Guía práctica de estudio 13: Patrones de diseño

Elaborado por:
M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:
M.C. Alejandro Velázquez Mena

Guía práctica de estudio 13: Patrones de diseño

Objetivo:

Implementar una aplicación en un lenguaje orientado a objetos utilizando algún patrón de diseño.

Actividades:

- Conocer diferentes patrones de diseño.
- Implementar una aplicación utilizando algún patrón de diseño.

Introducción

En la ingeniería de software, un patrón de diseño es una solución repetible y general para problemas de ocurrencia cotidianos en el diseño de software. Es importante aclarar que un patrón de diseño no es un diseño de software terminado y listo para codificarse, más bien es una descripción o modelo (template) de cómo resolver un problema que puede utilizarse en diferentes situaciones.

Por lo tanto, los patrones de diseño permiten agilizar el proceso de desarrollo de solución debido a que proveen un paradigma desarrollado y probado.

Un diseño de software efectivo debe considerar detalles que tal vez no sean visibles hasta que se implemente la solución, es decir, debe anticiparse a los problemas y tratar de cubrir todos los resquicios. La reutilización de patrones de diseño ayuda a prevenir los detalles sutiles que provocarían problemas más grandes, además de ayudar a la legibilidad de código para programadores o analistas que estén familiarizados con patrones.

Patrones de software

Cada patrón de diseño describe un problema que ocurre una y otra vez de manera cotidiana y describe el núcleo de la solución, de tal manera que esa solución se puede utilizar muchas veces.

De manera general, un patrón de diseño posee 4 elementos esenciales:

1. El **nombre del patrón** se utiliza para describir el diseño del problema, su solución y sus consecuencias en una o dos palabras.

2. El **problema** describe cuando se debe aplicar el patrón. Explica el problema y su contexto. Puede describir problemas de diseño específico, como sería la representación de algoritmos como objetos. Puede describir estructuras de clases u objetos que son sintomáticos de un diseño inflexible, para evitar caer en ellos. Algunas veces el problema puede incluir una lista de condiciones que se deben cumplir para que el patrón tenga sentido y sea viable su implementación.

3. La **solución** describe los elementos que forman el diseño, su relación, responsabilidades y colaboraciones. La solución no describe de manera particular un diseño o implementación, más bien provee una descripción abstracta del diseño de un problema y cómo, de manera general, el acomodo de elementos (clases y objetos) resuelve el problema.

4. Las **consecuencias** son el resultado y las recompensas de haber aplicado el patrón. Aunque las consecuencias son a menudo poco mencionadas, cuando se describe el diseño de decisiones se vuelven críticas para evaluar el diseño alternativo y poder entender el costo-beneficio de aplicar el patrón. Las consecuencias en la ingeniería de software a menudo tienen que ver con compensaciones en espacio y tiempo. Debido a que la reutilización es un factor en el diseño orientado a objetos, las consecuencias de un patrón de diseño incluyen su impacto en la flexibilidad del sistema, su extensibilidad y su portabilidad.

Los puntos de vista afectan la interpretación de qué es y qué no es un patrón. Lo que para una persona puede ser un patrón de diseño para otra puede ser un bloque primitivo de construcción.

El diseño de patrones no establece un diseño como una lista ligada o una tabla hash que puede ser codificado en clases y reutilizado una y otra vez. Tampoco son tan complejos como para llegar a especificar un diseño en un dominio específico para una aplicación o sistema.
El diseño de patrones es, más bien, una descripción de la comunicación que existe entre objetos y clases, las cuales se pueden personalizar para resolver un problema de diseño general en un contexto en particular.

Se consideran tres niveles dentro de los patrones de software:

- Patrones arquitectónicos: Los cuales describen soluciones al más alto nivel de software y hardware. Normalmente soportan requerimientos no funcionales (criterios de evaluación).

- Patrones de diseño: Describen soluciones en un nivel medio de estructuras de software. Normalmente soportan requerimientos funcionales (especificaciones técnicas del sistema).

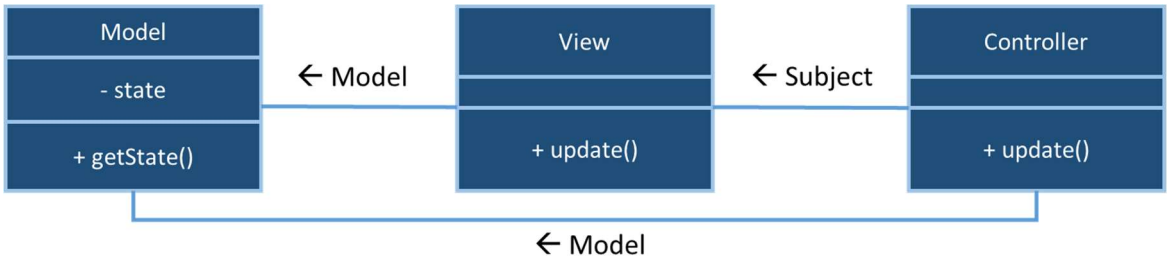
- Patrones de programación: Describen soluciones en el nivel de software más bajo, a nivel de clases y métodos. Normalmente soportan características específicas de un lenguaje de programación.

Así mismo, dentro del catálogo de los patrones de diseño existen tres grandes grupos: los creacionales, los estructurales y los de comportamiento.

MVC

El patrón de diseño Modelo-Vista-Controlador (MVC) asigna objetos con alguno de los tres roles. El patrón define los roles que juegan los objetos en la aplicación, así como la manera en la que éstos se comunican entre sí. Dentro de una implementación MVC, la colección de objetos del mismo tipo forman lo que se conoce como capa, por ejemplo, la capa del modelo.

La implementación del patrón MVC provee varios beneficios. Los objetos tienden a ser más reutilizados, además, las interfaces creadas suelen estar mejor definidas y, en general, suelen ser aplicaciones fáciles de extender.



Modelo MVC

Modelo

Los objetos que forman parte del Modelo encapsulan la información de la aplicación y definen la lógica con la que se van manipular los datos. Así mismo, un objeto del Modelo puede tener relaciones uno a uno o uno a muchos con otros objetos del Modelo.

Debido a que los objetos del Modelo representan conocimiento y experiencia relacionada con un dominio específico del problema, éstos pueden ser en problemas con un dominio similar.

Idealmente, los objetos del Modelo no deben tener una conexión explícita con los objetos de la Vista, que son los que muestran la información y permiten al usuario modificarla. Las acciones de los usuarios en la capa de Vista son comunicadas al Modelo a través del Controlador. Así, cuando un objeto del Modelo cambia, éste notifica al objeto del Controlador para que actualice los objetos de Vista necesarios.

Vista

Un objeto de Vista es el objeto de la aplicación que el usuario puede ver. El objeto de Vista sabe cómo mostrarse y puede responder a las acciones del usuario. El propósito máximo de un objeto de Vista es mostrar la información del objeto del Modelo y habilitar la edición de información.

Los objetos de Vista se enteran de los cambios en la información a través de los objetos del Controlador y la comunicación con el usuario. Así mismo, envían la información recibida del usuario a los objetos del Controlador para que éste se los envíe a los objetos del Modelo.

Controlador

Un objeto del Controlador actúa como intermediario entre uno o más objetos de la Vista y uno o más objetos del Modelo. Por lo tanto, los objetos del Controlador son un conducto a través del cual los objetos de la Vista se enteran de los cambios de los objetos del Modelo y viceversa. Los objetos del Controlador también pueden establecer y coordinar tareas de la aplicación y administrar el ciclo de vida de los objetos.

En general, los objetos del Controlador interpretan las acciones realizadas por los usuarios en los objetos de la Vista y comunican estas acciones hacia la capa del Modelo. Así mismo, cuando un objeto del Modelo cambia, el objeto del Controlador envía la información a los objetos de la Vista para que ésta pueda ser mostrada.

Nombre: Modelo-Vista-Controlador (MVC).

Problema:

Se requieren mostrar varias pantallas de manera gráfica (Vista) con la información que se encuentra en un repositorio de datos (Modelo). Debe existir un mediador (Controlador) entre las pantallas y el repositorio de datos para facilitar la comunicación y el paso de información.

Solución.

Se debe desacoplar la información (Modelo de datos) del código de la interfaz gráfica que construye la vista, permitiendo así construir varias vistas.

Cuando se ejecute la aplicación se puede registrar cada vista con el modelo de información correspondiente a través de un controlador de paso de información (Observador).

Consecuencias

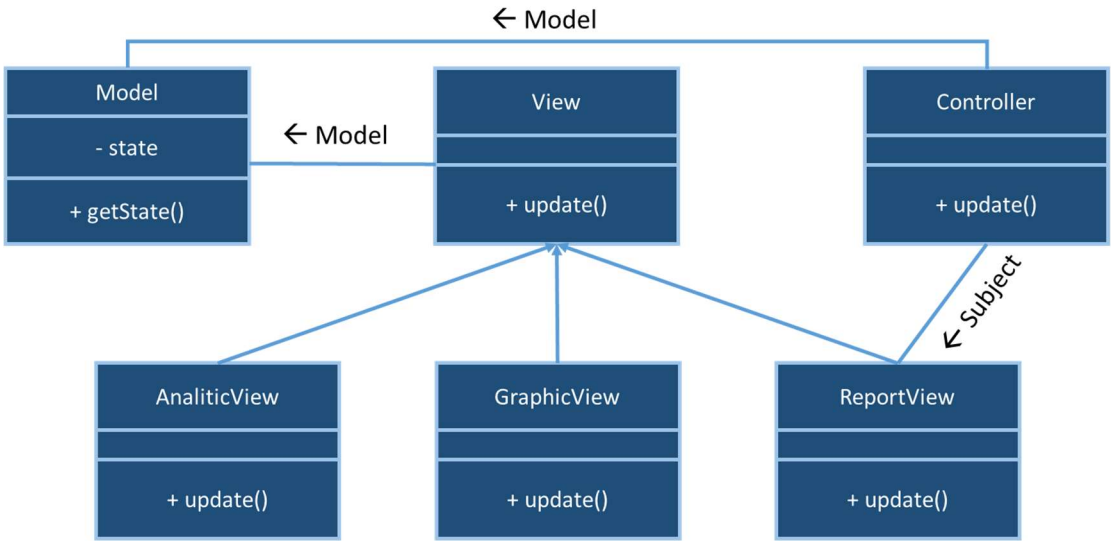
MVC desacopla vistas y modelos estableciendo un protocolo de suscriptor/notificador entre ellos, de tal manera que se debe asegurar que la vista refleje el estado del modelo. Así, si la información del modelo cambia, el modelo debe notificar a la vista que depende de él.

Se pueden acoplar varias vistas a un modelo, para mostrar diferente información. También se pueden crear nuevas vistas sin necesidad de reescribir el modelo.

Permite manejar las entradas del usuario sin necesidad de cambiar la presentación (Vista), ya que la capacidad de responder ante un estímulo de la vista se encapsula en el controlador, el cual puede implementar diversas respuestas dependiendo del estímulo enviado por la vista, sin modificar ésta.

Ejemplo:

Una aplicación debe mostrar información analítica (información numérica) y gráfica (gráfica de barras) a partir de una base de datos. Así mismo, la aplicación puede presentar un reporte de la información mostrada.



Patrón de diseño creacional

Este tipo de diseño abstrae el proceso de instanciación (creación de objetos), permitiendo con esto que el sistema sea independiente de cómo sus objetos son creados, se componen y se representan. Un patrón de diseño creacional de clases utiliza la herencia para variar la clase que crea una instancia, mientras que un patrón de diseño creacional de objetos delegará la instanciación a otro objeto.

Los patrones creacionales se vuelven importantes como sistemas que evolucionan para depender más de la composición de objetos que de la herencia de clases. A medida que esto sucede, el énfasis se desplaza lejos de un código duro y hacia la definición de un conjunto fijo de comportamientos con desempeños fundamentales que pueden ser parte de conjuntos más complejos. Así, la creación de objetos con un comportamiento particular requiere más que simplemente instanciar una clase.

En estos patrones existen dos temas recurrentes. El primero, se encapsula todo el conocimiento sobre una clase en específico que utiliza el sistema. El segundo, se oculta la manera en la que las instancias de las clases del sistema se crean y se unen entre sí. Por ende, los patrones creacionales proporcionan mucha flexibilidad en lo que se crea, quién lo crea, cómo lo crea y cuando lo hace. Esto permite configurar un sistema objetos producto que pueden variar ampliamente en su estructura y funcionalidad.

Es importante recalcar que algunas veces los patrones relacionales pueden competir entre sí por ser igual de rentables. Algunas otras veces se pueden complementar.

Los patrones creacionales más conocidos son:

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Siendo uno de los más utilizados y conocidos el patrón de diseño creacional MVC.

Patrón de diseño creacional Singleton

Nombre: Singleton.

Problema:

En la aplicación debe existir únicamente una instancia de una clase y ésta (la instancia) debe ser accesible para los clientes a través de un punto de acceso bien conocido.

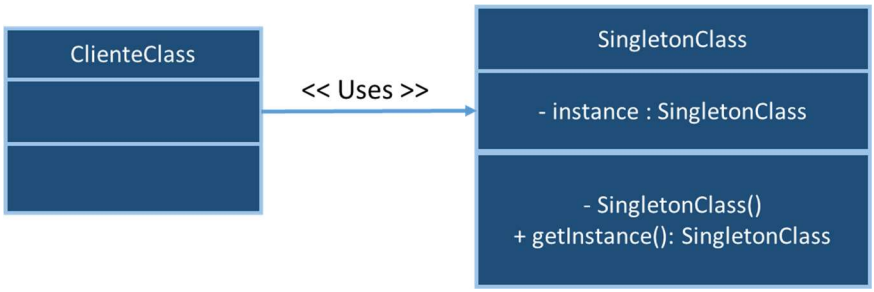
Para algunas aplicaciones es importante tener solamente una instancia en ejecución. Por ejemplo, en un sistema de impresoras se debe tener una sola cola de impresión.

Solución.

La clase debe ser la responsable de hacer el seguimiento para una única instancia. La clase puede asegurar que no se pueda crear otra instancia, así como proveer una ruta para acceder a dicho objeto.

Por lo tanto, lo que se debe hacer es una clase que:

a) Mantenga su constructor privado.
b) Mantenga una instancia como privada y estática.
c) Provea un método público y estático para acceder a la instancia.



Modelo Singleton

Consecuencias

Los beneficios de utilizar el patrón de diseño Singleton son:

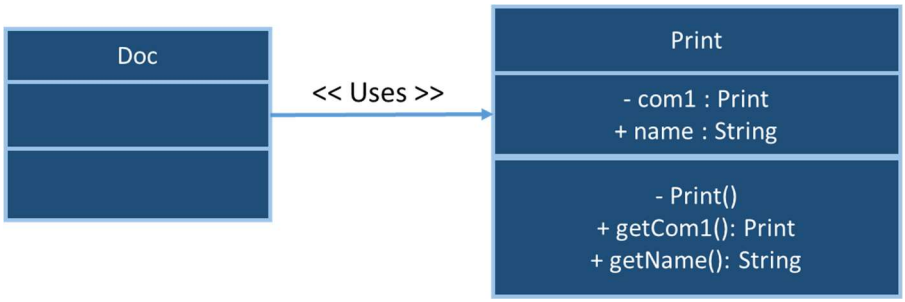
- Controlar el acceso a la única instancia. Debido a que se encapsula la única instancia, se posee un estricto control sobre cómo y cuándo se puede acceder a ella.
- Reducir el espacio de nombres. Evita llenar el espacio de nombres con variables globales, debido a que se almacena una única instancia.
- Refinar las operaciones y su representación. Una clase puede tener subclases y se puede configurar la aplicación para que se tenga una instancia de las subclases. Se puede, por tanto, crear una instancia de alguna subclase en tiempo de ejecución.
- Permitir un número variable de instancias. El patrón permite modificar para crear más de una instancia de la clase, así mismo se pueden controlar todas las instancias de manera general, lo único que hay que modificar son las operaciones que permiten acceso a la instancia.
- Es más flexible que las operaciones de clase. Se pueden empaquetar las funcionalidades del patrón mediante métodos de clase.

El patrón de diseño creacional Singleton se puede usar cuando:

- Debe existir exactamente una instancia de una clase y ésta debe ser accesible desde un punto de acceso bien conocido.
- Cuando una única instancia deba existir para una subclase y los clientes deban ser capaces de utilizar una subclase sin modificar su código.

Ejemplo:

Una impresora dentro de una compañía debe manejar todos los documentos que se envíen para mantener un orden y un registro, ese es un buen lugar para proveer una clase que mantenga una sola instancia de la impresora para todos los objetos que la utilicen. Por lo tanto, se puede usar el patrón Singleton para manejar las impresiones.



Patrón de diseño estructural

A los patrones de diseño estructural les interesa como están compuestos clases y objetos para formar estructuras grandes.

El patrón estructural de clases usa la herencia para componer la implementación de múltiples clases base. Esto es, cuando se mezclan varias clases base dentro de una (herencia múltiple), la clase resultante combina las propiedades de todas las clases base, generando con ello una estructura grande. Este patrón es muy útil cuando se quiere hacer que clases desarrolladas de manera independiente y en diferentes bibliotecas trabajen unidas.

El patrón estructural de objetos describe la manera en la que se componen los objetos para reaccionar ante una nueva funcionalidad. Los objetos poseen una composición flexible, esto es, tienen la habilidad de cambiar su forma en tiempo de ejecución, lo cual no es posible con una composición estática.

Los patrones estructurales más conocidos son:

- Composite
- Proxy
- Flyweight
- Fecade
- Bridge

El patrón estructural de objetos Composite (compuesto) describe como crear una jerarquía de clases a partir de clases con dos tipos de elementos: primitivos y compuestos. Los objetos compuestos permiten crear instancias de ambos elementos dentro de una estructura compleja.

El patrón Proxy actúa como un sustituto de otro objeto, el cual puede ser utilizado de diferentes maneras. Puede actuar como una representación local para un objeto dentro de un espacio remoto. Puede representar un objeto grande que se carga sobre demanda. Protege el acceso a objetos con información sensible.

El patrón Flyweight (peso mosca) define una estructura para compartir objetos. Los objetos se pueden compartir por, al menos, dos razones: eficiencia y consistencia. Flyweight se enfoca en compartir un espacio eficiente, para aquellas aplicaciones que utilizan muchos objetos, ya que se puede ahorrar bastante espacio compartiendo objetos en lugar de replicarlos. La información adicional que necesitan los objetos para realizar sus tareas se les pasa sobre demanda (cuando la necesitan).

El patrón Facade permite crear un solo objeto que representa un subsistema completo. Está compuesto por un conjunto de objetos y, para llevar a cabo sus tareas, envía mensajes a las instancias para comunicarles sus responsabilidades.

El patrón Bridge (puente) separa la abstracción de un objeto de su implementación, de tal manera que se puede modificar el comportamiento de manera independiente.

El patrón Decorator (decorador) describe la posibilidad de agregar funcionalidades a los objetos de manera dinámica. Crea los objetos de manera recursiva, de tal forma que permite agregar un número ilimitado de funciones adicionales. Por ejemplo, un objeto Decorador contiene un componente de interfaz de usuario, se puede agregar un decorador al componente como puede ser un borde o un sombreado o, incluso, se puede agregar una funcionalidad como una barra de desplazamiento o un zoom.

Patrón de diseño estructural Composite

Nombre: Composite.

Problema:

Se requiere notificar a un conjunto variado de objetos que un evento ha ocurrido en la aplicación.

Una modificación en un objeto debe desencadenar una serie de cambios en otros objetos, sin embargo, no se tiene la certeza del número de objetos que este cambio afectará.

Solución.

Crear una clase abstracta *Subject* la cual contenga un conjunto de observadores, de tal manera que cuando un cambio ocurra en Subject, el observador le informará a todos los objetos de su conjunto de dicho cambio.

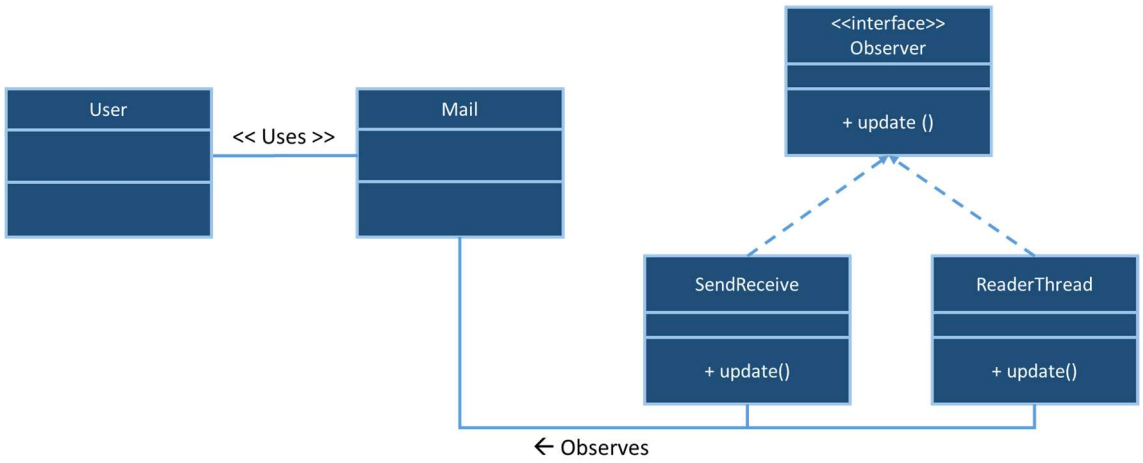
Consecuencias

Los beneficios de utilizar el patrón de diseño Composite son:

- Definir una jerarquía de clases que consiste en objetos primitivos y compuestos, donde los objetos primitivos pueden componer objetos más complejos, que a su vez pueden componer objetos más robustos y así de manera recursiva.
- Hacer la tarea del cliente simple, ya que puede tratar con estructuras compuestas u objetos de manera uniforme.
- Hacer fácil la creación de nuevas funcionalidades, debido a que los cambios se ven reflejados en las clases compuestas, el cliente no se ve obligado a agregar código y sí puede utilizar las nuevas funcionalidades.
- La creación de nuevas funcionalidades de manera sencilla provoca que sea difícil restringir los componentes nuevos, por la misma facilidad para agregar.

Ejemplo:

Una aplicación de correo debe estar actualizando constantemente su bandeja en busca de nuevo correo. Además, el usuario puede cerciorarse de manera manual de la recepción de nuevo correo. En ambos casos, la bandeja de entrada se debe actualizar si es que existe nuevo correo. Un observador se encarga de actualizar la información en la bandeja de entrada cuando es necesario en ambos casos.



Patrón de diseño de comportamiento

Los patrones de diseño de comportamiento están relacionados con los algoritmos y la asignación de responsabilidades entre objetos, es decir, no sólo describen los patrones de objetos o clases, también se encargan de los patrones de comunicación entre ellos. Este patrón se caracteriza por un complejo control de flujo, el cual es difícil manejar en tiempo de ejecución, por lo tanto, provoca que se cambie el enfoque lejos del control de flujo para concentrarse exclusivamente en la manera en la que los objetos se comunican.

Los patrones de comportamiento más conocidos son:

- Método Template
- Interpreter
- Mediator
- Chain of responsability
- Observer
- Strategy
- Command
- State

El patrón de comportamiento de clases utiliza la herencia para distribuir el comportamiento entre clases. Dentro de los patrones de este tipo se encuentran: el método Template (modelo) y el Interpreter (intérprete).

El método Template es un patrón muy utilizado y fácil de implementar, consiste en una definición abstracta de un algoritmo, paso a paso, donde en cada paso se invoca ya sea a una operación abstracta o a una operación primitiva.

El Interpreter define un conjunto de normas y reglas como una jerarquía de clases, e implementa un intérprete como una operación en las instancias de estas clases.

El patrón de comportamiento de objetos se basa la composición de objetos más que en la herencia. Algunos describen cómo un grupo de dos objetos cooperan para desarrollar una tarea que no podría realizar un objeto solo. Un detalle que sobresale es cómo el par de objetos se conocen para cooperar entre sí, ya que entonces los objetos deberían tener una referencia uno hacia el otro, lo que incrementaría el acoplamiento. Además, en un caso extremo, cada objeto debería poseer una referencia hacia todos los otros objetos.

El patrón Mediator (mediador) permite optimizar las referencias de los pares de objetos utilizando un objeto mediador entre ellos, y como los objetos no necesitan este objeto mediador se elimina el acoplamiento.

El patrón de Chain of responsability (cadena de responsabilidades) también permite bajar el acoplamiento entre objetos enviando solicitudes a un objeto utilizando una cadena de objetos candidatos, donde cualquier candidato podría cumplir la solicitud dependiendo de las condiciones de ejecución. El número de candidatos es ilimitado y se puede seleccionar qué objetos van a ser parte de la cadena de candidatos en tiempo de ejecución.

El patrón Observer (observador) define, mantiene y administra una dependencia entre objetos, informando en tiempo real sobre los cambios que se presentan en uno u otro objeto.

El patrón Strategy (estratégico) encapsula un algoritmo dentro de un objeto, haciendo con esto fácil de especificar y cambiar el algoritmo que utiliza un objeto.

El patrón Command (comando) encapsula una solicitud dentro de un objeto, de tal manera que éste se puede pasar como parámetro, guardar en un histórico o manipular de alguna otra manera.

El patrón State (estado) encapsula el estado de un objeto, de tal forma que el objeto puede cambiar su comportamiento cuando el estado del objeto cambie.

Patrón de diseño de comportamiento

Nombre: Observer.

Problema:

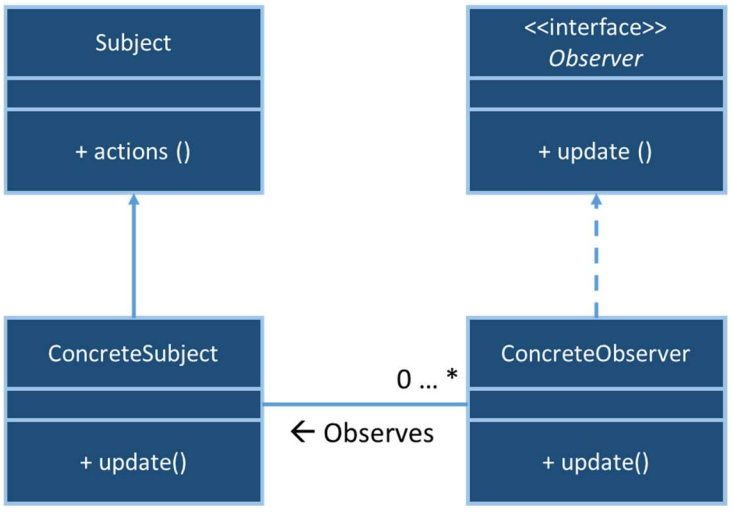
Se necesita notificar a un grupo de diferentes objetos que un evento ha ocurrido en la aplicación.

Un cambio en el estado de un objeto requiere que se cambie el estado de otros objetos, sin saber cuántos son los objetos que deben reflejar el cambio.

Solución.

Se crea una clase abstracta la cuál será la encargada de manejar un conjunto de objetos observadores en la aplicación.

Cuando un cambio ocurra en el sujeto observado, se debe notificar a todos los objetos observadores dentro del conjunto.



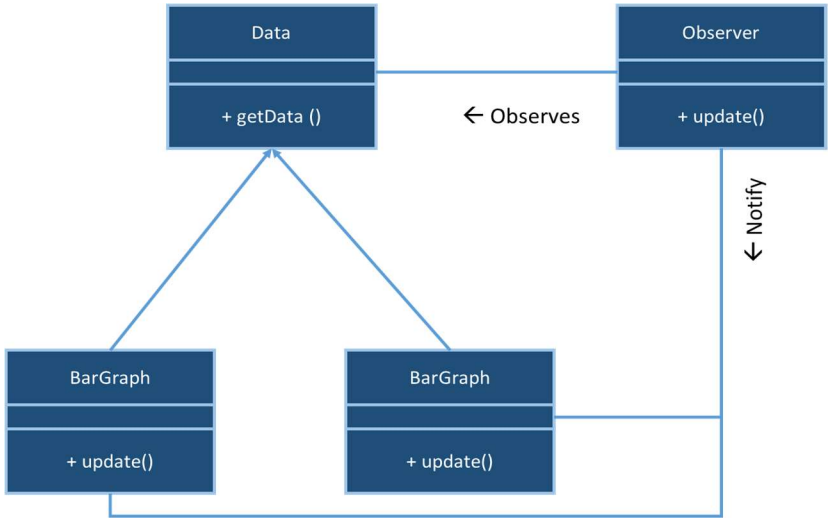
Modelo Singleton

Consecuencias

El patrón de diseño de comportamiento Observer permite partir un sistema en una colección de clases cooperativas, generando con esto consistencia de información entre los objetos relacionados. Permite que la clase tenga un bajo acoplamiento y se pueda reutilizar.

Ejemplo:

Se desean crear gráficas de una aplicación gráfica que permita separar el aspecto (la presentación) de la información subyacente. Además, se desea que los datos y la presentación de los mismos se puedan reutilizar de manera independiente y trabajar en conjunto.



Bibliografía

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