Patrones Creacionales – adaptación

Objetivo

- Proporcionar una práctica que permita reforzar los conceptos vistos, para obtener una mayor claridad del uso de patrones de diseño creacionales.
- Utilizar modelamiento UML para comprobar la estructura de los patrones planteados.

Introducción

Algunas personas definen un patrón como una solución recurrente para un problema en un contexto. Estos términos -- contexto, problema y solución -- merecen una pequeña explicación. Primero, ¿qué es un contexto? Un contexto es el entorno, situación, o condiciones interrelacionadas dentro de las cuales existe algo. Segundo, ¿qué es un problema? Un problema es una cuestión insatisfecha, algo que se necesita investigar y resolver. Un problema se puede especificar mediante un conjunto de causas y efectos.
Normalmente un problema está restringido al contexto en el que ocurre. Finalmente, la Solución se refiere a la respuesta al problema dentro de un contexto que ayuda a resolver las dificultades.
Entonces, si tenemos una solución a un problema en un contexto, ¿es un patrón? No necesariamente. También necesitamos asociar la característica de recurrencia con la Definición de un patrón. ¿Eso es todo? Quizás no. Los patrones deberían comunicar Soluciones de diseño a los desarrolladores y arquitectos que los leen y los utilizan. Como puede ver, aunque el concepto de patrón es bastante simple, definir realmente el término es muy complejo.
Hemos señalado sólo las referencias para que puedas indagar en más profundidad en la historia de los patrones y aprender sobre ellos en otras áreas. Sin embargo, debería tener en mente que la definición de patrón que hemos adoptado funciona. En nuestro catálogo, se describe un patrón de acuerdo a sus principales características: contexto, problema y solución, junto con otros aspectos importantes, como causas y consecuencias.
Un patrón describe, con algún nivel de abstracción, una solución experta a un problema.
Normalmente, un patrón está documentado en forma de una plantilla. Aunque es una práctica estándar documentar los patrones en un formato de plantilla especializado, esto no significa que sea la única forma de hacerlo. Además, hay tantos formatos de plantillas como autores de patrones, esto permite la creatividad en la documentación de patrones.
Los patrones solucionan problemas que existen en muchos niveles de abstracción. Hay patrones que describen soluciones para todo, desde el análisis hasta el diseño y desde la arquitectura hasta la implementación. Además, los patrones existen en diversas áreas de interés y tecnologías. Por ejemplo, hay un patrón que describe como trabajar con un lenguaje de programación específico o un segmento de la industria específico, como la medicina.

Patrones de creación: Estos patrones facilitan una de las tareas más comunes en la programación orientada a objetos: la creación de objetos en un sistema. La mayoría de sistemas OO, sea cual sea su complejidad, necesitan crear instancias de muchas clases, y estos patrones facilitan el proceso de creación ayudando a proporcionar las siguientes capacidades:

1. Instanciación genérica: permite crear objetos en un sistema sin identificar la clase específica en el código.
2. Simplicidad: algunos de los patrones facilitan la creación de objetos, por lo que los invocadores no tendrán que escribir grandes y complejos códigos para crear un objeto.
3. Restricciones de creación: algunos patrones fuerzan restricciones en el tipo o en el número de objetos que pueden crearse en el sistema.

Los patrones de creación GoF son los siguientes.

- **Abstract Factory** (Factoría Abstracta): proporciona un contrato para la creación de familias de objetos relacionados o dependientes sin tener que especificar su clase concreta.
- **Builder** (Constructor): simplifica la creación de objetos complejos definiendo una clase cuyo propósito es construir instancias de otra clase. Aunque puede haber mas de una clase en el producto, este constructor genera un producto principal porque siempre debe existir una clase principal.
- **Factory Method**: (Método de fabricación): permite definir un método estándar para crear un objeto, ademas del constructor propio de la clase, si bien el objeto a crear se delega a las subclases.
- **Prototype** (Prototipo): facilita la creación dinámica al definir clases cuyos objetos pueden crear copias de si mismos.
- **Singleton** (Único): permite crear una única instancia de esta clase en el sistema, a la vez permite que todas las clases tengan acceso a esta instancia.

De estos patrones, Abstract Factory y Factory Method se basan explícitamente en el concepto de definir la creación flexible de objetos; supone que en el sistema habrán subclases que hereden, o deriven, de las clases o interfaces que se crean. Como resultado, estos dos patrones se combinan frecuentemente con otros patrones de creación.

FORMATO DE DESCRIPCION DE PATRON DE DISEÑO	
Nombre del Patrón	Abstract Factory
También conocido como: Kit, ToolKit (Grupo de herramientas)	
Propiedades del Patron: De creación, Objeto. Nivel: Componente	
Propósito: Proporciona un contrato para crear familias de objetos relacionados o dependientes sin tener que especificar su clase concreta.	
Introducción: Suponga que tiene algunas clases en su sistema que son propensas a variar de acuerdo a nuevos tipos de implementación, por ejemplo si estuviese creando un programa en el que se adicionan clases para representar un mismo concepto en diferentes países. Una alternativa es pensar en todas las posibilidades y crear una clase con lógica suficiente para atender todas las nuevas características, o por otro lado, tendría que parar el sistema y modificar la clase cada vez que se den cambios. En vez de tener que adicionar lógica funcional a las clases, se crea una clase principal y se extiende a los casos particulares. Las instancias de estas clases concretas se crean a partir de una fábrica. Esto da una mayor libertad para extender el código sin realizar grandes modificaciones en la estructura del resto del sistema.	

Aplicabilidad

Use el patrón Abstract Factory cuando:

1. El cliente debe ser independiente del proceso de creación de los productos.
2. La aplicación debe configurarse con una o más familias de productos.
3. Es necesario crear los objetos como un conjunto, de forma que sean compatibles.
4. Desea proporcionar una colección de clases y únicamente quiere revelar sus contratos y sus relaciones, no sus implementaciones.

Descripción:

Algunas veces, una aplicación necesita utilizar varios recursos o sistemas operativos. Algunos ejemplos incluyen:

1. Gestión de ventanas
2. Un sistema de archivos.
3. Comunicación con otras aplicaciones o sistemas.

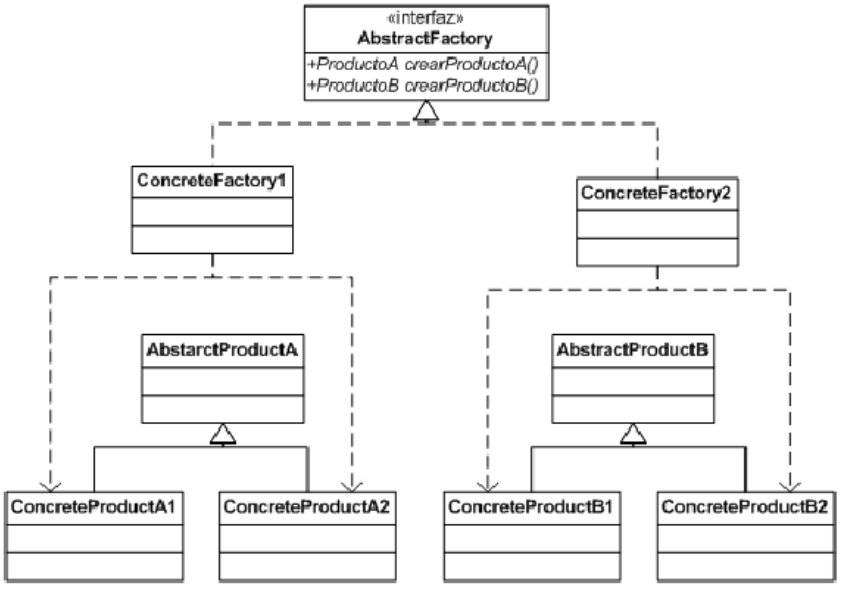
Es deseable que este tipo de aplicaciones sena los suficientemente flexibles como para ser capaces de utilizar diferentes recursos sin tener que re escribir el código de la aplicación cada vez que la aplicación introduce un nuevo recurso.

Una forma efectiva de resolver este problema es definir un creador de recursos genéricos, es decir, el patrón Abstract Factory. La fábrica tiene uno más métodos de creación, que pueden ser llamados para producir recursos genéricos o productos abstractos.

Java se ejecuta en muchas maquinas y cada una puede pertenecer a una plataforma diferente. Cada una de estas plataformas tiene implementaciones de sistemas de archivos o un sistema de ventanas. La solución que ha tomado java es abstraer los conceptos de archivo y ventana sin mostrar la implementación concreta. Se puede desarrollar una aplicación utilizando las capacidades genéricas de los recursos como si representaran funciones reales.

En tiempo de ejecución, la aplicación crea y utiliza objetos ConcreteFactory y ConcreteProduct. Las clases concretas son compatibles con AbstractFactory y AbstractProduct, de forma que pueden utilizar directamente las clases concretas, sin tener que reescribirlas o recompilarlas.

Implementacion:



Normalmente, los siguientes elementos son los que se utilizan para implementar el patrón Abstract Factory:

1. AbstractFactory: la clase abstracta o interfaz que define los métodos de creación de

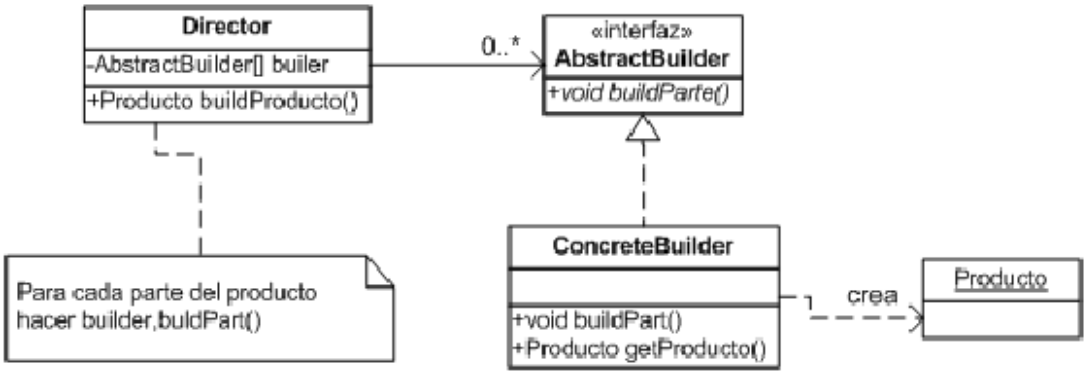
los productos abstractos.
2. AbstractProduct: la clase abstracta o interfaz que describe el comportamiento general de los recursos utilizados por la aplicación.
3. ConcreteFactory: la clase derivada de la fábrica abstracta. Implementa los métodos para crear uno o más productos concretos.
4. ConcreteProduct: la clase derivada de AbstractProduct, que proporciona implementación para un recurso o entrono operativo específicos.

Ventajas e Inconvenientes:
Una fabrica abstracta ayuda a incrementar la flexibilidad general de una aplicación. Esta flexibilidad se manifiesta tanto en tiempo de ejecución como en tiempo de compilación.
Durante el diseño, no hay que predecir todos los usos futuros de la aplicación. En su lugar, se crea un Framework general y entonces se desarrollan implementaciones independientes del resto de la aplicación. En tiempo de ejecución, la aplicación puede integrar fácilmente

FORMATO DE DESCRIPCION DE PATRON DE DISEÑO	
Nombre del Patrón	Builder
También conocido como: Constructor Especializado	
Propiedades del Patron: Tipo. De creación, Objeto. Nivel: Componente	
Propósito: Simplificar la creación de objetos complejos definiendo una clase cuyo propósito sea construir instancias de otra clase. Aunque puede haber más de una clase en el producto, el patrón Builder genera un producto principal porque siempre debe existir una clase principal.	
Introducción: En el proceso de construcción de un objeto pueden existir diferentes tipos de información necesarias lo cual hace que este proceso sea complejo. Particularmente, el objeto puede requerir que otros objetos y precondiciones estén dados para poder ser creado. Se puede entonces delegar a una clase el proceso de construcción de las partes necesarias para obtener una instancia estable e incluso se pueden allí validar el conjunto de precondiciones que hacen viable la utilización de la instancia.	
Aplicabilidad Utilice el patron Builder si una clase: 1. Tiene una estructura interna compleja (especialmente si tiene un conjunto variable de objetos relacionados). 2. Tiene atributos dependientes entre si. Una de las cosas que puede hacer el patrón Builder es forzar la construcción por etapas de un objeto complejo. Esto será necesario cuando los atributos del "Producto" son interdependientes. Por ejemplo, suponga que esta construyendo un pedido de compra. Necesitaría asegurarse de que esta en un estado concreto antes de construir el método de compra, porque el estado impactaría en otras partes del proceso como son los impuestos de venta aplicados al pedido. 3. Utiliza otros objetos del sistema que serian difíciles, o poco convenientes de obtener durante la creación.	
Descripción: Este patrón se llama Builder porque esta relacionado con la construcción de objetos complejos, posiblemente de diferentes fuentes. Conforme aumenta la complejidad de	

creación de objetos, gestionarla desde el método constructor puede ser difícil. Esto es especialmente cierto si el objeto depende exclusivamente de los recursos que están bajo su control.
Los objetos de negocio a menudo entran dentro de esta categoría. Frecuentemente necesitan datos de una base de datos durante la inicialización y necesitan asociarse con otros objetos de negocio para representar de forma adecuada el modelo de negocio. Otro ejemplo es el de los objetos compuestos de un sistema, como el objeto que representa un dibujo en un programa de edición grafica. Un objeto de este tipo necesitaría estar relacionado con un número de objetos tan pronto como fuera creado.
En estos casos, es conveniente definir otra clase (que será el objeto Builder o constructor) que fuera responsable de la construcción. El objeto constructor coordina la combinación del objeto producto: creación de recursos, almacenamiento de resultados intermedios y dotación de la estructura funcional ara la creación. Además, el objeto constructor puede adquirir recursos del sistema para la generación del objeto producto.

Implementacion:



Para la implementación del patrón Builder se necesita:
1. Director: tiene una eferencia a una instancia de AbstractBuilder. El Director llama a los métodos de creación de su instancia concreta de AbstractBuilder para tener todas las partes necesarias y poder así construir el objeto.
2. AbstractBuilder (Constructor Abstracto): es la interfaz que define los métodos disponibles para crear las distintas partes del producto.
3. ConcreteBuilder (Constructor Concreto): la clase que implementa la interfaz AbstractBuilder. La clase ConcreteBuilder implementa todos los métodos necesarios para crear un objeto Producto real. La implementacion de los métodos sabe como procesar la información del Director y como construir las respectivas partes del producto. La case ConcreteBuilder también tiene un método getProducto o un método de creación que devuelve una instancia de Producto
4. Producto: es el objeto resultante. Pude definir el producto como una interfaz (Opción preferible) o una clase.

Ventajas e Inconvenientes:
El patrón Builder facilita la gestión del flujo de control durante la creación de objetos complejos. Esto se manifiesta de dos formas:
1. Para objetos que necesitan una creación en fases (una secuencia de pasos para lograr que el objeto se active), el objeto Builder actúa como un objeto de alto nivel que supervisa el proceso. Puede coordinar y validar la creación de todos los recursos, y si fuera necesario, proporcionar, proporcionar una estrategia de emergencia en caso de que ocurra un error.

2. Para objetos que durante la creación necesitan de recursos existentes en el sistema, como conexiones a bases de datos u objetos del negocio existentes, el objeto builder proporciona también un único punto de acceso centralizado para gestionar esos recursos. Builder también proporciona un único punto de control de creación para sus productos, que puede ser utilizado por otros objetos del sistema. Al igual que oros patrones de creación, esto facilita las cosas para los clientes de sistemas software, porque solo necesitan acceder al objeto Builder para obtener un recurso.
El principal inconveniente de este patrón es que hay un alto grado de acoplamiento entre el objeto Builder, su producto y cualquier otro delegado para la creación utilizado durante la construcción de objetos. Los cambios que sucedan en el Producto creado por Builer a menudo generan modificaciones para el objeto Builder como para sus delegados.

Variaciones del Patrón:
En el nivel más elemental, es posible implementar un patrón Builder básico con una clase Builder que tenga un método de creación y su producto. Para dotarlo de mayor flexibilidad, los diseñadores suelen ampliar este patrón con una o mas de las siguientes aproximaciones:
1. Crear un Builder abstracto. Puede tener un sistema más genérico que sea capaz de albergar muchos tipos distintos de constructores si define una clase abstracta o una Interfaz que especifique los métodos de creación.
2. Definir múltiples métodos de creación para el Builder. Algunos Builder definen varios métodos de creación (básicamente sobrecargan su método de creación) para proporcionar diferentes formas de inicializar el recurso construido.
3. Crear delegados para la creación. Con esta variante, un objeto Director contiene el método de creación general de Producto y llama a una serie de métodos mas granulares en el objeto Builder. En este caso, el objeto Director actúa como un gestor para el proceso de creación del Builder.

Patrones relacionados:
Los patrones relacionados a Builder son: Composite (Compuesto). El patron Builder se utiliza a menudo para producir objetos Composite, porque suelen tener una estructura compleja.

FORMATO DE DESCRIPCION DE PATRON DE DISEÑO	
Nombre del Patrón	Factory Method
También conocido como: Virtual Constructor (Constructor Virtual)	
Propiedades del Patrón: Tipo: Creación, Nivel: Clase	
Propósito: Permite definir un método estándar para crear un objeto, además del constructor propio de la clase, si bien la decisión del objeto a crear es delegada en las subclases.	
Introducción: Cuando una aplicación esta compuesta por partes y estas partes tienen a su vez otras, es difícil el anexar una nueva parte y controlarla desde el objeto inicial donde está contenido. El problema se torna más complejo cuando es necesario modificar alguna de las partes interiores ya que el cambio se propagaría para tener que realizar también cambios en la parte contenedora general. Una posible solución es que las partes internas tengan facultades para administrar sus propios cambios y mediante un llamado el objeto principal obtenga una instancia interna que devuelve su estado ya estable. Esta solución delega a las partes la administración de su estado y facilita la disminución de la complejidad en el objeto contenedor	

Aplicabilidad

Utilice el patrón Factory Method cuando:

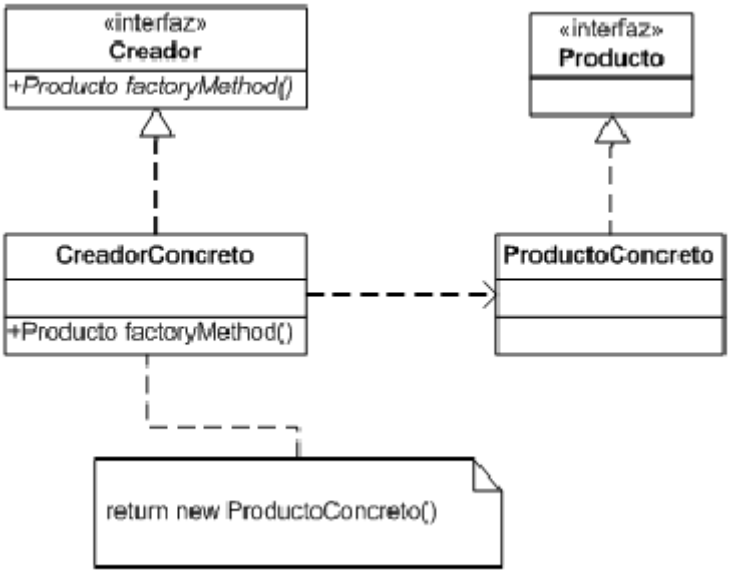
1. Desee crear un framework extensible. Esto significa proporcionar flexibilidad dejando algunas decisiones, como el tipo especifico del objeto a crear, para un momento posterior.
2. Desee que sea una subclase, en vez de la superclase, la que decida que tipo de objeto hay que crear.
3. Sabe cuando hay que crear un objeto, pero no conoce el tipo de objeto.
4. Necesita algunos constructores sobrecargados con la misma lista de parámetros, lo cual no esta permitido en Java. En ves de eso, utilice varios métodos de fabricación con distinto nombre.

Descripción:

Este patrón se llama Factory Method porque crea (fabrica) objetos cuando lo necesita Cuando empieza a escribir una aplicación, a menudo esta claro que tipo de componentes se utilizaran. Normalmente, se tiene una idea general de las operaciones que deben tener ciertos componentes, pero la implementación se realiza en otro momento, por lo que pueden surgir cosas que no tuvimos en cuenta.

Esta flexibilidad puede ser alcanzada utilizando interfaces para estos componentes. Pero, el problema de programar interfaces es que no se puede crear un objeto a partir de una interfaz. Se necesita una clase que las implemente para obtener un objeto. En vez de escribir una clase en su aplicación, puede extraer la funcionalidad del constructor e introducirla en un método. Ese método es el método de fabricación. Esto produce un objeto ConcreteCreator cuya responsabilidad es crear los objetos adecuados. Ese objeto ConcreteCreator crea instancias de una implementación (ConcreteProduct) de una interfaz
(Procudct).

Implementacion:



Para implementar el padrón FactoryMethod se necesitan:

1. Producto: la interfaz de los objetos creados por la fábrica.
2. ProductoConcreto: la clase que implementa Producto. Los objetos de esta clase son creados por CreadorConcreto.
3. Creador: la interfaz en la que se definen los métodos de fabricación (factoryMethod).

4. CreadorConcreto: la clase que hereda de Creador, proporciona una implementación para factoryMethod. Puede devolver cualquier objeto que implemente la interfaz Producto.

Ventajas e Inconvenientes:
El objeto contenedor puede ser muy genérico. Solo necesita saber como solicitar una funcionalidad de un elemento. La información de cómo funciona el elemento compuesto se mantiene en cada uno de ellos. Todo esto hace que el contenedor sea mas modular, facilitando la inclusión de nuevos elementos que pueden ser gestionados sin modificar el corazón del propio programa.
JDBC utiliza el patron Factory Method en muchas de sus interfaces. Una vez que se ha cargado un controlador correcto, se puede intercambiar por otro controlador JDBC. El resto de la aplicación no sufre ninguna modificación.
El inconveniente de este patrón es el hecho de que, para añadir un nuevo tipo de producto, debe introducir una nueva clase de implementación y cambiar al CreadorConcreto que ya existe, o crear una nueva clase que implemente la interfaz Producto.

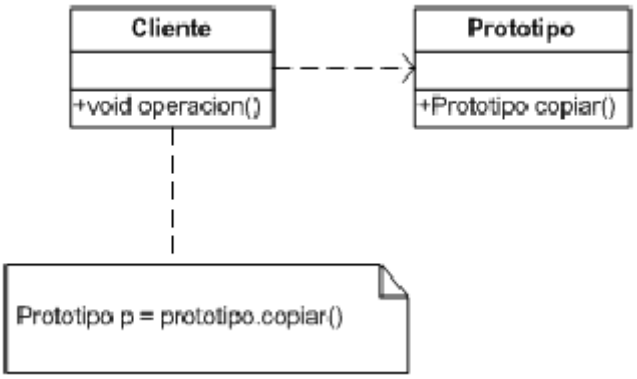
Variaciones del Patrón:
Hay distintas variaciones del patron:
1. Creador puede proporcionar una implementacion estándar para los métodos de fabricación. Por ello, Creador no tiene que ser una clase abstracta o una interfaz, sino una clase completamente definida. La ventaja es que no se necesita crear una subclase de Creador.
2. Producto puede ser implementada como una clase abstracta. Debido a que Producto ya es una clase, puede incluir implementaciones para los otros métodos.
3. El método de fabricación puede tomar algún parámetro. Por tanto, puede crear más de un tipo de Producto basándose en el parámetro dado. Esto reduce el número de métodos de fabricación necesarios.

Patrones relacionados:
Entre los patrones relacionados se incluye los siguientes:
1. Abstract Factory: podrá utilizar uno o más métodos de fabricación.
2. Prototype: evita crear subclases de Creador.
3. Template Method (Método Plantilla): los métodos plantilla normalmente llaman a los métodos de fabricación.

FORMATO DE DESCRIPCION DE PATRON DE DISEÑO	
Nombre del Patrón	Prototype
También conocido como: Plantilla de generación	
Propiedades del Patrón: Tipo: de creación, objeto. Nivel: Clase única	
Propósito: Facilita la creación dinámica al definir clases cuyos objetos pueden crear copias de si mismos.	
Introducción: En algunos casos es necesario obtener una copia de la información de un objeto para utilizarla en otro contexto. Una solución es hacer visible los atributos del objeto, pero ello acarrea el violar el principio de encapsulamiento de la POO. Una solución mas interesante es dotar a la clase que requiere este comportamiento de los métodos de copiado o clonación que permitan el obtener copias y utilizarlas en forma segura dentro del programa.	
Aplicabilidad Utilice el patrón Prototype cuando necesite crear un objeto que sea una copia de un objeto existente.	

Descripción:
El patrón Prototype tiene un nombre adecuado; como sucede con otros prototipos, hay un objeto utilizado como base para crear una nueva instancia con los mismos valores. Como proporciona un comportamiento de creación basada en un estado existente, es posible que los programas lleven a cabo operaciones como la copia dirigida por el usuario, al tiempo que permite inicializar los objetos a un estado que ha sido establecido durante el uso del sistema. Esto suele ser preferible a inicializar el objeto con algún conjunto de valores genéricos.
En los editores gráficos y de texto encontramos los ejemplos clásicos para este patrón, donde las características copiar-pegar pueden mejorar la productividad de los usuarios. Algunos sistemas empresariales también utilizan esta aproximación, produciendo un modelo a partir de un objeto de negocio existente. Después, la copia puede ser modificada para que adopte el nuevo comportamiento deseado.

Implementación:



Para implementar el patrón Prototype se necesita:
1. Prototipo: proporciona un método de copia. Ese método devuelve una instancia de la misma clase con los mismos valores que la instancia Prototipo original. La nueva instancia puede ser una copia profunda o superficial del original.

Ventajas e Inconvenientes:
El patrón prototype es bastante útil porque permite que los sistemas generen una copia de un objeto, con variables ya establecidas a un valor (presumiblemente) significativo, en vez de depender de un estado básico definido por el constructor.
Una consideración clave para este patrón es la profundidad de la copia.
1. Una copia solo duplica los elementos de alto nivel de una clase; esto proporciona una copia más rápida, pero que no siempre resulta apropiada. Como las referencias se copian desde el original a la copia, aun se refieren a los mismos objetos. Los objetos de bajo nivel son compartidos entre las copias del objeto, por lo que los cambios en uno de los objetos afectaran a todas las copias.
2. Las operaciones de copia profunda no solo duplican los atributos de alto nivel, sino también los objetos de bajo nivel. Esto suele llevar más tiempo, y puede ser muy costoso para objetos de una estructura arbitrariamente compleja. Esto asegura que los cambios en una copia se aíslan en las otras copias.

Variaciones del Patrón:
Incluyen:
1. Constructor de copia: una variación del prototipo es un constructor de copia. Este tipo de constructores toma una instancia de la misma clase como parámetro y devuelve una nueva copia con los mismos valores del parámetro.
Un ejemplo es la clase String, en la que se puede crear una nueva cadena utilizando una llamada como new String(“texto”);

2. Producto puede ser implementada como una clase abstracta. Debido a que Producto ya es una clase, puede incluir implementaciones para los otros métodos.
3. El método de fabricación puede tomar algún parámetro. Por tanto, puede crear más de un tipo de Producto basándose en el parámetro dado. Esto reduce el número de métodos de fabricación necesarios.

Patrones relacionados:
Entre los patrones relacionados se incluye los siguientes:
1. Abstract Factory: las Abstract Factory pueden utilizar el patrón Prototype para crear nuevos objetos basándose en el uso actual de la fábrica.
2. Factory Method: Los métodos de fabricación pueden utilizar un Prototype para actuar como plantilla para los nuevos objetos.

FORMATO DE DESCRIPCION DE PATRON DE DISEÑO	
Nombre del Patrón	Singleton
También conocido como: Única Instancia	
Propiedades del Patrón: Tipo: Creación Nivel: objeto	
Propósito: Permite tener una única instancia de esta clase en el sistema, a la vez que se permite que todas las clases tengan acceso a esa única instancia.	

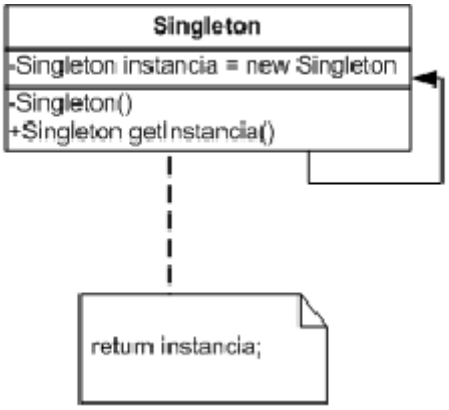
Introducción: En algún momento necesitara un objeto global; uno que sea accesible desde cualquier parte pero que solo deba ser creado una sola vez. Ud. necesitara que todas las partes de la aplicación sean capaces de utilizar el objeto, pero que todas deban utilizar la misma instancia.
Un ejemplo de esto es un historial, un ejemplo de acciones que ha llevado a cabo un usuario durante el uso de la aplicación. Muchas partes de la aplicación utilizan el mismo objeto ListaHistorial para introducir acciones que ha ejecutado el usuario, o bien para deshacer las acciones previas.
Una forma de lograrlo es hacer que la aplicación principal cree un objeto global y, después, pasar una referencia a el a todos los objetos que lo necesiten. Sin embargo, puede ser difícil el determinar como pasar la referencia y averiguar que partes de la aplicación necesitan crear el objeto. Otro inconveniente de esta solución es que impide que otros objetos puedan crear una instancia del objeto global, en el caso de la ListaHistorial.
Otra forma de crear valores globales es utilizando variables estáticas. En este caso, la aplicaron puede tener varios objetos estáticos dentro de una clase y acceder a ellos directamente.
Esta aproximación tiene los siguientes inconvenientes:
1. Un objeto estático no basta, ya que se crea al mismo tiempo que se crea la clase y no ofrece la oportunidad de proporcionar ningún dato antes de ser instanciado.
2. No se tiene control sobre quien accede al objeto. Cualquiera puede acceder a una instancia estática, públicamente disponible.
3. Si es necesaria la existencia del objeto Singleton, es necesario modificar todo el código del Cliente.

Aplicabilidad
Se puede utilizar el patrón Singleton cuando solo quiere una instancia de una clase, pero esta instancia debe estar disponible globalmente.

Descripción:
El patron Singleton asegura que se crea un máximo de una instancia en la memoria (De

ahí proviene su nombre). Para asegurar que se tiene control sobre la Instanciación, se debe hacer su constructor privado.
Esto implica un problema: es imposible crear una instancia, por lo que se proporciona un método estático (getInstancia()) para acceder a una instancia de esa clase. Este método crea una instancia única, en caso de que aun no exista, y devuelve la referencia al llamador del método. La referencia al Singleton también se almacena como un atributo privado estático de la clase singleton para llamadas futuras.
Aunque el método e acceso puede crear el Singleton, la mayoría de las veces se crea en el momento que se carga la clase. Solo es necesario posponer la construcción si tiene que hacer algún tipo de inicialización antes de instanciar el Singleton.

Implementación:



Para implementar el patron Singleton se necesita:
1. Singleton: proporciona un constructor privado, mantiene una referencia estática a la única instancia de la clase, y proporciona un método estático de acceso para devolver una referencia a la instancia única.
El resto de la implementacion de la clase Singleton es completamente normal. El método estático de acceso pude tomar decisiones sobre que tipo de instancia crear, basándose en la propiedades del sistema o los parámetros pasados al método de acceso.

Ventajas e Inconvenientes:
1. La clase Singleton es la única clase que puede crear una instancia de si misma. No se pude obtener ninguna instancia sin utilizar el método estático proporcionado.
2. No necesita pasar la referencia a todos los objetos que acceden al singleton.
3. Sin embargo, el patron Singleton puede presentar problemas de acceso multihilo, dependiendo de como se haya realizado su implementacion. Se debe tener cuidado dado que puede ocasionar deadLocks entre hilos en una aplicación.

Variaciones del Patrón:
1. Una de las opciones mas exploradas del patron Singleton, es tener más de una instancia dentro de la clase. La ventaja es que el resto de la aplicación puede permanecer inalterada, mientras que todos aquellos objetos que conocen la existencia de múltiples instancias pueden utilizar algún método para obtener una de las otras instancias.
2. El método de acceso al Singleton puede ser el punto de acceso al conjunto total de instancias, aunque todas tengan un tipo distinto. El método de acceso puede determinar, en tiempo de ejecución, que tipo de instancia devolver. Esto podría parecer extraño, pero es muy útil cuando se utiliza la carga dinámica de clases. El sistema que utiliza un Singleton puede permanecer igual, mientras que la implementación puede ser diferente.

Patrones relacionados:

Entre los patrones relacionados están:
1. AbstracFactory
2. Builder
3. Prototype

Escenario
Corroborar lo visto, en un IDE de desarrollo con ejemplos listos para ejecutar en Eclipse.

Tarea 1: Importe los patrones en un IDE de desarrollo

1. Utilizando eclipse, cree un espacio de trabajo nuevo (WorkSpace).
2. Cree un nuevo proyecto denominado Patrones de Diseño y en el cree la estructura de paquetes dados a continuación.
    a. edu.logica.[***nombrepatron***]
3. Importe el código desde el material del laboratorio y cree los diagramas correspondientes a cada patrón, buscando su coincidencia con los gráficos presentados en la forma canónica del patrón.
4. Ejecute en cada caso la clase EjecutarPatron para verificar su funcionamiento.
5. Agregue funcionalidad relevante a cada implementación.

Fin de laboratorio.