto de interfaz para incluir todas las secuencias de peticiones permitidas.

receptor
El objeto destino de una petición.

redefinición.
Volver a definir una operación (heredada de una clase padre) en una subclase.

referencia de objetos
Un valor que identifica a otro objeto.

relación de agregación
La relación entre un objeto agregado y sus partes. Una clase define esta relación con sus instancias (objetos agregados).

relación de asociación
Una clase que se refiere a otra clase tiene una asociación con esa clase.

signatura
La signatura de una operación define su nombre, parámetros y tipo de retorno.

subclase
Una clase que hereda de otra.

subsistema
Un grupo independiente de clases que colaboran para llevar a cabo una serie de responsabilidades.

subtipo
Un tipo que es subtipo de otro si su interfaz contiene la interfaz de aquél.

supertipo
El tipo padre del que hereda otro tipo.

tipo
El nombre de una determinada interfaz.

tipo parametrizado

Un tipo que deja sin especificar alguno de sus tipos constituyentes. Los tipos sin especificar se proporcionan como parámetros en el momento de su uso. También se les llama plantillas.

toolkit

Una colección de clases que proporciona una funcionalidad útil pero que no definen el diseño de una aplicación.

variable de instancia

Un elemento de datos que define parte de la representación de un objeto.

Bibliografía

[AIS77] Alexander, Christofer / Ishikawa, Sara / Silverstein, Murray / Jacobson, Max/ Fiksdahl-King, Ingrid / Angel, Shlomo. *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, New York, 1977.

[Ala00] Alarcón, Raúl. *Diseño orientado a objetos con UML*, pág. 9-26, Grupo EIDOS, Madrid España, 2000.

[Boo04] Booch, Grady. *Object-Oriented Analysis and Design with Applications*, Addison-Wesley, 2004.

[BRJ05] Booch, Grady / Rumbaugh, James / Jacobson, Ivan. *The unified modeling language user guide*, Person Education, 2005.

[Coo98] Cooper, James. *The Design Patterns Java Companion*, pág. 14-51, Addison Wesley, 1998.

[CS03] Chonoles, Michael Jesse / Schardt, James. *UML 2 for Dummies*, pág. 21-23, Wiley Publishing, Inc., Indianapolis, Indiana, 2003.

[FS99] Fowler, Martin / Scout, Kendall. *UML gota a gota*, pág. 1-58, Addison Wesley Longman de México, 1999.

[GHJ95] Gamma, Erich / Helm, Richard / Johnson, Ralph / Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Gra02] Grand, Mark. *Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition, en A Brief History of Patterns*, pág. 3-4, Wiley Publishing, Inc., Indianapolis, Indiana, 2002.

[JZ91] Johnson, Ralph E. / Zweig, Jonathan. *Delegation in C++*, Journal of Object-Oriented Programming, pág. 27, 4 de noviembre de 1991.

[Lau03] Laurent, Grégoire. *UML Quick Reference Card*, The Quick Reference Site (http://www.digilife.be/quickreferences), 2003.

[OJ93] Opdyke, William F. / Johnson, Ralph E. *Creating abstract superclasses by refactoring*. En *Proceedings of the 21st Annual Computer Science Conference*, pág. 66-73, Indianapolis, febrero de 1993.

[Sch98] Schmuller, Joseph. *Aprendiendo UML en 24 horas*, pág. 67-172, Prentice Hall, México, 1998.

[Sny86] Snyder, Alan. *Encapsulation and inheritance in object-oriented languajes*. En *Object-Oriented Programing Systems, Languajes, and Applications Conference Proceedings*, pág. 38-45, ACM Press, 1986.

[ST06] Shalloway, Alan / Trott, James R. *Pattern Oriented Design: Using Design Patterns From Analysis to Implementation*, Manuals for Pattern Oriented Design by Net Objetives (http://www.netobjetives.com), 2006.

[SM01] Stelting, Stephen / Maassen, Olav. *Applied Java Patterns*, Prentice Hall PTR Author Supplements: viewable source code examples for material supplemental to *Applied Java Patterns* (http://authors.phptr.com/appliedjavapatterns/code.html), 2001.

[YA03] Yacoub, Sherif M. / Ammar, Hany H. *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems*, Addison Wesley, 2003.