UNTDF
Universidad Nacional de Tierra del Fuego,
Antártida e Islas del Atlántico Sur

Patrones de Diseño

El Arq. Christopher Alexander dijo, "Cada patrón describe un problema que ocurre una y otra vez en nuestro ambiente, y describe entonces el núcleo de la solución a ese problema, de tal forma que ud. puede usar esa solución un millón de veces, sin nunca hacerlo de la misma forma dos veces"

Patrones de Diseño

Tipos de Patrones (Software)

- Patrones de arquitectura: Aquellos que expresan un esquema organizativo estructural fundamental para sistemas de software.
- Patrones de diseño: Aquellos que expresan esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de software.
- Dialectos: Patrones de bajo nivel específicos para un lenguaje de programación o entorno concreto.

Antipatrón de diseño: es semejante a un patrón, pero intenta prevenir contra errores comunes de diseño en el software. La idea de los antipatrones es dar a conocer los problemas que acarrean ciertos diseños muy frecuentes, para intentar evitar que diferentes sistemas acaben una y otra vez en el mismo callejón sin salida por haber cometido los mismos errores.

Patrones de Diseño

Patrón de diseño: Los patrones de diseño son descripciones de objetos que se comunican y clases que son personalizadas para resolver un problema de diseño general en un contexto particular.

Descripción de un patrón

Por lo general, un patrón tiene tiene 4 elementos esenciales:

1. nombre
2. problema
3. solución
4. consecuencias

Patrones de Diseño

1. El ***nombre*** del patrón es una referencia que podemos utilizar para describir un problema de diseño, sus soluciones, y consecuencias en una o dos palabras. Nombrar un patrón inmediatamente incrementa nuestro vocabulario de diseño. Esto nos permite diseñar a un nivel mas alto de abstracción. Tener un vocabulario de patrones nos permite hablar sobre ellos con nuestros colegas, en nuestra documentación, y aun con nosotros mismos. Facilita pensar acerca de diseños y comunicarlos y sus trade-offs a otros. Encontrar buenos nombres es una de las partes mas difíciles de construir un catálogo.

Patrones de Diseño

2. El ***problema*** describe cuando aplicar el patrón. Explica el problema y su contexto. Debe describir diseños específicos tales como representar algoritmos como objetos. Debe describir estructuras de clases u objetos que son sintomáticas de un diseño inflexible. Algunas veces el problema incluirá una lista de condiciones que deben ser alcanzadas para que tenga sentido aplicar el patrón.

Patrones de Diseño

3. la ***solución*** describe los elementos que forman el diseño, sus relaciones, responsabilidades, y colaboraciones. La solución no describe un diseño concreto particular o implementación, porque un patrón es como un template que puede ser utilizado en muchas situaciones diferentes. En su lugar, el patrón provee una descripción abstracta de un problema de diseño y como un arreglo general de elementos (clases y objetos en nuestro caso) solucionan el problema.

Patrones de Diseño

4. las ***consecuencias*** son los resultados y trade-offs de aplicar el patrón.
Aunque las consecuencias son a menudo ignoradas cuando describimos decisiones de diseño, son críticas para evaluar alternativas de diseño y para entender el costo y beneficios de aplicar el patrón. Las consecuencias para el software frecuentemente involucran trade-offs de espacio y tiempo. Deben abordar además problemas de lenguajes e implementaciones. Como el reuso es a menudo un factor en el diseño orientado a objetos, las consecuencias de un patrón incluyen su impacto en la flexibilidad del sistema, extensibilidad, o portabilidad. Listar las consecuencias de forma explicita ayuda a entender y evaluarlas.

Patrones de Diseño

Los patrones de diseño pretenden:

- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Asimismo, no pretenden:

- Imponer ciertas alternativas de diseño frente a otras.
- Eliminar la creatividad inherente al proceso de diseño.

selección y uso

No es obligatorio utilizar los patrones, solo es aconsejable en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable. "Abusar o forzar el uso de los patrones puede ser un error".

Patrones de Diseño

Catálogo de patrones de diseño (GoF)

Design Patterns: Elements of Reusable Object-Oriented Software - Erich Gamma, Ralph Johnson, Richard Helm, John Vlissides (1994)

Los patrones son clasificados de acuerdo a sus objetivos en estructurales, de comportamiento y creacionales.

- **Estructurales:** Estos patrones describen como relacionar clases y objetos para formar nuevas estructuras mas grandes y añadir funcionalidades.
- **De comportamiento:** Ayudan a definir la comunicación entre los objetos de un sistema.
- **Creacionales:** Hablan de la forma de como crear instancias de objetos su propósito es ocultar la manera en que se crea e inicializa un objeto.

Patrones de Diseño

Estructurales

- *Adapter(Adaptador):* Se encarga de adaptar(en vez de tener que cambiar el código) por ejemplo algún código ya existente en nuestro programa a alguna librería.
- *Bridge(Puente):* Separa una abstracción de su implementación.
- *Composite(Objeto compuesto):* Permite tratar objetos compuestos como si fueran simples.
- *Decorator(Envoltorio):* Proporciona funcionalidades a una clase dinamicamente.
- *Façade(Fachada):* Proporciona una interfaz simple para acceder a un grupo de interfaces de un sistema.
- *Flyweight(Peso ligero):* Disminuye la repetición cuando muchos objetos poseen información idéntica.
- *Proxy:* tiene como propósito proporcionar un subrogado o intermediario de un objeto para controlar su acceso.

Patrones de Diseño

De comportamiento

- *Chain of Responsibility(Cadena de responsabilidad):* Evita acoplar el emisor de una petición a su receptor dando a más de un objeto la posibilidad de responder a una petición.
- *Command(Orden):* Encapsulamiento de una operación en un objeto.
- *Interpreter(Interprete):* En un lenguaje, determina una gramática para dicho lenguaje, y las herramientas que se ocupan para interpretarlo.
- *Iterator(Iterador):* define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección sin importar su implementación.
- *Mediator(Mediador):* Especifica como un objeto puede coordinar a objetos de diferentes clases, funcionando como un conjunto.
- *Memento(Recuerdo):* Tiene por finalidad almacenar el estado de un objeto (o del sistema completo) en un momento dado de manera que se pueda restaurar en ese punto de manera sencilla.

Patrones de Diseño

De comportamiento (continuación)

- *Observer(Observador):* Define una dependencia del tipo uno-a-muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, notifica este cambio a todos los dependientes.
- *State(Estado):* Permite que un objeto cambie su comportamiento cada vez que se modifique su estado interno.
- *Strategy(Estrategia):* Permite tener varios métodos para resolver un problema, y elegir en tiempo de ejecución el método adecuado.
- *Template Method(Método plantilla):* Determina en un procedimiento la estructura de un algoritmo, repartiendo en las subclases ciertos pasos.
- *Visitor(Visitante):* Define otras operaciones sobre una jerarquía de clases sin cambiar las clases sobre las que funciona.

Patrones de Diseño

Creacionales

- *Abstract Factory(Fabrica abstracta):* permite trabajar con objetos de diferentes familias de manera que las familias no se mezclan entre ellas.
- *Builder(Constructor y virtual):* 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.
- *Factory Method(Método de fabricación):* Crea objetos de una clase de un subtipo determinado.
- *Prototype(Prototipo):* Se crean objetos complejos "clonando" objetos existentes.
- *Singleton(Instancia única):* Asegura que exista una única instancia para una clase y la creación de una forma de acceso global a dicha instancia.

Patrones de Diseño

Patrón State

Propósito: Permite a un objeto alterar su comportamiento según el estado interno en que se encuentre.

Motivación: El patrón State está motivado por aquellos objetos en que, según su estado actual, varía su comportamiento ante los diferentes mensajes. Como ejemplo se toma una clase TCPConection que representa una conexión de red, un objeto de esta clase tendrá diferentes respuestas según su estado (Listening, Close o Established). Por ejemplo la invocación al método Open de un objeto de la clase TCPConection diferirá su comportamiento si la conexión se encuentra en Close o en Established.

Problema: Existe una extrema complejidad en el código cuando se intenta administrar comportamientos diferentes según una cantidad de estados diferentes. Asimismo el mantenimiento de este código se torna dificultoso, e incluso se puede llegar en algunos casos puntuales a la incongruencia de estados actuales por la forma de implementación de los diferentes estados en el código (por ejemplo con variables para cada estado).

Patrones de Diseño

Consideraciones: Se debe contemplar la complejidad comparada con otras soluciones.

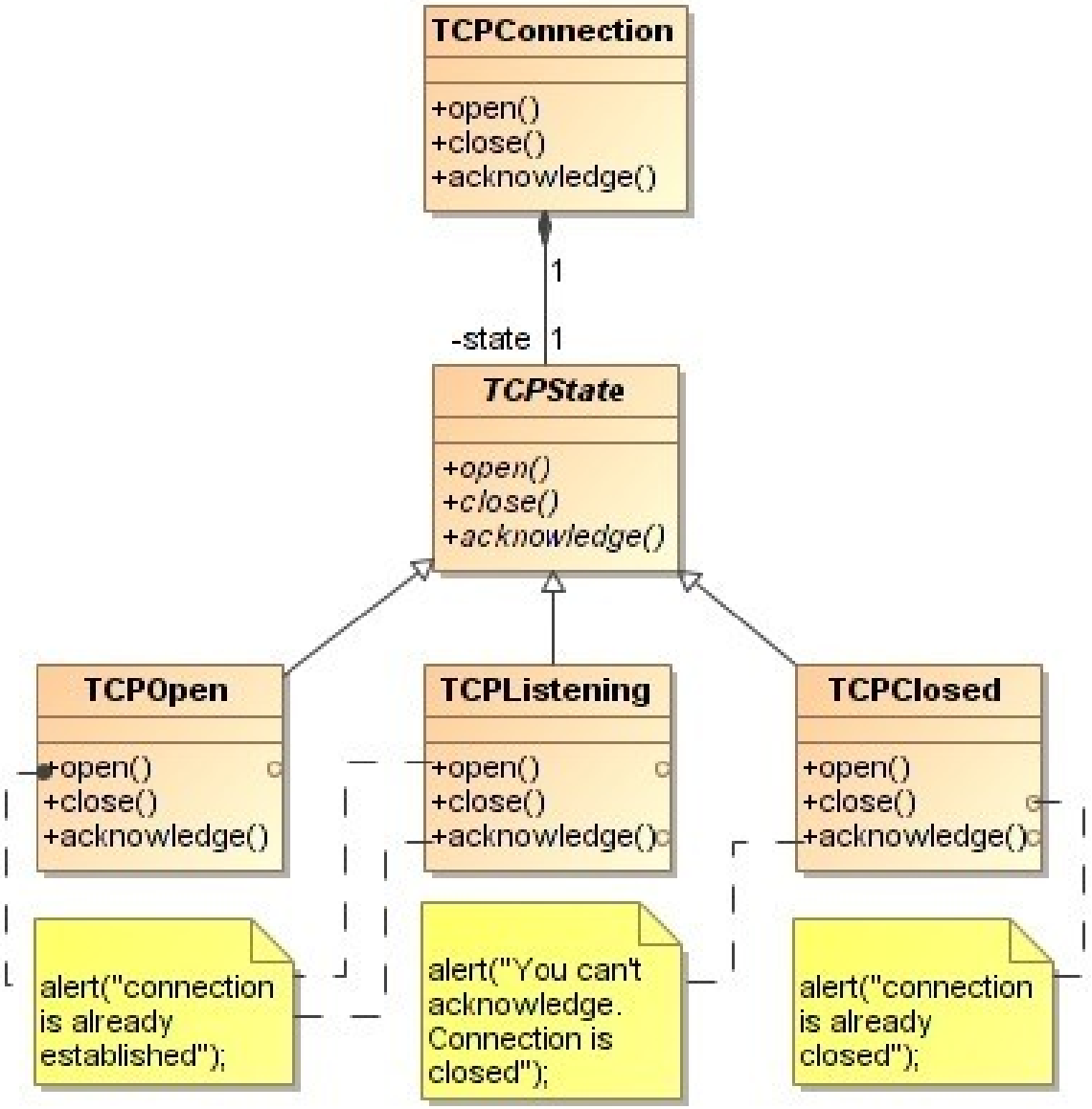
Solución: Se implementa una clase para cada estado diferente del objeto y el desarrollo de cada método según un estado determinado. El objeto de la clase a la que le pertenecen dichos estados resuelve los distintos comportamientos según su estado, con instancias de dichas clases de estado. Así, siempre tiene presente en un objeto el estado actual y se comunica con este para resolver sus responsabilidades.

La idea principal en el patrón State es introducir una clase abstracta TCPState que representa los estados de la conexión de red y un interfaz para todas las clases que representan los estados propiamente dichos. Por ejemplo la clase TCPEstablished y la TCPClose implementan responsabilidades particulares para los estados establecido y cerrado respectivamente del objeto TCPConnection.

Patrones de Diseño

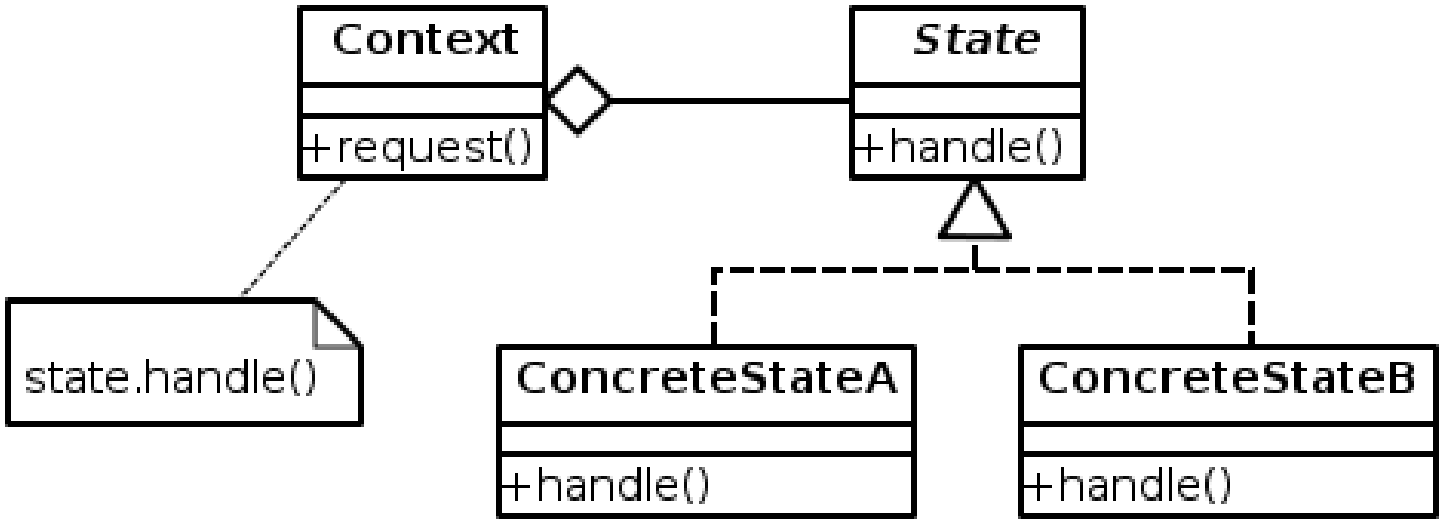
La clase TCPConnection mantiene una instancia de alguna subclase de TCPState con el atributo state representando el estado actual de la conexión. En la implementación de los métodos de TCPConnection habrá llamadas a estos objetos representados por el atributo state para la ejecución de las responsabilidades, así según el estado en que se encuentre, enviará estas llamadas a un objeto u otro de las subclases de TCPState.

Patrones de Diseño



Patrones de Diseño

Estructura UML



Patrones de Diseño

Participantes

- *Context(Contexto):* Este integrante define la interfaz con el cliente. Mantiene una instancia de ConcreteState (Estado Concreto) que define su estado actual
- *State (Estado):*Define una interfaz para el encapsulamiento de la responsabilidades asociadas con un estado particular de Context.
- *Subclase ConcreteState:*Cada una de estas subclases implementa el comportamiento o responsabilidad de Context.

El Contexto (Context) delega el estado específico al objeto ConcreteState actual Un objeto Context puede pasarse a sí mismo como parámetro hacia un objeto State. De esta manera la clase State puede acceder al contexto si fuese necesario. Context es la interfaz principal para el cliente. El cliente puede configurar un contexto con los objetos State. Una vez hecho esto, los clientes no tendrán que tratar con los objetos State directamente. Tanto el objeto Context como los objetos de ConcreteState pueden decidir el cambio de estado.

Patrones de Diseño

Colaboraciones

El patrón State puede utilizar el patrón Singleton cuando requiera controlar que exista una sola instancia de cada estado. Lo puede utilizar cuando se comparten los objetos como Flyweight existiendo una sola instancia de cada estado y ésta es compartida con más de un objeto.

Ventajas y desventajas

Se encuentran las siguientes ventajas:

- Se localizan fácilmente las responsabilidades de los estados específicos, dado que se encuentran en las clases que corresponden a cada estado. Esto brinda una mayor claridad en el desarrollo y el mantenimiento posterior. Esta facilidad la brinda el hecho que los diferentes estados están representados por un único atributo (state) y no envueltos en diferentes variables y grandes condicionales.

Patrones de Diseño

- Hace los cambios de estado explícitos puesto que en otros tipos de implementación los estados se cambian modificando valores en variables, mientras que aquí al estar representado cada estado.
- Los objetos State pueden ser compartidos si no contienen variables de instancia, esto se puede lograr si el estado que representan esta enteramente codificado en su tipo. Cuando se hace esto estos estados son Flyweights sin estado intrínseco.
- Facilita la ampliación de estados
- Permite a un objeto cambiar de clase en tiempo de ejecución dado que al cambiar sus responsabilidades por las de otro objeto de otra clase la herencia y responsabilidades del primero han cambiado por las del segundo.

Se encuentran la siguiente desventaja:

- Se incrementa el número de subclases.

Patrón Strategy

Propósito: El patrón estrategia permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades.

Patrones de Diseño

Motivación/Problema: Suponiendo un editor de textos con diferentes algoritmos para particionar un texto en líneas (justificado, alineado a la izquierda, etc.), se desea separar las clases clientes de los diferentes algoritmos de partición, por diversos motivos:

- Incluir el código de los algoritmos en los clientes hace que éstos sean demasiado grandes y complicados de mantener y/o extender.
- El cliente no va a necesitar todos los algoritmos en todos los casos, de modo que no queremos que dicho cliente los almacene si no los va a usar.
- Si existiesen clientes distintos que usasen los mismos algoritmos, habría que duplicar el código, por tanto, esta situación no favorece la reutilización.

Patrones de Diseño

Consideraciones:

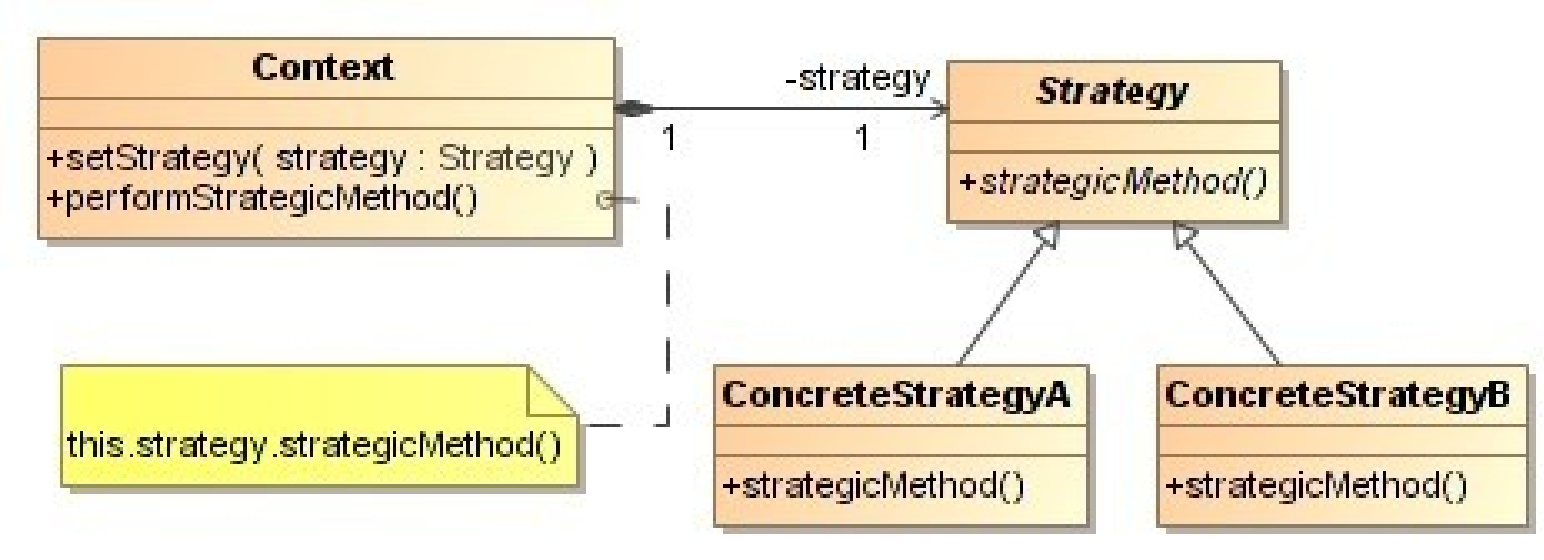
Se utiliza este patrón cuando:

- Muchas clases relacionadas difieren sólo en su comportamiento.
- Se necesita diferentes variantes de un algoritmo.
- Se presentan muchos condicionales en el código, es posible que sea necesario aplicar este patrón.
- Si un algoritmo utiliza información que no deberían conocer los clientes, la utilización del patrón estrategia evita la exposición de dichas estructuras.

Solución: La solución que el patrón estrategia supone para este escenario pasa por encapsular los distintos algoritmos en una jerarquía y que el cliente trabaje contra un objeto intermediario contexto. El cliente puede elegir el algoritmo que prefiera de entre los disponibles, o el mismo contexto puede ser el que elija el más apropiado para cada situación.

Patrones de Diseño

Estructura UML:



Patrones de Diseño

Participantes:

- Contexto (Context) : Es el elemento que usa los algoritmos, por tanto, delega en la jerarquía de estrategias. Configura una estrategia concreta mediante una referencia a la estrategia necesaria.

- Estrategia (Strategy): Declara una interfaz común para todos los algoritmos soportados. Esta interfaz será usada por el contexto para invocar a la estrategia concreta.

- Estrategia Concreta (ConcreteStrategy): Implementa el algoritmo utilizando la interfaz definida por la estrategia.

Patrones de Diseño

Colaboraciones:

Es necesario el intercambio de información entre estrategia y contexto. Este intercambio puede realizarse de dos maneras:

- Mediante parámetros en los métodos de la estrategia.
- Mandándose, el contexto, a sí mismo a la estrategia.

Los clientes del contexto lo configuran con una estrategia concreta. A partir de ahí, solo se interactúa con el contexto.

Patrones de Diseño

Ventajas y Desventajas:

•La herencia puede ayudar a factorizar las partes comunes de las familias de algoritmos (sustituyendo el uso de bloques de instrucciones condicionales). En este contexto es común la aparición conjunta de otros patrones como el patrón template method.

• El uso del patrón proporciona una alternativa a la extensión de contextos, ya que puede realizarse un cambio dinámico de estrategia.

• Los clientes deben conocer las diferentes estrategias y debe comprender las posibilidades que ofrecen.

• Como desventaja, aumenta el número de objetos creados, por lo que se produce una penalización en la comunicación entre estrategia y contexto (hay una indirección adicional).

Patrón Singleton

Propósito: garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella

Motivación/Problema:

Se utiliza este patrón cuando:

- debe haber exactamente una instancia de una clase, y debe ser accesible a los clientes desde un punto de acceso conocido.
- la instancia única deba ser extensible por medio de subclases, y los clientes deban ser capaces de utilizar una instancia extendida sin modificar su código.

Consideraciones:

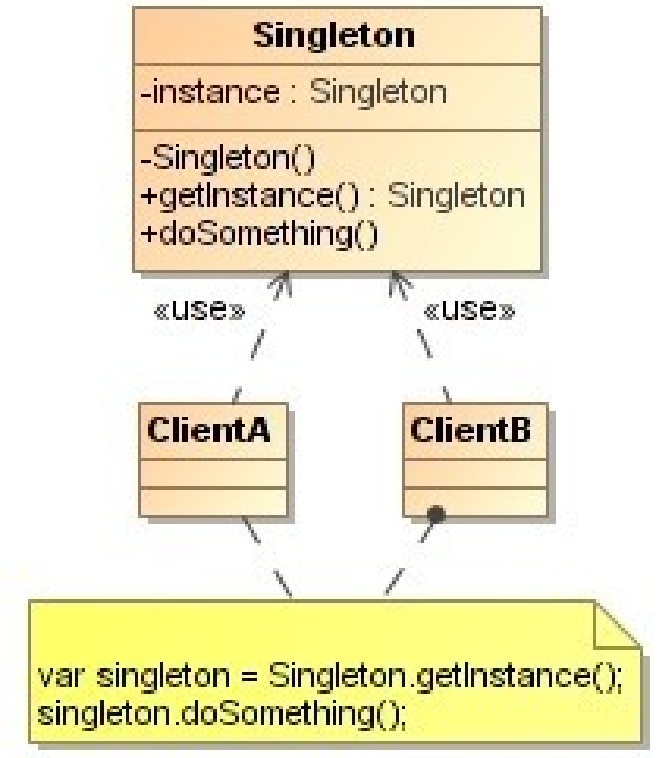
La instrumentación del patrón puede ser delicada en programas con múltiples hilos de ejecución. Si dos hilos de ejecución intentan crear la instancia al mismo tiempo y esta no existe todavía, sólo uno de ellos debe lograr crear el objeto. La solución clásica para este problema es utilizar exclusión mutua en el método de creación de la clase que implementa el patrón.

Solución:

- Asegurar una única instancia.
  - Sobreescribiendo el constructor.
  - Teniendo una referencia a si mismo.
  - Creando un método que devuelva la única instanca.
- Elegir una subclase del singleton
  - mediante la inicialización de la variable -instance, con una instancia de la subclase a utilizar.
  - agregando una variable de entorno que permita a getInstance saber, mediante estructuras condicionales, qué tipo de subclase debe instanciar.
  - tomando la implementación de getInstance() de la clase padre y ponerla en la subclase.
  - registrando cada subclase Singleton bajo un nombre determinado dentro de un registro. Cuando se necesita un singleton, getInstance() consulta el registro solicitando el singleton por su nombre. El registro busca la correspondiente singleton (si existe) y lo devuelve. Este enfoque libera getInstance() de conocer todas las subclases Singleton posibles.

Patrones de Diseño

Estructura UML:



Patrones de Diseño

Participantes:

• Singleton: es el responsable de crear y mantener una única referencia a si mismo. Define una operación getInstance que permite a los clientes acceder a su instancia única. Su constructor debe ser escondido de la interfaz pública (protegido o privado), para que la clase no pueda ser directamente instanciada, o sobreescrito y que llame al método getInstance.

Colaboraciones:

Client-Singleton: los clientes sólo pueden instanciar la clase Singleton a través de su método getInstance.

Ventajas y desventajas:

• Controla el acceso a su única instancia.

• Evita contaminar el espacio de nombres de variables globales que almacenan los casos individuales.

• La clase Singleton puede tener subclases, y es fácil de configurar un cliente con una instancia de esta subclase.

• Permite un número variable de instancias, utilizando el mismo enfoque. Sólo tiene que cambiar el método getInstance.

•Más flexible que clases estáticas, ya que el Singleton puede ser redefinido por las subclases y las clases estáticas no.

Patrones de Diseño

Patrón Factory Method

Propósito: Factory Method define una interfaz para crear objetos, pero deja que sean las subclases quienes decidan qué clases instanciar; permite que una clase delegue en sus subclases la creación de objetos.

Motivación: los Frameworks usan clases abstractas para definir y mantener relaciones entre objetos. Un framework es tambien responsable de crear esos objetos.

Consideremos un framework de aplicaciones que pueden presentar multiples documentos al usuario. Las abstracciones clave en este framework son las clases aplicación y documento. Ambas clases son abstractas, y los clientes tienen que “subclasearlas” para realizar sus implementaciones específicas de sus aplicaciones. Para crear una aplicación que las grafique, por ejemplo, definimos las clases DrawingApplication y DrawingDocument. La clase Aplicacion es responsable de administrar Documentos y los creara cuando se requieran—por ejemplo, cuando los usuarios seleccionen abrir o nuevo de un menu.

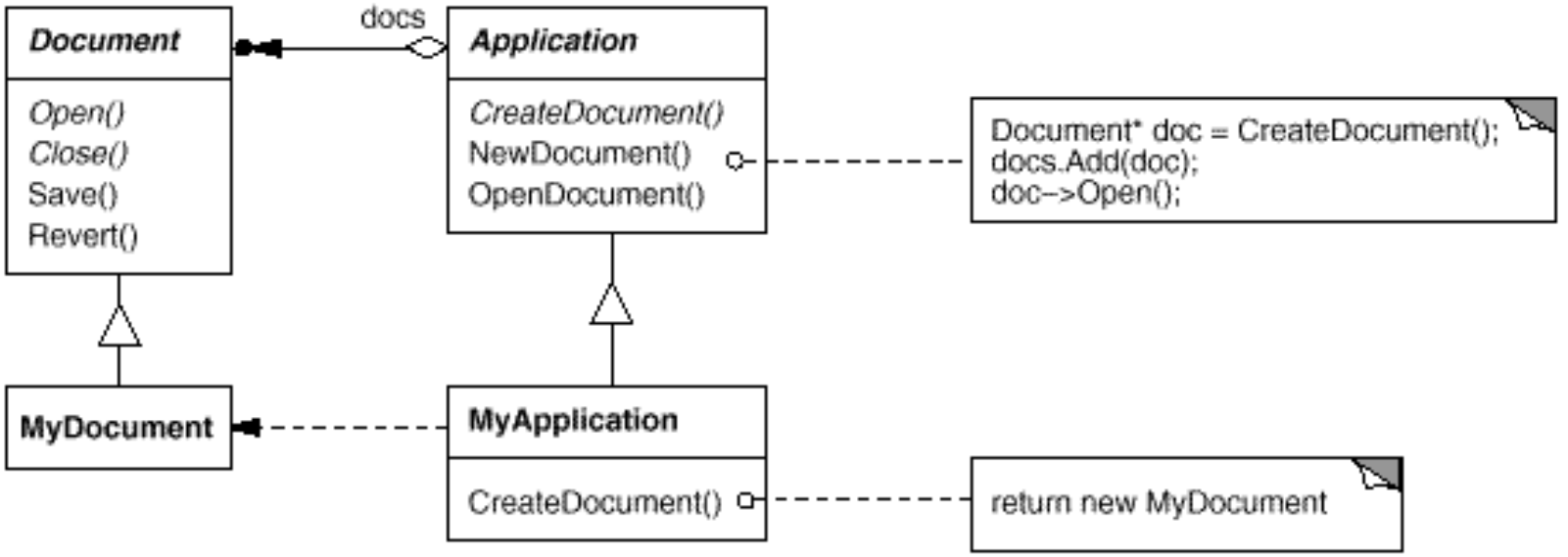
Patrones de Diseño

Como la subclase Documento particular a instanciar es especifica de la aplicación, la clase aplicación no puede predecir la subclase del documento a instanciar— La clase Aplicacion solo sabe cuando un nuevo documento debe ser creado, no que tipo de documento crear. Esto crea un dilema: El framework debe instanciar clases, pero solo conoce clases abstractas, las cuales no pueden ser instanciadas.

El patrón Factory Method ofrece una solución. Encapsula el conocimiento de que subclase de documento crear y mueve este conocimiento fuera del framework.

La subclases de Aplicacion redefinen un método abstracto CreateDocument en la Aplicación para retornar la subclase de documento apropiada. Una vez que una subclase de aplicación es instanciada, entonces puede instanciar documentos específicos de la aplicación sin conocer sus clases. Llamamos CreateDocument un factory method porque es responsable de "manufacturar" un objeto.

Patrones de Diseño



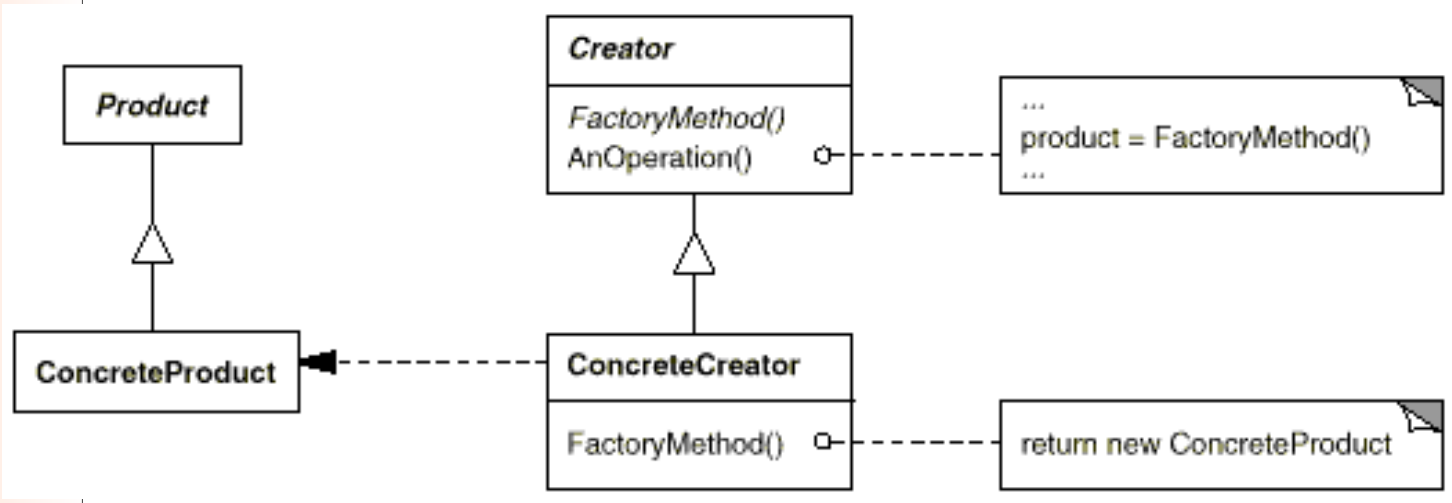
Consideraciones:

Usar 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 queremos localizar concretamente en qué subclase de auxiliar se delega.

Patrones de Diseño

Solución:
Estructura UML:



Patrones de Diseño

Participantes:

- **Producto**, define la interfaz de los objetos que crea el método de fabricación.
- **Creador**, declara el método de fabricación, el cual devuelve un objeto del tipo Producto. También puede definir una implementación predeterminada del método de fabricación que devuelve un objeto ProductoConcreto. Puede llamar al método de fabricación para crear un objeto Producto.
- **ProductoConcreto**, implementa la interfaz Producto.
- **CreadorConcreto,** redefine el método de fabricación para devolver una instancia de ProductoConcreto.

Colaboraciones:

- Creador delega en sus subclases la definicion del factory method de modo que este retorne una instancia del ProductoConcreto apropiado.

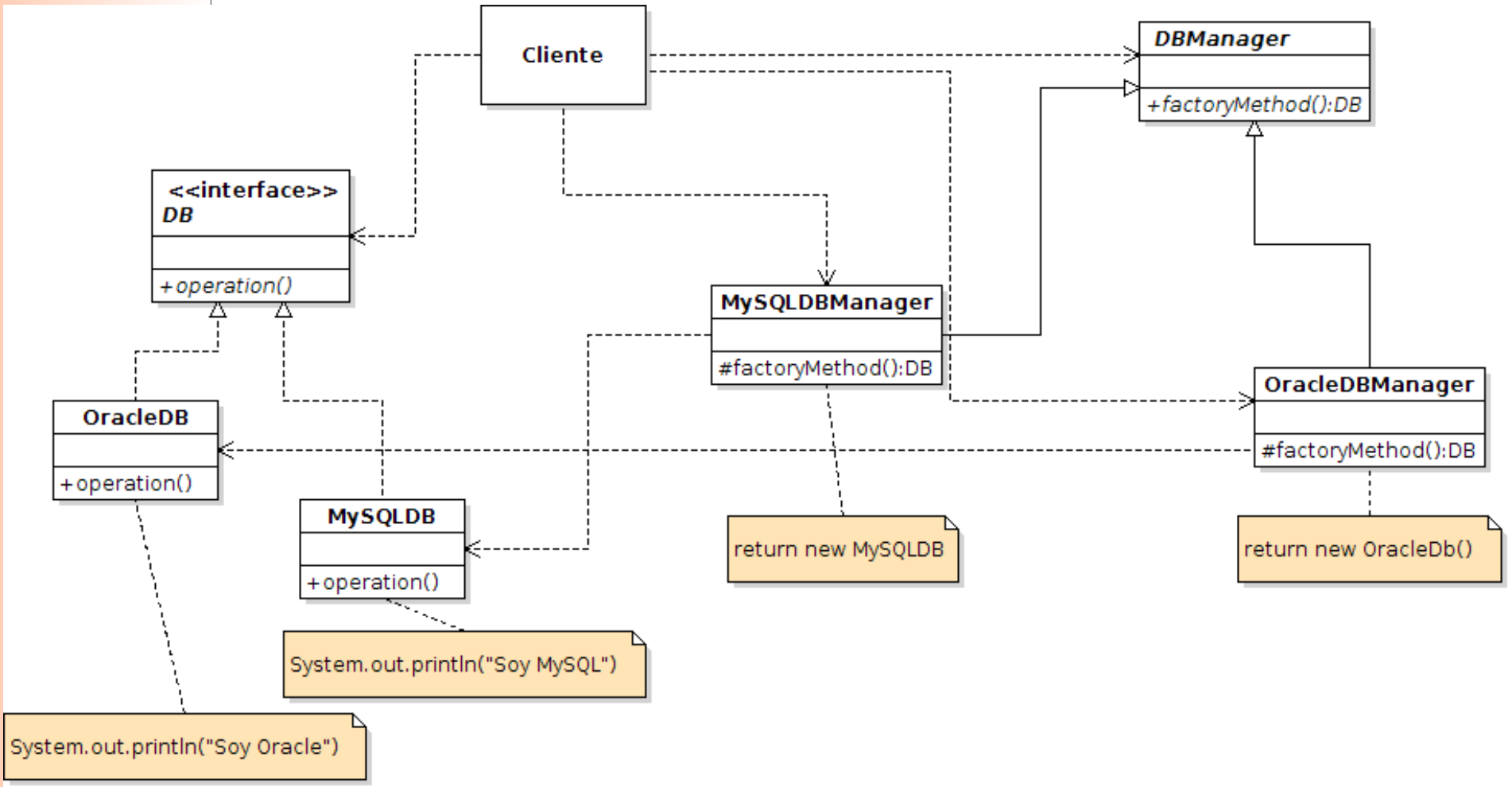
Patrones de Diseño

Ventajas y desventajas:

- Proporciona enganches para las subclases. Crear objetos dentro de una clase con un método de fabricación es siempre más flexible que hacerlo directamente. Conecta jerarquías de clases paralelas.
- Elimina la necesidad de acoplar las clases en el código. El código interactúa con interfaces entonces puede trabajar con cualquier clase que implemente cierta interfaz.
- Permite a las subclase para proveer una versión extendida de un objeto, porque crear un objeto dentro de una clase es mas flexible que crear un objeto directamente en el cliente.

Patrones de Diseño

Ejemplo:



Patrones de Diseño

Creador:

```
public abstract class DBManager {
 public abstract DB factoryMethod();
}
```

CreadorConcreto:

```
public class MySQLDBManager extends DBManager {
 protected DB factoryMethod() {
       return new MySQLDB();
 }
}
public class OracleDBManager extends DBManager {
 protected DB factoryMethod() {
       return new OracleDB();
 }
}
```

Patrones de Diseño

Producto:

```
public interface DB {
 public void operation();
}
```

ProductoConcreto:

```
public class MySQLDB implements DB {
 public void operation() {
       System.out.println("Soy MySQL");
 }
}
public class OracleDB implements DB {
 public void operation() {
       System.out.println("Soy Oracle");
 }
}
```

Patrones de Diseño

Main:

```
public class Cliente {
public static void main(String[] args) {
 DBManager manager = new MySQLDBManager();
 DB mysql = manager.factoryMethod();
 mysql.operation();

 manager = new OracleBDManager();
 DB oracle = manager.factoryMethod();
 oracle.operation();
}
}
```

Patrones de Diseño

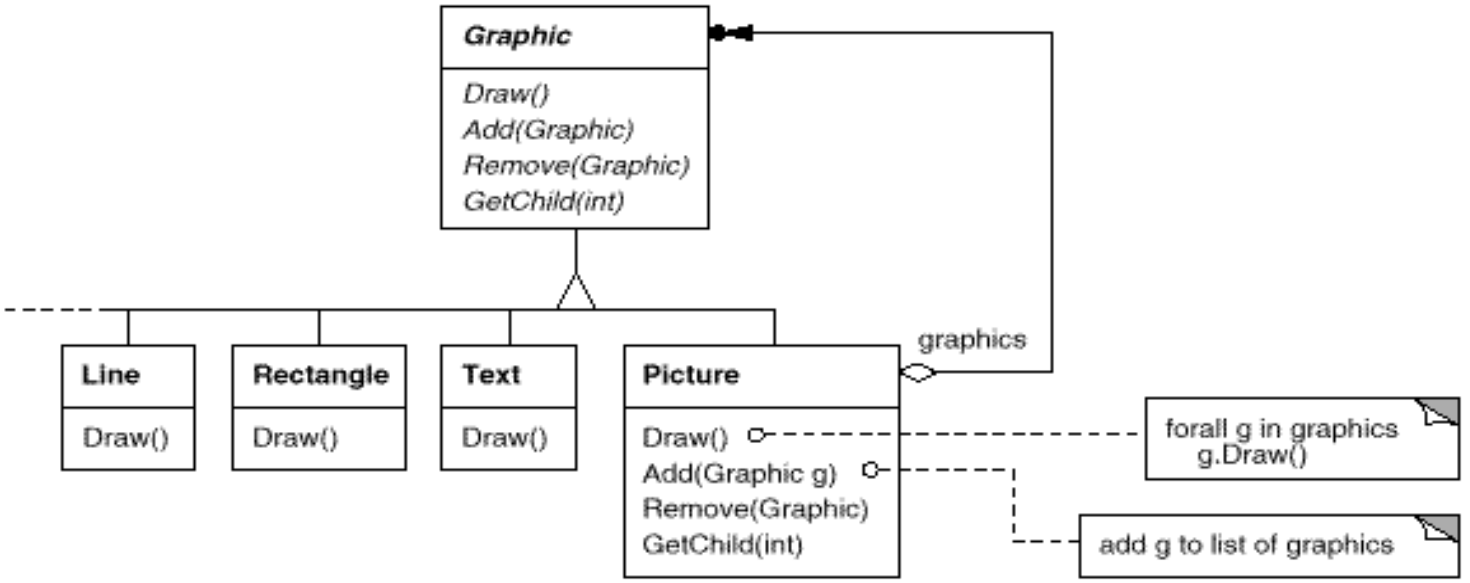
Patrón Composite

Propósito: 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.

Motivación/Problema: Los editores gráficos permiten a los usuarios construir componentes complejos a partir de componentes simples. El usuario puede agrupar componentes para construir componentes mas grandes, los cuales a su vez pueden ser agrupados para construir objetos aún mas grandes. Una implementación simple puede definir clases para primitivas gráficas tales como texto y lineas, y otras clases que actuen como contenedores para estas primitivas.

Pero hay un problema con esta solución: El código que usa estas clases debe tratar las primitivas y contenedores de forma diferente, aún si la mayor parte del tiempo los usuarios las tratan de forma identica. Tener que distinguir estos objetos hace a la aplicación mas compleja. El patrón Composite describe como utilizar composición recursiva para que los clientes no tengan que hacer esta distinción.

Patrones de Diseño



Consideraciones:

Utilizar el patrón Composite cuando

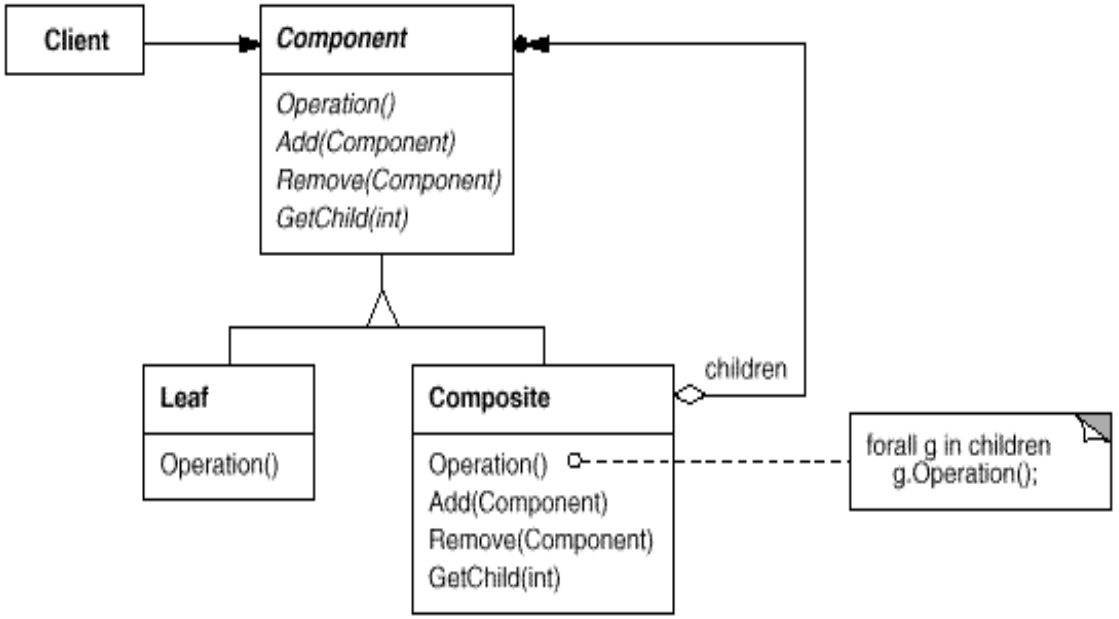
- Se quiere representar jerarquías de objetos parte-todo.
- Se quiere que los clientes puedan ignorar las diferencias entre composiciones de objetos y objetos individuales. Los clientes trataran a todos los objetos de la estructura compuesta de manera uniforme.

Patrones de Diseño

Solución: La clave del patrón composite es la clase abstracta Component que representa ambas primitivas y sus contenedores.

Component declara las operaciones comunes entre primitivas y contenedores y ademas declara las operaciones que todos los objetos compuestos comparten tales como operaciones para acceder y manejar sus hijos.

Estructura UML:



Patrones de Diseño

Participantes:

Component: Declara la interfaz para los objetos de la composicion, es la interfaz de acceso y manipulación de los componentes hijo e implementa algunos comportamientos por defecto.

Client: Manipula objetos a través de la interfaz proporcionada por Component.

Composite: Define el comportamiento de los componentes compuestos, almacena a los hijos e implementa las operaciones de manejo de los componentes.

Leaf: Definen comportamientos para objetos primitivos del compuesto.

Colaboraciones:

Los clientes usan la interfaz Component para interactuar con objetos en la estructura compuesta. Si el receptor es una hoja(Leaf), entonces el pedido es atendido en forma directa. Si el receptor es un Composite, entonces este delega la solicitud a sus componentes hijos, y puede realizar operaciones adicionales antes y después de delegar la solicitud.

Patrones de Diseño

Ventajas y desventajas:

- Permite a los clientes manejar estructuras complejas y simples de manera uniforme.
- El cliente no debe preocuparse por saber con que tipo de objetos está tratando
- Es fácil agregar nuevos componentes

Desventaja:

- Puede ser difícil restringir los componentes de los objetos compuestos.

Patrones de Diseño

Patrón Decorator

Propósito: Agrega responsabilidades adicionales a un objeto en forma dinámica Los decoradores proveen una alternativa flexible a utilizar herencia para extender la funcionalidad.

Motivación/Problema: Se puede querer agregar responsabilidades a objetos individuales, no a toda una clase. Un toolkit de interfaz gráfica del usuario, por ejemplo, debe permitir agregar propiedades como bordes o comportamientos tales como desplazamiento a cualquier componente de la interfaz del usuario.

Una forma de agregar responsabilidades es con la herencia. Heredando un borde de otra clase pone un borde alrededor de ca instancia de una subclase. Esto no es flexible, porque la elección del borde es realizada de forma estática Un cliente no puede controlar como y cuando decorar el componente con un borde.

Patrones de Diseño

Una solución mas flexible es encapsular el componente en otro objeto que agregue el borde. El objeto que encapsula el objeto es llamado decorador. El decorador conforma la interface del componente que decora de forma que su presencia es transparente a los componentes de los clientes. El decorador reenvia las solicitudes a el componente y debe realizar acciones adicionales (tales como dibujar un borde) antes o despues de reenviar la solicitud. La transparencia permite anidar decoradores en forma recursiva, permitiendo un numero ilimitado de responsabilidades adicionales.

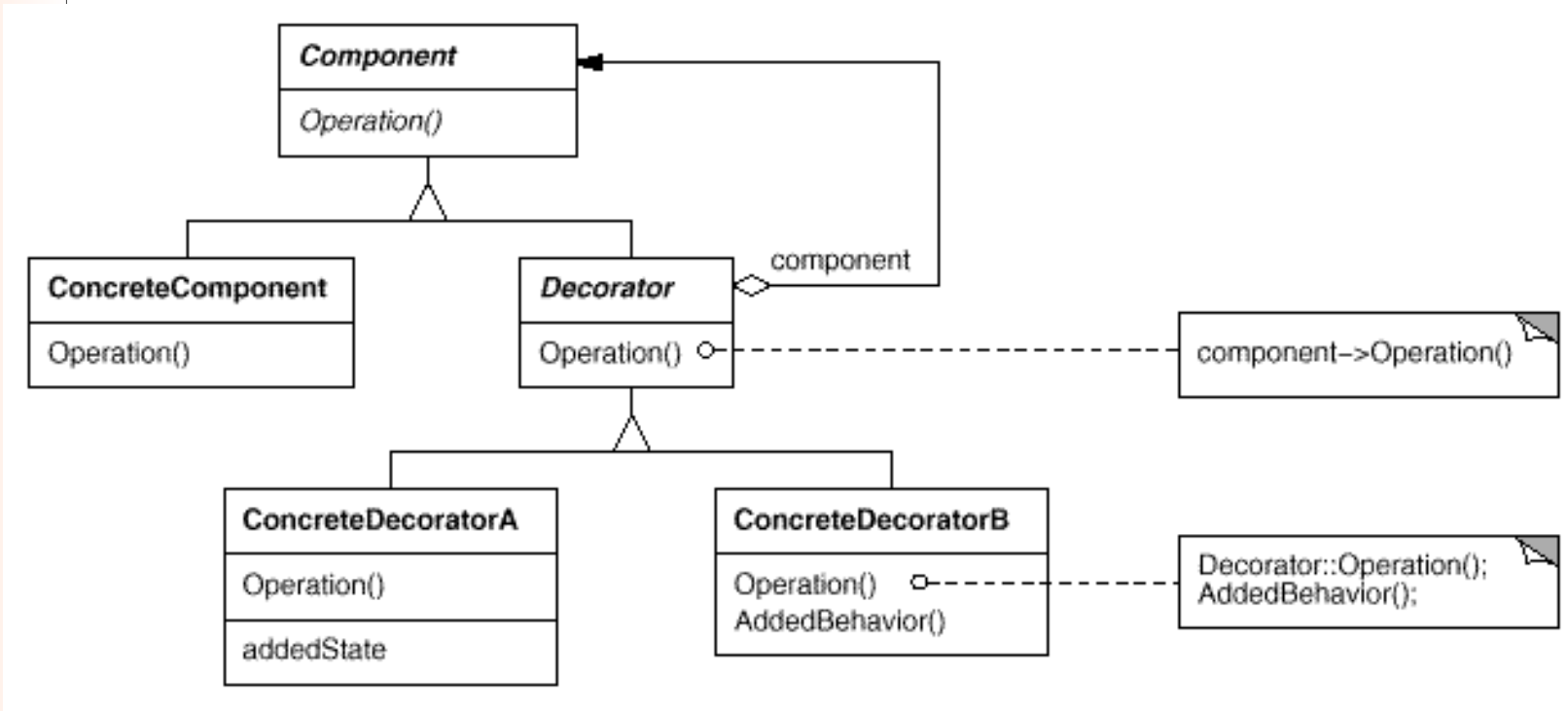
Consideraciones:

Aplicar el patrón decorator cuando,

- Se quiere añadir responsabilidades a objetos individuales de forma dinámica y transparente
- Las responsabilidades de un objeto pueden ser retiradas
- Cuando la extensión mediante la herencia no es viable.
- Hay una necesidad de extender la funcionalidad de una clase, pero no hay razones para extenderlo a través de la herencia.
- Existe la necesidad de extender dinámicamente la funcionalidad de un objeto y quizás quitar la funcionalidad extendida.

Patrones de Diseño

Solución:
Estructura UML:



Patrones de Diseño

Participantes:

- **Componente,** Define la interfaz para los objetos que pueden tener responsabilidades añadidas.
- **Componente Concreto,** Define un objeto al cual se le pueden agregar responsabilidades adicionales.
- **Decorador,** Mantiene una referencia al componente asociado. Implementa la interfaz de la superclase Componente delegando en el componente asociado.
- **Decorador Concreto,** Añade responsabilidades al componente.

Colaboraciones:

- El decorador redirige las peticiones al componente asociado.
- Opcionalmente puede realizar tareas adicionales antes y después de redirigir la petición.

Patrones de Diseño

Ventajas y desventajas:

- Más flexible que la herencia. Al utilizar este patrón, se pueden añadir y eliminar responsabilidades en tiempo de ejecución. Además, evita la utilización de la herencia con muchas clases y también, en algunos casos, la herencia múltiple.
- Evita la aparición de clases con muchas responsabilidades en las clases superiores de la jerarquía. Este patrón nos permite ir incorporando de manera incremental responsabilidades.
- Genera gran cantidad de objetos pequeños. El uso de decoradores da como resultado sistemas formados por muchos objetos pequeños y parecidos.
- Puede haber problemas con la identidad de los objetos. Un decorador se comporta como un envoltorio transparente. Pero desde el punto de vista de la identidad de objetos, estos no son idénticos, por lo tanto no deberíamos apoyarnos en la identidad cuando estamos usando decoradores.