

Tarea #1

Autores:

Josselyn Mora Santamaria

[josselyn.mora.santamaria@est.una.ac.cr](mailto:josselyn.mora.santamaria@est.una.ac.cr)

Joshua Granados Loría

[joshua.granados.loria@est.una.ac.cr](mailto:joshua.granados.loria@est.una.ac.cr)

Kevin Mora Valverde

[kevin.mora.valverde@est.una.ac.cr](mailto:kevin.mora.valverde@est.una.ac.cr)

Universidad Nacional de Costa Rica.

Curso: Programación III

II Ciclo-NCR 51172-Grupo 89

Profesor: Máster Rubén Mora Vargas

04 de octubre del 2021

Contenido

[Introducción 3](#_Toc84180260)

[Contenido 4](#_Toc84180261)

[Abtract Factory 4](#_Toc84180262)

[Flyweight 6](#_Toc84180263)

[Memento 8](#_Toc84180264)

[Conclusión 11](#_Toc84180265)

[Recomendaciones 12](#_Toc84180266)

[Bibliografías 13](#_Toc84180267)

[Anexos 14](#_Toc84180268)

# Introducción

# Contenido

## Abtract Factory

El abstract factory o fábrica abstracta en español, es parte de los patrones de diseño propuestos por el Gang of Four (Gamma, Helm, Johnson y Vlissides) los cuales se utilizan como una guía que los programadores utilizan acorde a las funciones o exigencias de la aplicación que deseen desarrollar (Blanco & Universidad de Cantabria, s.f.).

Dicho patrón de diseño crea una sola interfaz para crear una familia completa de productos, el usuario que la utiliza solo ve la interfaz, no ocupa conocer las clases que la componen. Además, es conocida con el nombre de kit.

Abstract factory es un patrón de diseño con calificación creacional con un ámbito de objetos, es decir actúa como una fábrica que crea objetos relacionados entre sí o que dependan entre ellos. También realiza una encapsulación de reglas de instanciación, esta permite que en la clase abstract factory se puedan ocultar el proceso para la instanciación de un grupo de objetos (Polanco, F. G., s.f.).

La motivación al utilizar dicho patrón es poder crear aplicaciones independientes de la interfaz con la cual el usuario interactúa.

Además, es ideal para cuando una aplicación requiere manejar muchas familias de productos.

Según la Universidad Carlos III de Madrid (2007) para llevar a cabo la creación de una fábrica abstracta se debe establecer lo siguiente:

**AbstractFactory**: Es la interfaz encargada de crear los objetos abstractos.

**ConcreteFactory:** Implementa las operaciones para crear objetos concretos.

**AbstractProduct:** Define la interfaz de un objeto producto.

**ConcreteProduct:** Implementa la interfaz AbstractProduct.

**Client:** Utiliza las interfaces de AbstractFactory y AbstractProduct.

La siguiente información fue obtenida por medio del estudio de Blanco y Universidad de Cantabria (s.f.):

**Ventajas:**

* Facilita el cambio de familia de productos. Ya que la encargada de la familia de productos es la clase concrete y esta solo aparece una vez en el código.
* Permite que la aplicación pueda usar varios productos a la vez siempre y cuando sean de la misma familia.

**Desventajas:**

* Es complicado agregar nuevos productos, ya que afecta a la interfaz y por ende a las subclases.

Estructura del Abstract Factory:

En la figura 1 se puede observar un diagrama de cómo está compuesto el patrón de diseño AbstractFactory:

El cliente interactúa con las interfaces las cuales ayudaran al mismo en la utilización de la aplicación. La clase cliente accede a los métodos de ConcreteFactory por medio de la clase Abstract Factory. Y para utilizar los productos creados utiliza la interfaz AbstractProduct.

ConcreteFactory se encarga de implementar la interfaz AbstractFactory, posee las operaciones de crear y retornar objetos.

Y quien define los objetos creados es la ConcreteFactory.

Figura 1

Estructura del Abstract Factory

Diagrama

Descripción generada automáticamente

*Nota:* Estructura del patrón de diseño AbstractFactory. Pertenece al siguiente documento: *Abstract Factory* (Polanco, F. G., s.f.).

Diferencia entre Factory Method y Abstract Factory:

La diferencia entre estos dos métodos se basa en su objetivo, el abstract factory tiene como objetivo crear un conjunto de objetos relacionados entre sí, mientras que el Factory Method aplazar hacia una determinada clase o subclase el tipo de objeto que se va a instanciar.

## Flyweight

Este diseño de patrón sirve para reducir el uso de memoria, esto mediante la compartición de partes en común entre varios objetos, así se reduce una gran cantidad de código al no tener que mantener la misma información en todos los objetos, este proceso va mucho de la mano con lo que es la herencia ya que cumplen una función muy similar. Según Shvets (2019) Flyweight tiene la siguiente estructura:

1. Se tiene que entender que Flyweight es una manera de optimizar el código de la aplicación que se esté creando, se suele implementar cuando el programa creado consume mucha RAM y no se logra disminuir ese consumo de otra manera.
2. Clase Flyweight: En esta clase se encuentra los estados del objeto original del cual los objetos hijos van a heredar, este estado que se almacena dentro del objeto Flyweight se le denomina como intrínseco y los que se les pasa a sus métodos se denomina extrínseco.
3. Clase Contexto: Dentro de esta clase se encuentra el estado extrínseco, este estado es único entre todos los objetos originales, en el momento en el que un contexto se logra unir con uno de los objetos Flyweight se logra un estado completo del objeto original.
4. Cuando se invoque un método de un objeto de tipo Flyweight se debe de pasar aquellas partes del estado que sean útiles mediante los parámetros del método, además los métodos que le dan un comportamiento al objeto Flyweight se pueden utilizar en la clase de contexto.
5. Clase cliente: En esta se va a realizar todo calculo o almacenamiento del estado extrínseco de los distintos objetos Flyweight. Desde una vista del propio cliente, un Flyweight es un tipo de objeto de tipo plantilla el cual puede ser configurado durante la ejecución del programa, esto pasa porque envía información contextual dentro de los parámetros de sus distintivos métodos.
6. Clase Fabrica Flyweight: Mediante esta clase se gestionan los grupos existentes de objetos Flyweight, por lo tanto, la clase cliente no se hace cargo de crear directamente los objetos Flyweight, si no que llama a la fábrica y se le envían los parámetros que se necesite del estado intrínseco. La fábrica hace una revisión de los objetos Flyweight que se crearon previamente y devuelve uno existente que pueda coincidir con los parámetros enviados y si no llega a encontrar ninguno entonces crea uno nuevo.

En la siguiente imagen se puede observar un diagrama UML con la estructura mencionada anteriormente.

Figura 2

Diagrama UML de Flyweight.

Diagrama

Descripción generada automáticamente con confianza baja

*Nota:* Diagrama UML de la estructura del patrón de diseño Flyweight. Tomado de Refactoring Guru(Shvets, 2019).

Según el estudio de Shvets (2019) las ventajas y desventajas de utilizar este diseño es:

Ventajas:

* El ahorro del uso de la memoria RAM.

Desventajas:

* El código puede convertirse un poco complicado de entender.
* Si hay cambios de RAM por ciclos de CPU el cálculo de una nueva parte de la información se realizará cada vez que se invoque un método Flyweight.

Según Shvets (2019) para implementar este patrón de diseño en un código se deben de seguir los siguientes pasos:

1. Se tienen que dividir los campos de una clase para que se convierta en la clase Flyweight, esta clase se divide en dos partes:

* Estado intrínseco: Son aquellos campos que van a tener la información única de un objeto, esta va a estar duplicada a través de todos los objetos del mismo tipo.
* Estado extrínseco: Son los campos que almacenan toda información contextual que es única en cada objeto.

1. Los campos que incorporan el estado intrínseco se tienen que dejar en la clase sin embargo estos no deben de ser alterados, se tiene que asegurar que los valores iniciales se encuentren dentro del constructor.
2. Se tienen que revisar los métodos que utilizan el estado extrínseco para agregarles un nuevo parámetro y suplantar el campo.
3. Se crea la clase fabrica para encargarse de un grupo de objetos Flyweight, una vez se encuentre esta clase lista, los clientes podrán solicitar cualquier objeto Flyweight.
4. Dentro de la clase cliente se debe guardar o calcular los valores del estado extrínseco para así lograr la invocación de los métodos de los objetos Flyweight, si se desea se puede mover el estado extrínseco a la clase contexto.

## Memento

El patrón de diseño memento es el encargado de hacer posible la acción de restaurar objetos o elementos a su estado original o al estado deseado, este patrón trabaja mediante la toma de “imágenes” instantáneas a los objetos en tiempo real, con el fin de poder devolverlos a ese momento sin quebrantar la encapsulación que posee dicho objeto. En palabras técnicas, “el patrón memento es un objeto que aloja una copia del estado interno de otro elemento” (Gamma, Helm, Johnson & Vlissides, 2003, p.141).

Este patrón trabaja bajo la siguiente estructura:

Figura 3

Patrón Memento

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

*Nota:* Estructura del patrón de diseño memento. Pertenece al siguiente documento:Patrones de Diseño Elementos de software orientado a objetos reutilizable*.* (Gamma Et al, 2003, p.141).

En esta estructura, tal y como lo comentan Gamma y compañía (2003) se puede apreciar que cuenta con una clase principal llamada “Originator”, esta cuenta con una variable de estado, que almacena la condición en la cual se mantiene el objeto a la hora de tomar la instantánea.

También cuenta con dos métodos: el primero se encarga de utilizar el memento para volver al estado deseado y el segundo es el que crea el memento que guarda la instantánea del interior del objeto; seguido de esto, está la clase “Memento”, ésta cuenta con los get y los sets del atributo estado, esta clase se encarga de guardar los estados del objeto Originator, también vela por la protección del objeto ante los intentos de acceso de objetos externos que no sea el padre.

Estas clases (Memento), cuentan con la interacción de dos interfaces, de las cuales una de ellas tiene más acceso que la otra, el Caretaker cuenta con una interacción menor ya que este solo tiene la posibilidad de pasar el memento a otros, en cambio el Originator cuenta con una interacción mayor, debido a que él puede acceder a los datos para devolver un objeto a su estado original, por consiguiente, únicamente él tiene la posibilidad de interactuar con el estado interno de éste.

Por último, está la clase Caretaker, ésta se encarga de almacenar de forma segura el memento, también, está clase no inspecciona los objetos y tampoco puede realizar alguna acción sobre ellos.

**Ventajas y desventajas al utilizar el patrón de diseño memento**.

Como principal ventaja de la implementación de este patrón de diseño es la recuperación adecuada de un objeto en caso de daños o errores, esto brinda la seguridad de los datos en caso de daño, además ofrece confiabilidad en la aplicación que utilice este patrón debido a la fácil recuperación de datos que ofrece.

En el caso de las desventajas, autores como Gamma, Helm, Johnson y Vlissides para el año 2003, proponen las siguientes:

1. Conservación de los límites de la encapsulación:

Este patrón impide la exposición de los datos a los cuales solo tiene acceso el Originator, pero irónicamente, estos datos son almacenados fuera del mismo por lo que el memento encubre la información a los demás objetos con el objetivo de mantener los límites del encapsulamiento.

1. Costoso:

Los mementos pueden producir grandes inversiones para una empresa por los altos costos de almacenamiento, ya que al estar copiando gran cantidad de datos para luego guardarlos en el memento o bien si se está constantemente creando, restableciendo y eliminando mementos puede generan un gasto considerable para las organizaciones.

1. Dificultad de implementación:

En algunos lenguajes de programación puede ser muy complicado gestionar que sea solo el Originator el encargado de acceder a los estados de los objetos.

**Implementación.**

Para una implementación adecuada de este patrón de diseño, según lo que recomienda Gamma, Helm, Johnson y Vlissides (2003), se debe tener en cuenta dos aspectos muy importantes: el primero es el soporte del lenguaje y el segundo es el guardar únicamente los cambios incrementales.

* Soporte del lenguaje: la implementación de este patrón conlleva la creación de dos interfases (la de creadores y la de los otros objetos), por esta razón es muy importante que sea un lenguaje que soporte dos niveles de implantación estática, ejemplo de este lenguaje es C++ que permite que la clase Originator sea amiga de Memento y permite que se privatice la interfaz del Memento, como ejemplo:

Figura 4

Implementación del patrón de diseño memento en el lenguaje C++.

Interfaz de usuario gráfica, Texto, Aplicación, Word

Descripción generada automáticamente

*Nota:* Ejemplo de implementación del patrón de diseño memento en el lenguaje C++. Pertenece al siguiente documento:Patrones de Diseño Elementos de software orientado a objetos reutilizable*.* (Gamma Et al, 2003, p.142).

* Guardar únicamente los cambios incrementales: en el momento en que se crea un memento y luego es devuelto a su estado, éste puede guardar solo los cambios con respecto al estado del creador, el uso de un antecedente puede representar un orden de las acciones hacia un memento, por lo que estos tienen la posibilidad de almacenar únicamente los cambios realizados y no todo el estado.

# Conclusión

# Recomendaciones

# Bibliografías

Shvets, A. (2019). *Flyweight*. Refactoring.Guru. <https://refactoring.guru/es/design-patterns/flyweight>

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2003). *Patrones de Diseño Elementos de software orientado a objetos reutilizable*. Editorial PEARSON EDUCACION.S.A. Recuperado de <https://profeuttec.yolasite.com/resources/Patrones%20de%20dise%C3%B1o%20-%20Erich%20Gamma.pdf>

# Anexos

En el siguiente enlace de GitHub se puede encontrar un ejemplo programado dedicado a cada patrón de diseño investigado.

<https://github.com/joshua292000/Tarea-1.git>