Universidad Autónoma Metropolitana

División Ciencias Naturales e Ingeniería

Patrones de Diseño en la Ingeniería de Software

TRABAJO ESCRITO DE INVESTIGACIÓN
CORRESPONDIENTE A LOS PROYECTOS TERMINALES
I, II Y III" "TRIMESTRES 19-P, 19-O Y 20-I

P R E S E N T A :

Héctor Ponce Rodriguez

TUTOR

Dr. Pedro Pablo Gonzalez Peréz

Cuajimalpa, Cd. Mx., 2020

Dedicatoria ...

Agradecimientos

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Descripción del proyecto

En este bloque se introducen todos los aspectos generales del proyecto desarrollado, además del contenido general de esta memoria. “Framework por definir...” es el proyecto final de carrera realizado por Héctor Ponce Rodriguez, estudiante de ingeniería en computación, de la Universidad Autónoma Metropolitana Unidad Cuajimalpa.

1.1.1. Motivación

Durante el transcurso de la carrera, las asignaturas que me despertaban un mayor interés eran aquellas relacionadas con la ingeniería de software. Este interés personal provocaba una mayor motivación al realizarlas y por tanto unos mejores resultados en estas asignaturas.

Por otro lado, en el plan de estudios encontré una gran diversidad de asignaturas de proyecto dedicadas a la ingeniería del software, lo que me llevo a buscar mi Proyecto Terminal en esta rama.

Cuando el Dr. Pedro Pablo Gozález Perez me presento la oferta del presente proyecto me pareció una gran oportunidad por varias razones. Podría aplicar mis conocimientos de UML y Patrones de Diseño para desarrollar el proyecto, además de practicar varios lenguajes que no se contemplan durante los estudios, como JavaScript, Php

y HTML5. Además, en algunas asignaturas era difícil encontrar una aplicación para realizar diagramas UML que se adaptara a las necesidades que se requerían, o en su defecto, poder traducir un diagrama UML a difetentes lenguajes de programación.

1.1.2. Objetivos

El objetivo de este proyecto es crear una herramienta JavaScript, que permita realizar la visualización de diagramas UML sobre Patrones de Diseño y traducción de los mismo a diferentes lenguajes de programación, utilizando para ello las nuevas características incorporadas en el lenguaje HTML5 y JavaScript.
El usuario podrá interactuar dinámicamente con el diagrama UML sobre el patrón que guste, para poder modificarlo, adaptarlo, guardarlo y posteriormente traducirlo a diferentes lenguajes de programación como: Java, Python o Php. Se permitirá la generación de los diagramas tanto gráficamente, añadiendo los elementos desde una interfaz, como textualmente a través de un lenguaje específico.

1.1.3. Justificación

Todo desarrollo de cualquier proyecto de ingeniería, desde la construcción de un procesador de textos a un software de comunicaciones para Internet, requiere de etapas de modelado que permitan experimentar y visualizar el sistema que se construirá. Dado que son unas etapas tan importantes a la hora de construir un sistema, una herramienta que facilite este proceso beneficiará tanto al resultado final, como al tiempo invertido en la generación del mismo.
Uno de los problemas que nos podemos encontrar durante este proceso, es no disponer de una herramienta que se ajuste a nuestras necesidades, y vernos obligados a instalar una herramienta específica que no cubra todos nuestros requisitos. Existen muchas herramientas CASE (Computer-aided software engineering) (Leblang and Chase Jr 1984)

1.1.4. Acerca del documento

Dado que nuestra herramienta estará implementada en HTML5, los usuarios sólo necesitaran un navegador web para utilizarla, sin la necesidad de instalar software adicional. HTML5 está ganando importancia de manera muy rápida, revolucionando el diseño web. Por tanto, nuestra herramienta se beneficiará de sus ventajas.

Capítulo 2

Introducción a patrones de diseño

2.1. Historia de patrones de diseño

Los patrones de diseño no son conceptos oscuros y sofisticados, sino todo lo contrario. Los patrones son soluciones típicas a problemas comunes en el diseño orientado a objetos. Cuando una solución se repite una y otra vez en varios proyectos, alguien finalmente le pone un nombre y describe la solución en detalle. Así es básicamente cómo se descubre un patrón (Yacoub and Ammar 2004)

El concepto de patrones fue descrito por primera vez por Christopher Alexander en A Pattern Language: Towns, Buildings, Construction . El libro describe un "lenguaje"para diseñar el entorno urbano. Las unidades de este lenguaje son patrones. Pueden describir qué tan altas deben ser las ventanas, cuántos niveles debe tener un edificio, qué tan grandes se supone que deben ser las áreas verdes en un vecindario, etc.

La idea fue recogida por cuatro autores: Erich Gamma, John Vlissides, Ralph Johnson y Richard Helm. En 1994, publicaron Patrones de diseño: elementos de software orientado a objetos reutilizables , en el que aplicaron el concepto de patrones de diseño a la programación. El libro presenta 23 patrones que resuelven varios problemas de diseño orientado a objetos y se convirtió en un best-seller muy rápidamente. Debido a su largo nombre, la gente comenzó a llamarlo 'el libro de la pandilla de los cuatro', que pronto se redujo a simplemente 'el libro GoF'.

Desde entonces, se han descubierto docenas de otros patrones orientados a objetos. El

enfoque de patrones se hizo muy popular en otros campos de programación, por lo que ahora también existen muchos otros patrones fuera del diseño orientado a objetos.(1)

2.2. Qué es un patron de diseño

Esta fue la primer pregunta que me hice cuando comencé a investigar sobre este tema. Al principio no tenía mucha idea de por dónde comenzar, por lo que mi primera reacción fue realizar una búsqueda en Internet y obtener de esta manera alguna base sobre la cual apoyarme. La definición que más me gustó fue la siguiente:
"Los patrones de diseño son el esqueleto de las soluciones a problemas comunes en el desarrollo de software." (Gamma, Helm, and Johnson 1995)
Son como planos prefabricados que puede personalizar para resolver un problema de diseño recurrente en su código.
No se puede simplemente encontrar un patrón y copiarlo en nuestro programa, como poder hacerlo con las funciones o bibliotecas que tenemos disponibles en el mercado. El patrón no es un código específico, sino un concepto general para resolver un problema en particular. Puede seguir los detalles del patrón e implementar una solución que se adapte a la semántica de nuestro propio programa.(Peña 2017)
Los patrones a menudo se confunden con algoritmos, porque ambos conceptos describen soluciones típicas a algunos problemas conocidos. Si bien un algoritmo siempre define un conjunto claro de acciones que pueden lograr algún objetivo, un patrón es una descripción más de alto nivel de una solución. El código del mismo patrón aplicado a dos programas diferentes puede ser diferente.(Alpízar, Rodriguez, and Bolaños)
Una analogía con un algoritmo es una receta de cocina: ambos tienen pasos claros para lograr un objetivo. Por otro lado, un patrón es más como un plano: puede ver cuál es el resultado y sus características, pero el orden exacto de implementación depende de nosotros.
En otras palabras, brindan una solución ya probada y documentada a problemas de desarrollo de software que están sujetos a contextos similares. Debemos tener presente los siguientes elementos de un patrón: su nombre, el problema (cuando aplicar un

patrón), la solución (descripción abstracta del problema) y las consecuencias (costos y beneficios)(Tedeschi 2015).
Grande fue mi sorpresa al averiguar que existen varios patrones de diseño popularmente conocidos, los cuales se clasifican como se muestra a continuación:

- Patrones Creacionales: Inicialización y configuración de objetos.
- Patrones Estructurales: Separan la interfaz de la implementación. Se ocupan de cómo las clases y objetos se agrupan, para formar estructuras más grandes
- Patrones de Comportamiento: Más que describir objetos o clases, describen la comunicación entre ellos.

En la figura 2.1 podemos ver un ejemplo de error común al momento de estar implementando código:

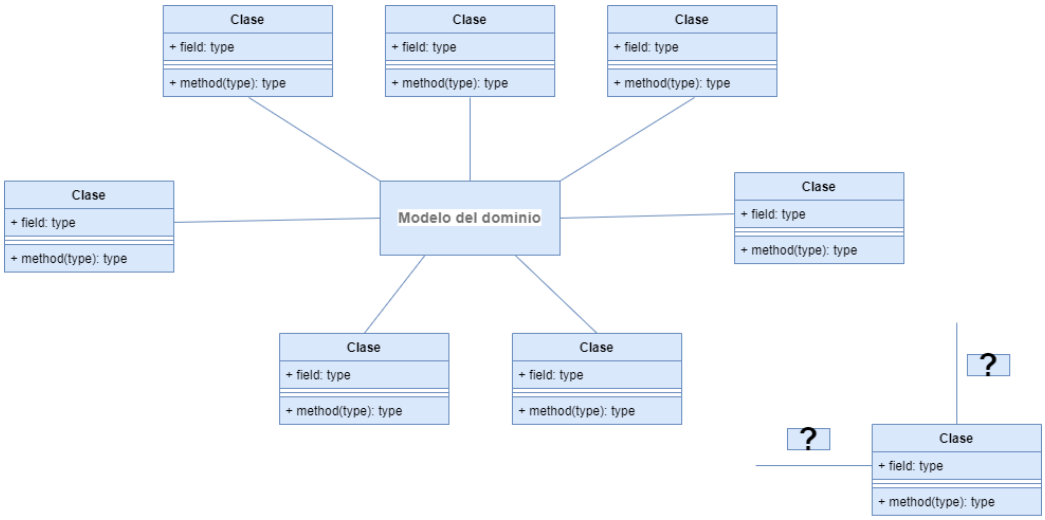

Figura 2.1: Agregar una nueva clase al programa no es tan simple si el resto del código ya está acoplado a las clases existentes.

En general, un patrón tiene cuatro elementos esenciales:

- El **nombre del patrón** es un identificador que podemos usar para describir un problema de diseño, sus soluciones y consecuencias en una o dos palabras. Nombrar un patrón aumenta inmediatamente nuestro vocabulario de diseño.

Nos permite diseñar a un nivel más alto de abstracción. Tener un vocabulario para patrones nos permite hablar sobre ellos con nuestros colegas, en nuestra documentación e incluso con nosotros mismos. Hace que sea más fácil pensar en diseños y comunicarlos y sus compensaciones a otros.(Gamma, Helm, and Johnson 1995)

- El **problema** describe cuándo aplicar el patrón. Explica el problema y su contexto. Podría describir problemas de diseño específicos, como cómo representar algoritmos como objetos. Podría describir estructuras de clase u objeto que son sintomáticas de un diseño inflexible. A veces, el problema incluirá una lista de condiciones que deben cumplirse antes de que tenga sentido aplicar el patrón.(Gamma, Helm, and Johnson 1995)

- La **solución** describe los elementos que componen el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño o implementación concreta en particular, porque un patrón es como una plantilla que se puede aplicar en muchas situaciones diferentes. En cambio, el patrón proporciona una descripción abstracta de un problema de diseño y cómo lo resuelve una disposición general de elementos (clases y objetos en nuestro caso).(Gamma, Helm, and Johnson 1995)

- Las **consecuencias** son los resultados y las compensaciones de aplicar el patrón. Aunque las consecuencias a menudo no se expresan cuando describimos las decisiones de diseño, son críticas para evaluar las alternativas de diseño y para comprender los costos y beneficios de aplicar el patrón. Las consecuencias para el software a menudo se refieren a compensaciones de espacio y tiempo. También pueden abordar problemas de lenguaje e implementación. Dado que la reutilización es a menudo un factor en el diseño orientado a objetos, las consecuencias de un patrón incluyen su impacto en la flexibilidad, extensibilidad o portabilidad de un sistema. Enumerar estas consecuencias explícitamente le ayuda a comprenderlas y evaluarlas.(Gamma, Helm, and Johnson 1995)

2.3. Elementos de un patrón de diseño

Un patrón de diseño generalmente se encuentra conformado por los siguientes puntos esenciales para obtener una clara descripción:

- **a) Nombre del patrón** Describe, en una o dos palabras, un problema de diseño junto con sus soluciones y consecuencias.
- **b) Función** Detalla cuándo aplicar el patrón, es una explicación del problema y su contexto.
- **c) Estructura** Describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones.

2.4. ¿Por qué debería aprender patrones de diseño?

Los patrones de diseño son un conjunto de herramientas de soluciones probadas para problemas comunes en el diseño de software. Incluso si nunca encuentra estos problemas, conocer los patrones sigue siendo útil porque le enseña cómo resolver todo tipo de problemas utilizando los principios del diseño orientado a objetos.(Gamma, Helm, Johnson, and Vlissides 2003)
Los patrones de diseño definen un lenguaje común que nosotros podemos usar para comunicarse de manera más eficiente. Podemos decir: 'Oh, solo use un Singleton para eso', y todos entenderán la idea detrás de nuestra sugerencia. No es necesario explicar qué es un singleton si conoce el patrón y su nombre.

En la figura 2.2 podemos ver una analogía del mundo real y como podemos entender la importancia de los patrones de diseño.
Un claro ejemplo de por qué debemos aprender a usar patrones de diseño es cuando empezamos a desarrollar software, es común que cada quien utilice su propia lógica, conocimientos y experiencia para crear código.

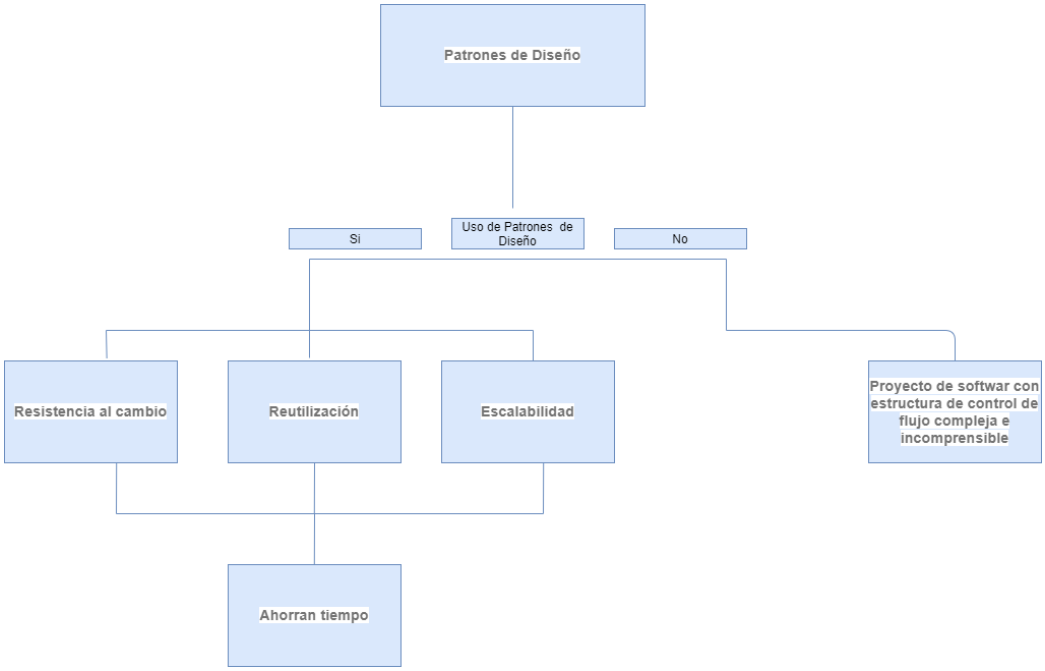

Figura 2.2: Diagrama de flujo para aplicar el uso de patrnes de diseño

Y esto muchas veces resulta en desarrollos complejos que sólo su creador entiende. Pero, ¿es posible desarrollar un módulo que otro programador pueda aprovechar entender y mejorar? La respuesta está en los patrones de diseño. Estos básicamente son modelos muestra que sirven como guía para que los programadores trabajen sobre ellos.

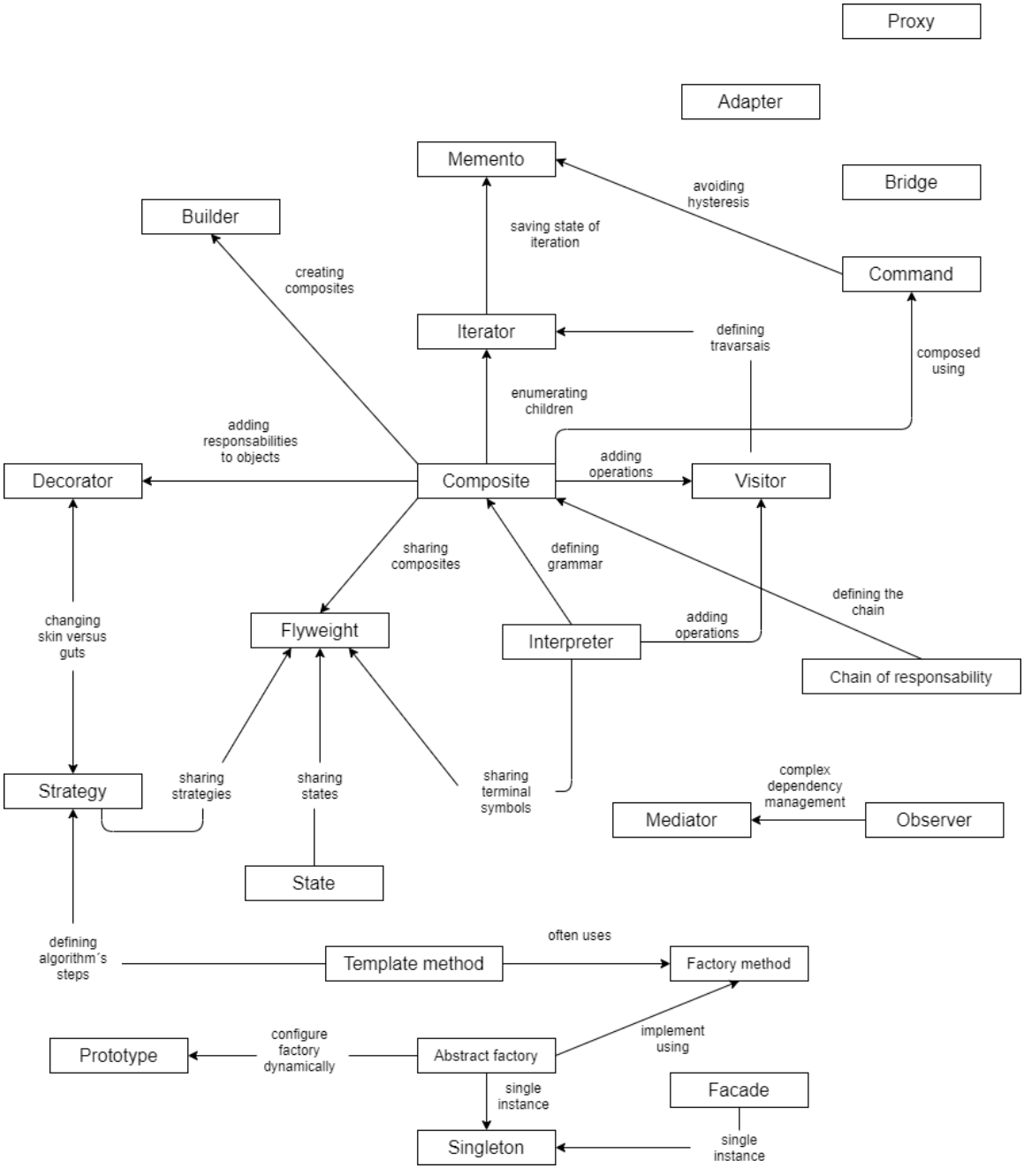

Figura 2.3: Relaciones entre patrones de diseño

Capítulo 3

Clasificacion Patrones de Diseño (GAMMA)

Como ya habiamos mencionado anteriormente, los patrones de diseño se clasifican en tres grandes grupos

3.1. Patrones de Creación

Los patrones de creación están asociados al proceso de creación de objetos, ya que entre objetos se delegan los procesos de creación que se llevaran acabo.

Abstraen el proceso de creación de instancias. Ayudan a hacer a un sistema independiente de cómo se crean, se componen y se representan sus objetos. Un patrón de creación de clases usa la herencia para cambiar la clase de la instancia a crear, mientras que un patrón de creación de objetos delega la creación de la instancia en otro objeto.

Los patrones de creación se hacen más importantes a medida que los sistemas evolucionan para depender más de la composición de objetos que de la herencia de clases. Cuando esto sucede, se pasa de codificar una serie de comportamientos fijos a definir un conjunto más pequeño de comportamientos fundamentales que pueden componerse con otros más complejos. Así, para crear objetos con un determinado comportamiento es necesario algo más que simplemente crear una instancia de una clase. (Gamma,

Helm, Johnson, and Vlissides 2003)

Hay dos temas recurrentes en estos patrones. En primer lugar, todos ellos encapsulan el conocimiento sobre las clases concretas que usa el sistema. Segundo, todos ocultan cómo se crean y se asocian las instancias de estas clases. Todo lo que el sistema como tal conoce acerca de los objetos son sus interfaces, tal y como las definen sus clases abstractas. Por tanto, los patrones de creación dan mucha flexibilidad a qué es lo que se crea, quién lo crea y cuándo. Permiten configurar un sistema con objetos "producto" que varían mucho en estructura y funcionalidad. La configuración puede ser estática (esto es, especificada en tiempo de compilación) o dinámica (en tiempo de ejecución).

Entre los principales patrones que se encuentran en este grupo podemos listar los siguientes:

3.1.1. Abstract factory

- **Función:** Trabajar con objetos de distintas familias de manera que no se mezclen entre sí, haciendo transparente el tipo de familia concreta que se esté usando.

- **Estructura:**

  - Class(cliente): Representa la persona o evento que dispara la ejecución del patrón.
  - AbstractProduct (One, Two): Interfaces que definen la estructura de los objetos para crear familias.
  - Product (One, Two): Clases que heredan de AbstractProduct con el fin de implementar familias de objetos concretos.
  - Platform (One, Two): Representan las fábricas concretas que servirán para crear las instancias de todas las clases de la familia. En esta clase debe existir un método para crear cada una de las clases de la familia.
  - AbstractPlatform: Define la estructura de las fábricas y deben proporcionar un método para cada clase de la familia.

La estructura que cumple este patrón se muestra en la Figura 3.1.

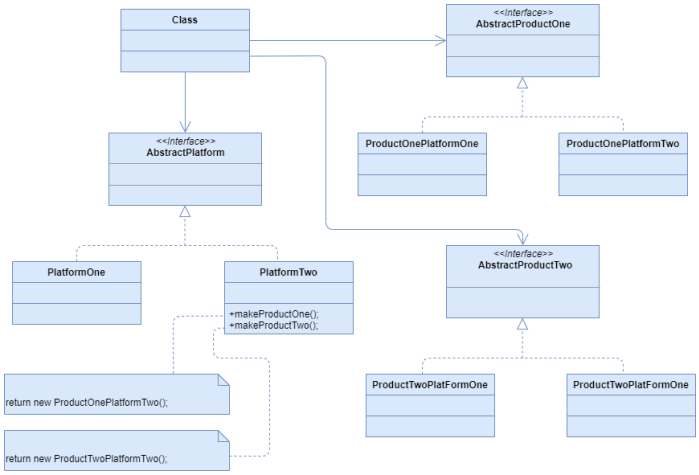

Figura 3.1: Diagrama UML Patrón Abstract Factory

3.1.2. Factory Method

- **Función:** Centralizar en una clase constructora la creación de objetos de un subtipo de un tipo determinado.

- **Estructura:**

  - Framework: Representa de forma abstracta el objeto que queremos crear, mediante esta interface se definen la estructura que tendrá el objeto creado.

  - Aplicacion (uno, dos): Representa una implementación concreta de la interface Framework, la cual es creada a través del Producto(uno,dos).

  - Producto: Este componente puede ser opcional, sin embargo, se recomienda la creación de un producto(AbstractFactory) que define el comportamiento por default de los Producto(uno,dos).

  - Producto(uno,dos): Representa una fábrica concreta la cual es utilizada para la creación de los ConcreteProduct, esta clase hereda el comportamiento básico del producto(AbstractFactory).
    La estructura que cumple este patrón se muestra en la Figura 3.2.

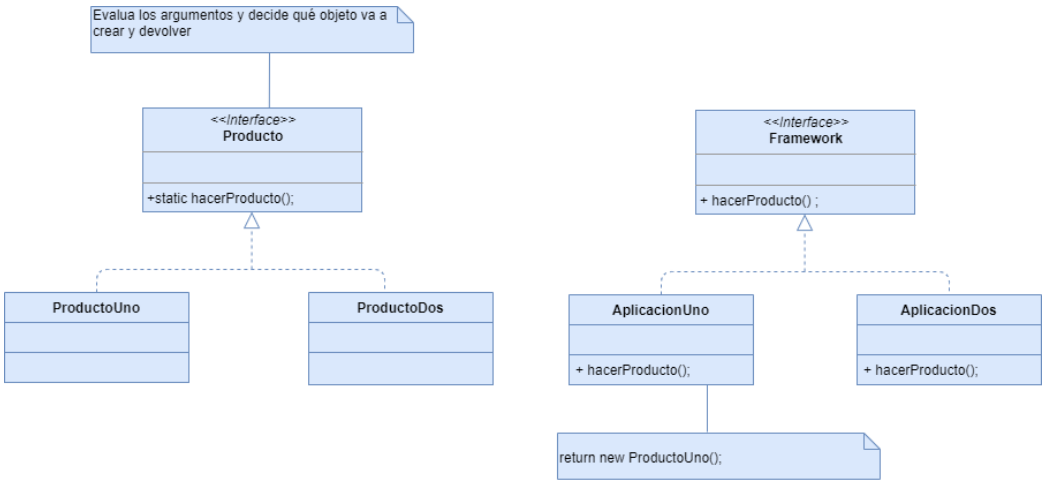

Figura 3.2: Diagrama UML Patrón Factory Method

3.1.3. Builder

- **Función:** Facilitar la abstracción del proceso de creación de un objeto complejo, centralizándolo en un único punto.

- **Estructura:** Como se aprecia en la figura 3.3

  El lector encapsula el análisis de la entrada común. La jerarquía de constructores hace posible la creación polimórfica de muchas representaciones u objetivos peculiares.

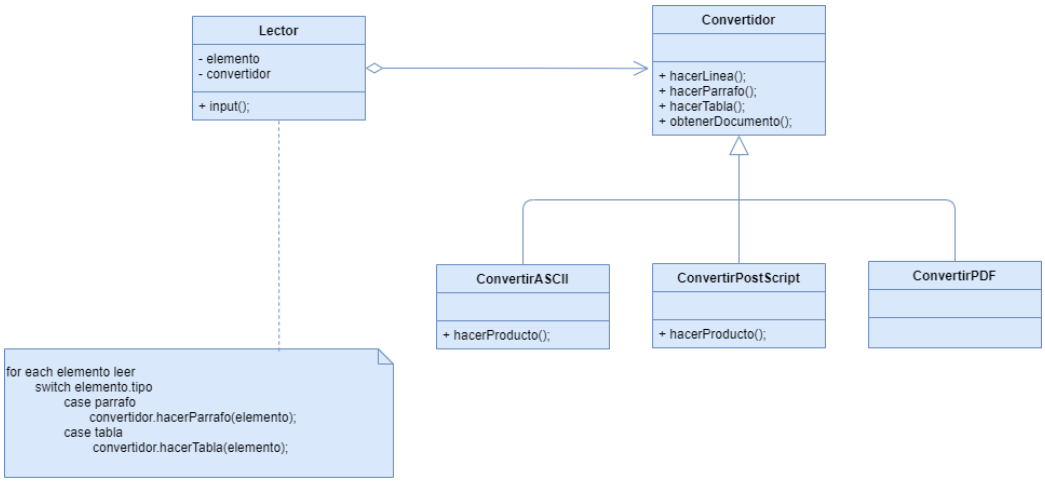

Figura 3.3: Diagrama UML Patrón Builder

3.1.4. Prototype

- **Función:** Crear nuevos objetos clonándolos de una instancia ya existente.

- **Estructura:** Como se aprecia en la figura 3.4

  Prototype es una interfaz que define la operación de clonado. Será implementada por todos los objetos que puedan ser clonados. En ocasiones es implementado como una clase abstracta. La clase cliente crea nuevos objetos mediante la clonación de los mismos. Mientras que PrototypeConcretoN es una clase que puede ser instanciada mediante la clonación de un prototipo. Implementa la interfaz Prototype.

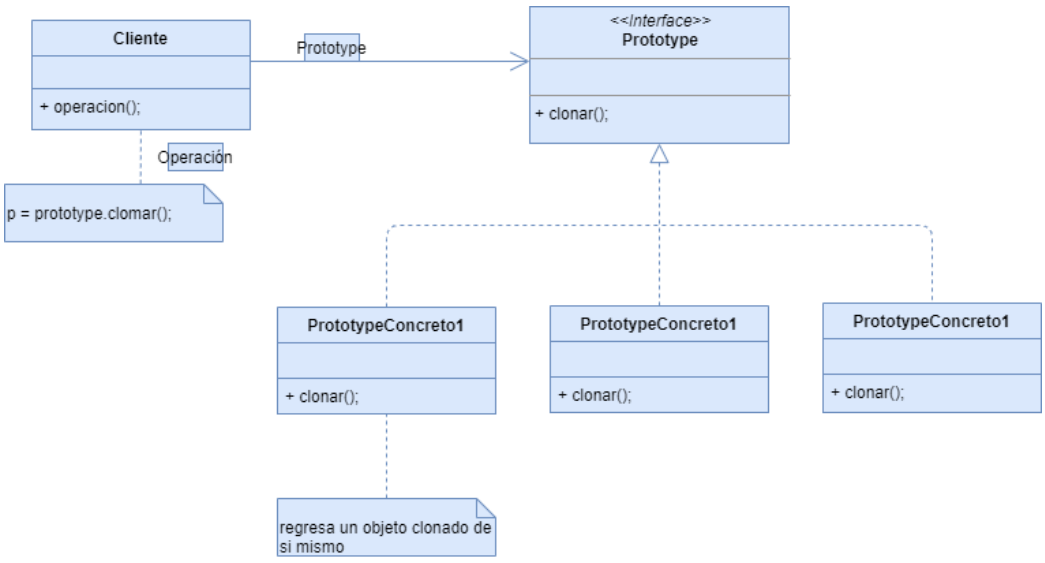

Figura 3.4: Diagrama UML Patrón Prototype

3.1.5. Singleton

- **Función:** Garantizar la existencia de una única instancia para una clase y la creación de un mecanismo de acceso global a la misma.

- **Estructura:** Como se aprecia en la figura 3.5

  La clase Singleton declara el método estático obtenerInstancia que devuelve la misma instancia de su propia clase. El constructor de Singleton debe estar oculto del código del cliente. Llamar al método obtenerInstancia debería ser la única forma de obtener el objeto Singleton.

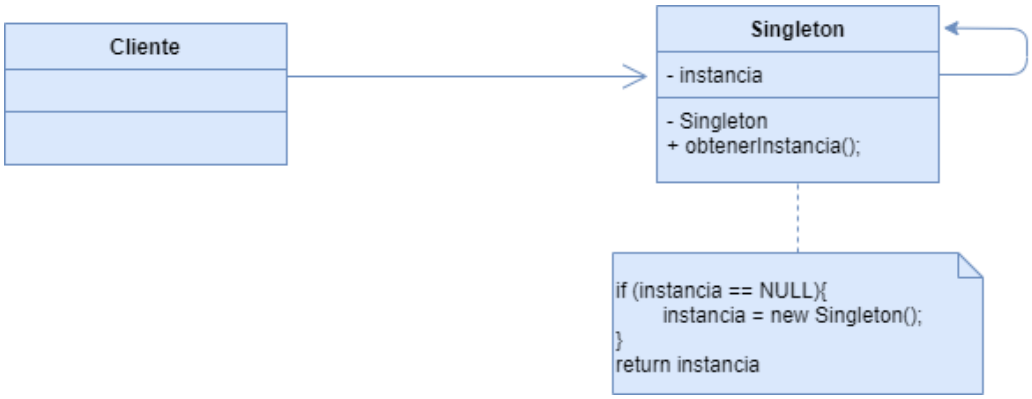

Figura 3.5: Diagrama UML Patrón Singleton

3.2. Patrones Estructurales

Los patrones estructurales tratan la composición de clases u objetos; para lo cual hacen uso de dos recursos, dependiendo del tipo de composición, es decir que para clases es necesaria la herencia mientras que para los objetos se describen formas de ensamblar objetos. Entre los principales patrones que se encuentran en este grupo podemos listar los siguientes:

3.2.1. Adapter

- **Función:** Adaptar una interface para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.
- **Estructura:** Esta implementación utiliza el principio de composición de objetos: el adaptador implementa la interfaz de un objeto y envuelve el otro. Se puede implementar en todos los lenguajes de programación populares.
  La estructura que cumple este patrón se muestra en la Figura 3.6.

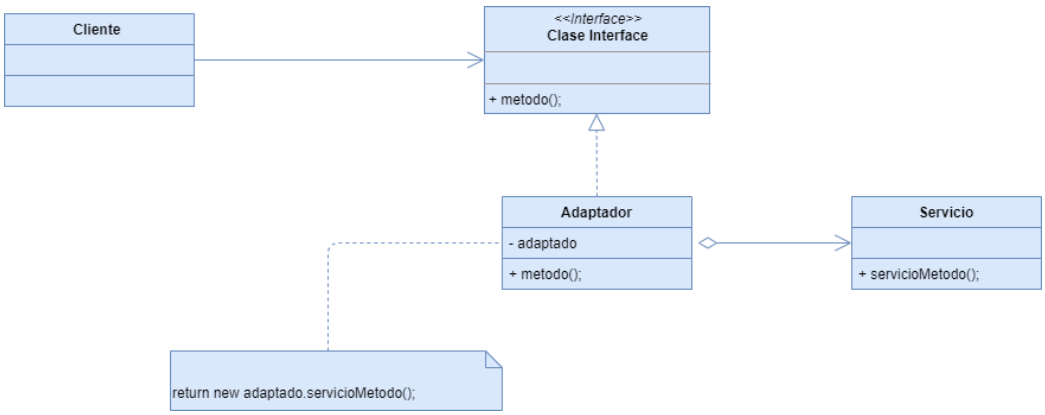

Figura 3.6: Diagrama UML Patrón Adapter

3.2.2. Bridge

- **Función:** Desacoplar una abstracción de su implementación.
- **Estructura:** El Cliente no quiere tratar con detalles dependientes de la plataforma. El patrón Bridge encapsula esta complejidad detrás de una 'envoltura' de abstracción. Bridge enfatiza la identificación y desacoplamiento de la abstracción de ïnterfaz"de la abstracción de ïmplementación".

  La estructura que cumple este patrón se muestra en la Figura 3.7.

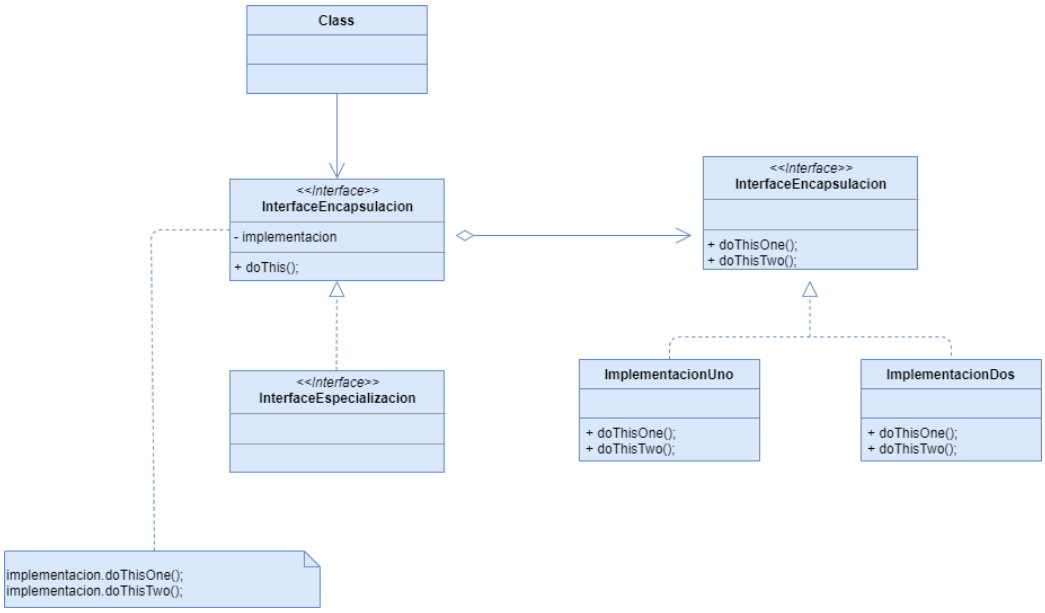

Figura 3.7: Diagrama UML Patrón Bridge

3.2.3. Composite

- **Función:** Manejar objetos compuestos como si se tratase de un simple.
- **Estructura:** La interfaz Componente describe operaciones que son comunes a elementos simples y complejos. El Cliente trabaja con todos los elementos a través de la interfaz del componente. Como resultado, el cliente puede trabajar de la misma manera con elementos simples o complejos.

  La estructura que cumple este patrón se muestra en la Figura 3.8.

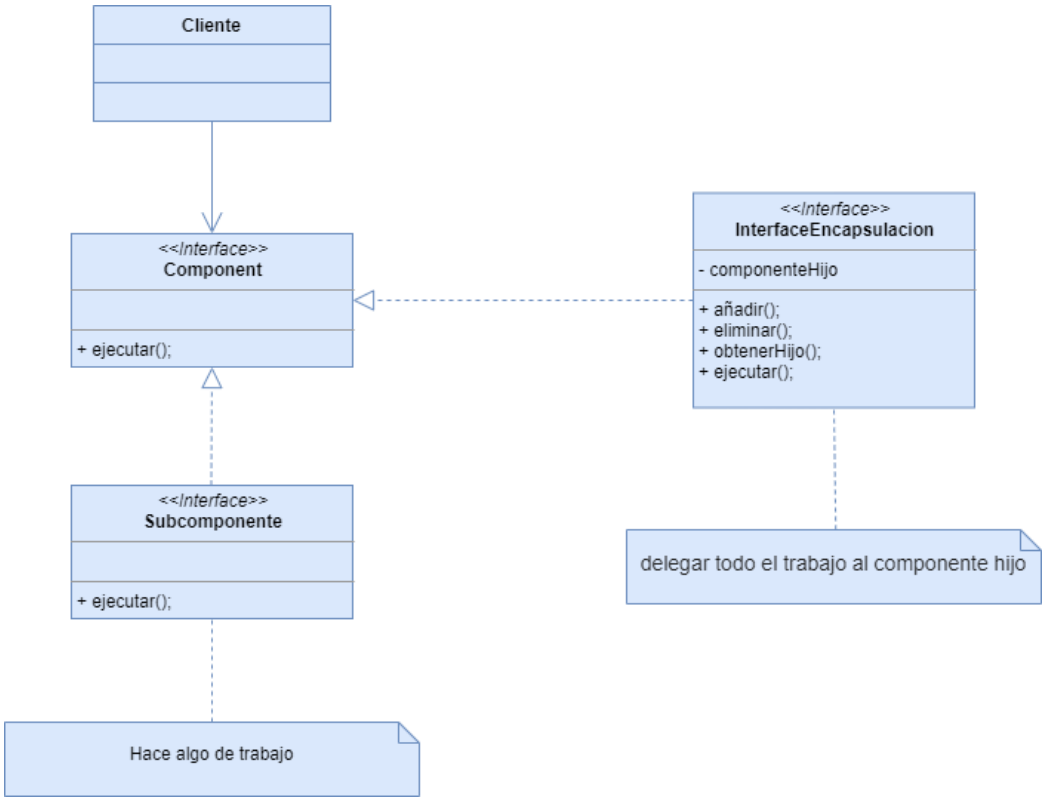

Figura 3.8: Diagrama UML Patrón Composite

3.2.4. Decorator

- **Función:** Facilitar la añadidura de funcionalidad a una clase dinámicamente.
- **Estructura:** El cliente siempre está interesado FuncionalidadCentral.hacerAlgo(). El cliente puede, o no, estar interesado en OpcionUno.hacerAlgo() y OpcionDos.hacerAlgo(). Cada una de estas clases siempre delega a la clase base Decorator, y esa clase siempre delega al objeto 'wrappee' contenido.
  La estructura que cumple este patrón se muestra en la Figura 3.9.

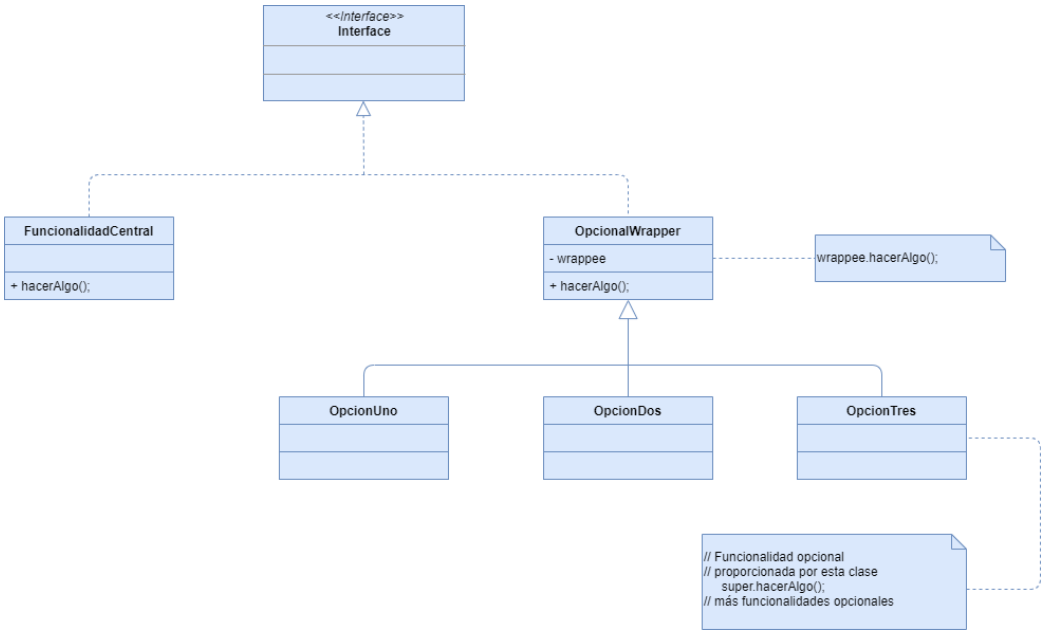

Figura 3.9: Diagrama UML Patrón Decorator

3.2.5. facade

- **Función:** Facilitar la añadidura de funcionalidad a una clase dinámicamente.
- **Estructura:** Facade proporciona acceso conveniente a una parte particular de la funcionalidad del subsistema. Sabe a dónde dirigir la solicitud del cliente y cómo operar todas las partes móviles.

  La estructura que cumple este patrón se muestra en la Figura 3.10.

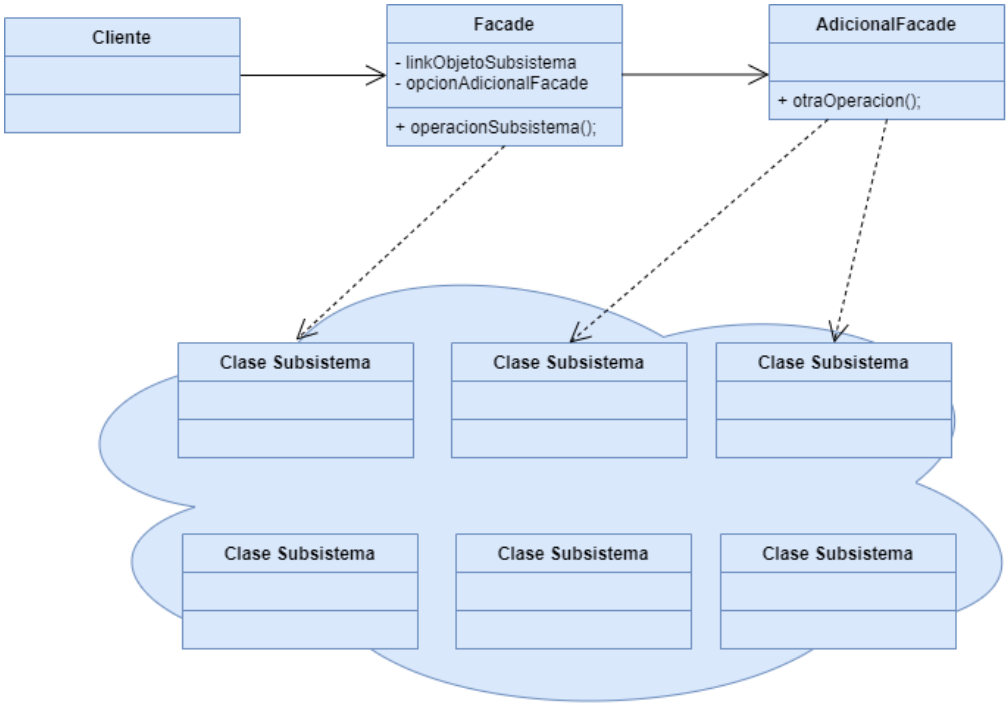

Figura 3.10: Diagrama UML Patrón Facade

3.2.6. Flyweight

- **Función:** Reducir la redundancia en situaciones en las que se presentan grandes cantidades de objetos que poseen idéntica información.

- **Estructura:** El patrón Flyweight se almacenan en el repositorio de una fábrica. El cliente se abstiene de crear Flyweights directamente y los solicita a la Fábrica. Flyweight no puede sostenerse por sí solo. Cualquier atributo que haga imposible compartir debe ser proporcionado por el cliente siempre que se haga una solicitud al Flyweight.

  La estructura que cumple este patrón se muestra en la Figura 3.11.

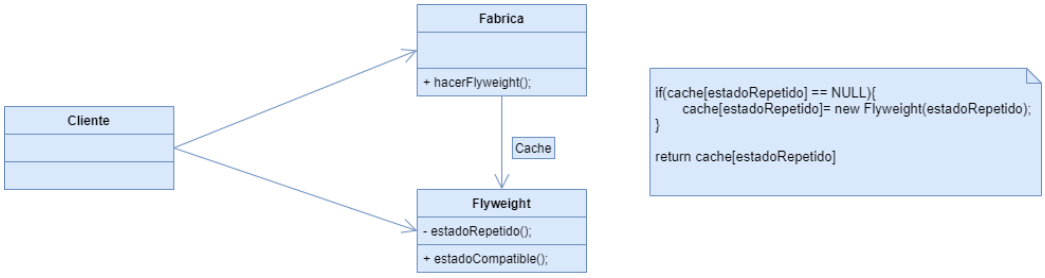

Figura 3.11: Diagrama UML Patrón Flyweight

3.2.7. Proxy

- **Función:** Mantiene un representante de un objeto.
- **Estructura:** El Cliente debe trabajar con servicios y proxies a través de la misma interfaz. De esta manera, puede pasar un proxy a cualquier código que espere un objeto de servicio. La clase Proxy tiene un campo de referencia que apunta a un objeto de servicio.

  La estructura que cumple este patrón se muestra en la Figura 3.12.

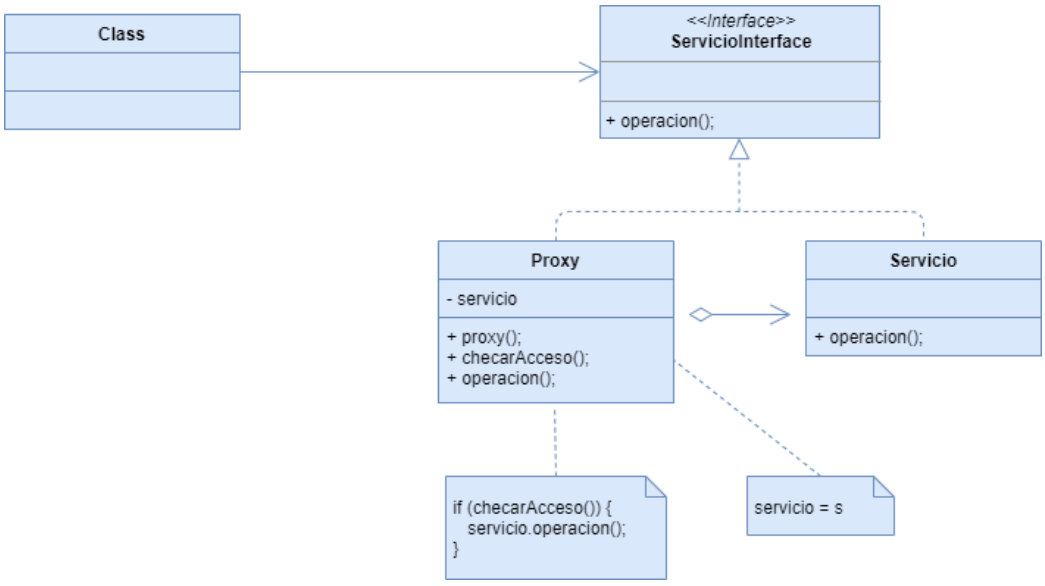

Figura 3.12: Diagrama UML Patrón Proxy

3.3. Patrones de Comportamiento

Los patrones comportamiento caracterizan el modo en que las clases y objetos interactúan y se distribuyen la responsabilidad. Este tipo de patrones, en las clases hacen uso de la herencia para describir algoritmos y flujos de control; mientras que los de objetos describen cómo cooperan un grupo de objetos para realizar una tarea que ninguno podría llevar a cabo por sí solo. Entre los principales patrones que se encuentran en este grupo podemos listar los siguientes:

3.3.1. Chain of responsibility

- **Función:** Establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada.

  La estructura que cumple este patrón se muestra en la Figura 3.13.

- **Estructura:** La clase Handler declara la interfaz, común para todas las clases ConcreteHandlers. Por lo general, contiene un solo método para manejar solicitudes, pero a veces también puede tener otro método para configurar el siguiente controlador en la cadena.

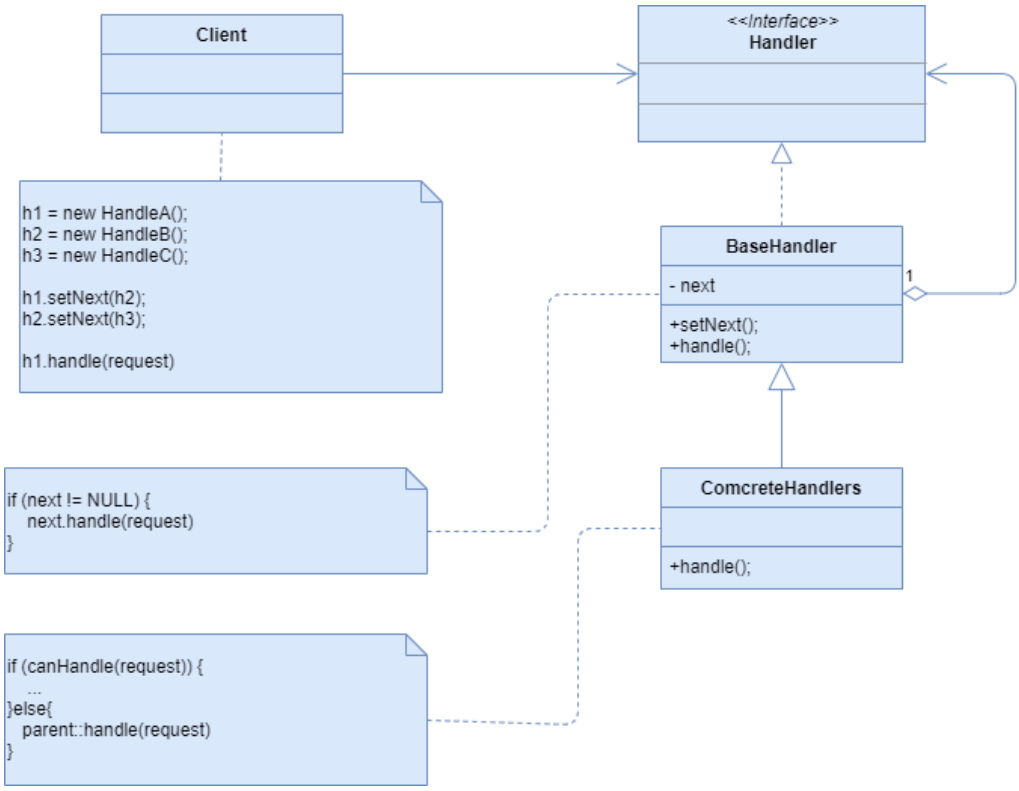

Figura 3.13: Diagrama UML Patrón Chain of responsibility

3.3.2. Command

- **Función:** Encapsular una operación en un objeto, permitiendo ejecutarla sin necesidad de conocer el contenido de la misma.

- **Estructura:** El Invocador no sabe quién es el Receptor ni la acción que se realizará, tan sólo invoca un Comando que ejecuta la acción adecuada.
  La estructura que cumple este patrón se muestra en la Figura 3.14.

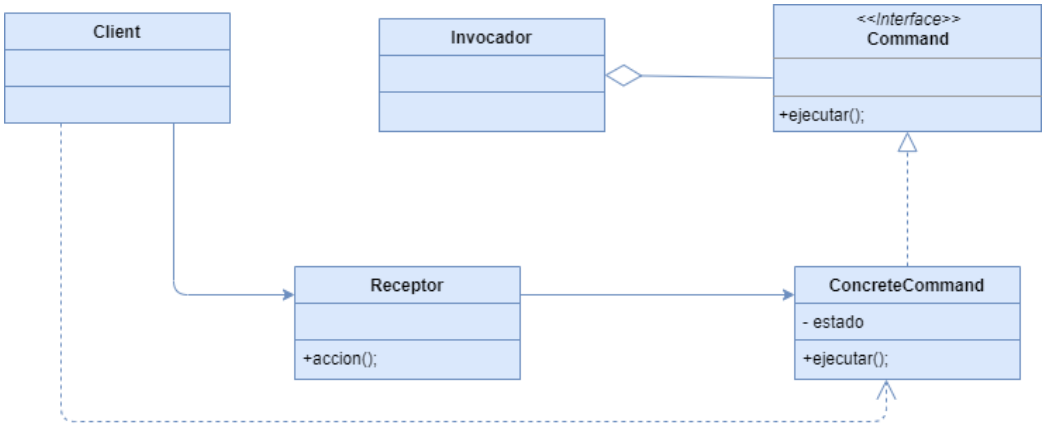

Figura 3.14: Diagrama UML Patrón Command

3.3.3. Iterator

- **Función:** Según el libro de Gang of Four, podemos realizar recorridos sobre objetos compuestos independientemente de la implementación de estos.

- **Estructura:** Este patrón de diseño nos resultará útil para acceder a los elementos de un array o colección de objetos contenida en otro objeto.
  La estructura que cumple este patrón se muestra en la Figura 3.15.

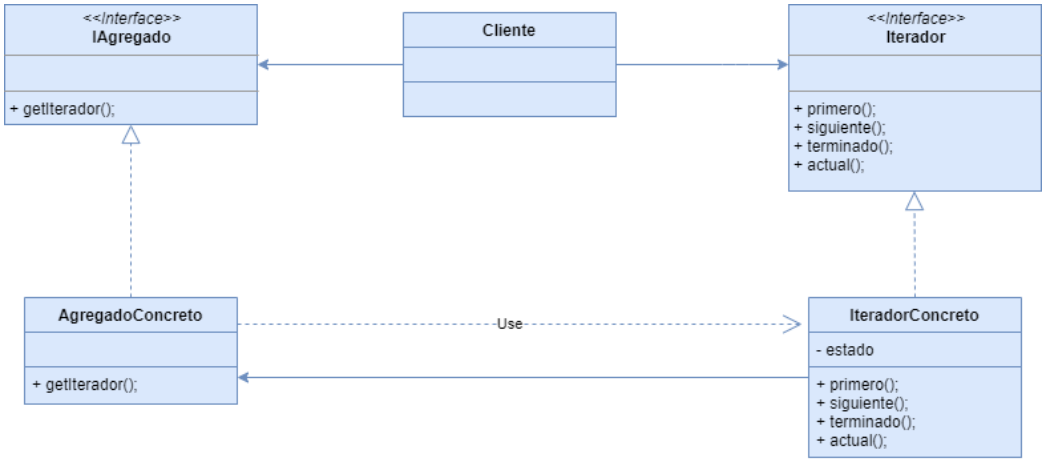

Figura 3.15: Diagrama UML Patrón Iterator

3.3.4. Interpreter

- **Función:** Definir una gramática para un lenguaje dado, así como las herramientas necesarias para interpretarlo.
- **Estructura:**
  - Cliente: Actor que dispara la ejecución del interpreter.
  - Contexto: Objeto con información global que será utilizada por el intérprete para leer y almacenar información global entre todas las clases que conforman el patrón, este es enviado al interpreter el cual lo replica por toda la estructura.
  - ExpresionAbstracta: Interface que define la estructura mínima de una expresión.
  - ExpresionTeminal: Se refiere a expresiones que no tienen más continuidad y al ser evaluadas o interpretadas terminan la ejecución de esa rama. Estas expresiones marcan el final de la ejecución de un sub-árbol de la expresión.
  - ExpresionNoTerminal: Son expresiones compuestas y dentro de ellas existen más expresiones que deben ser evaluadas. Estas estructuras son interpretadas utilizando recursividad hasta llegar a una expresión Terminal.

La estructura que cumple este patrón se muestra en la Figura 3.16.

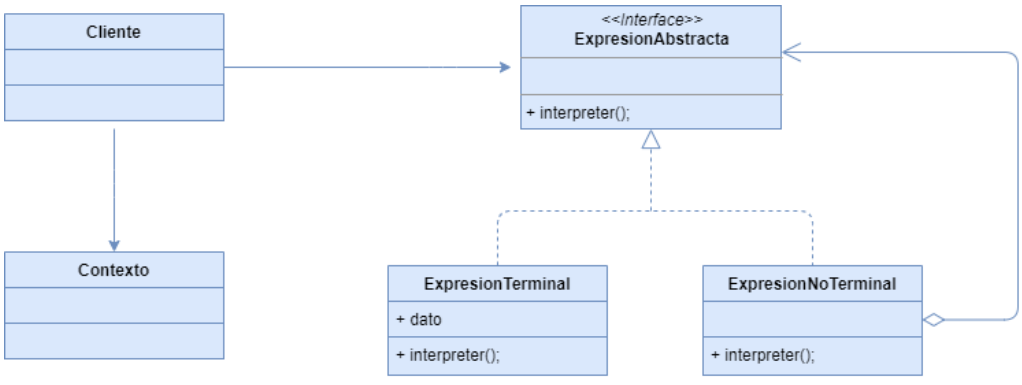

Figura 3.16: Diagrama UML Patrón Interpreter

3.3.5. Mediator

- **Función:** Definir un objeto que coordine la comunicación entre otros de distintas clases, pero que funcionan como un conjunto.
- **Estructura:**
  - Cliente: Componente que inicia la comunicación con el resto de los componentes por medio del mediador.
  - Componentes: Componentes que son parte de la red de comunicación por medio del mediador, éstos pueden ser diversos objetos que comparten el mismo mediador para comunicarse.
  - Mediador: Componente que sirve de mediador entre el resto de componentes, tiene como principal rol canalizar los mensajes entrantes al destinatario correspondiente.

La estructura que cumple este patrón se muestra en la Figura 3.17.

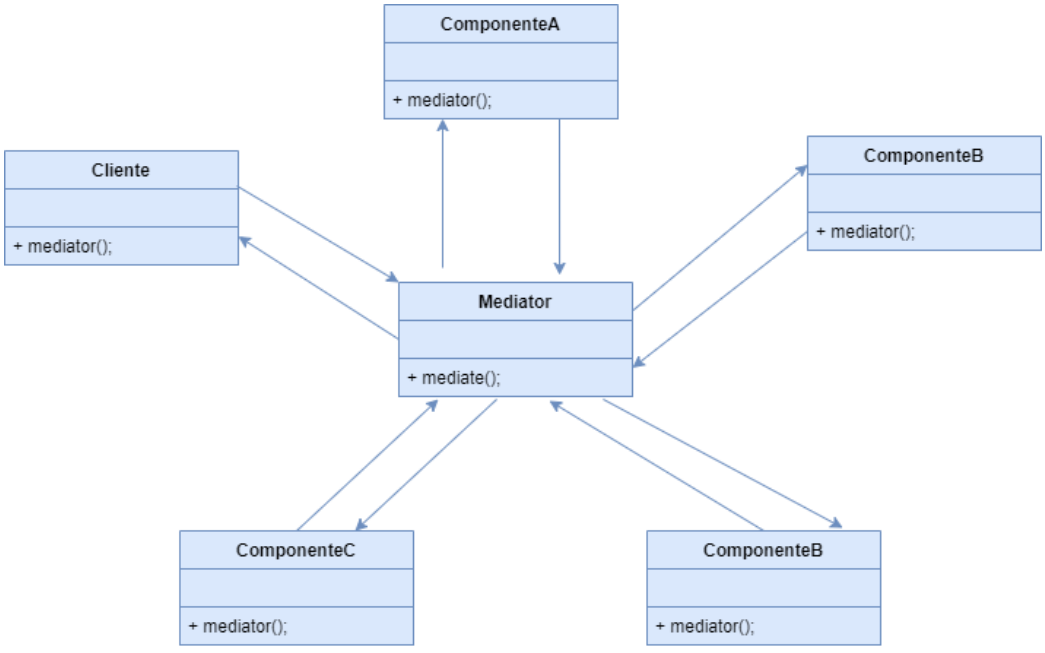

Figura 3.17: Diagrama UML Patrón Mediator

3.3.6. Memento

- **Función:** Volver a estados anteriores del sistema.
- **Estructura:**
  - Cliente: Componente que afecta al Originator y registra el nuevo estado con el Caretaker. En otras palabras, es quien realiza el cambio sobre el objeto y registra el estado.
  - Originador: Es el componente que cambia de estado.
  - Memento: Componente que almacena el estado del Originator en un momento determinado.
  - Vigilante: Componente que registra los cambios del Originator. Este componente nos permite viajar entre los distintos estados del Originator.

La estructura que cumple este patrón se muestra en la Figura 3.18.

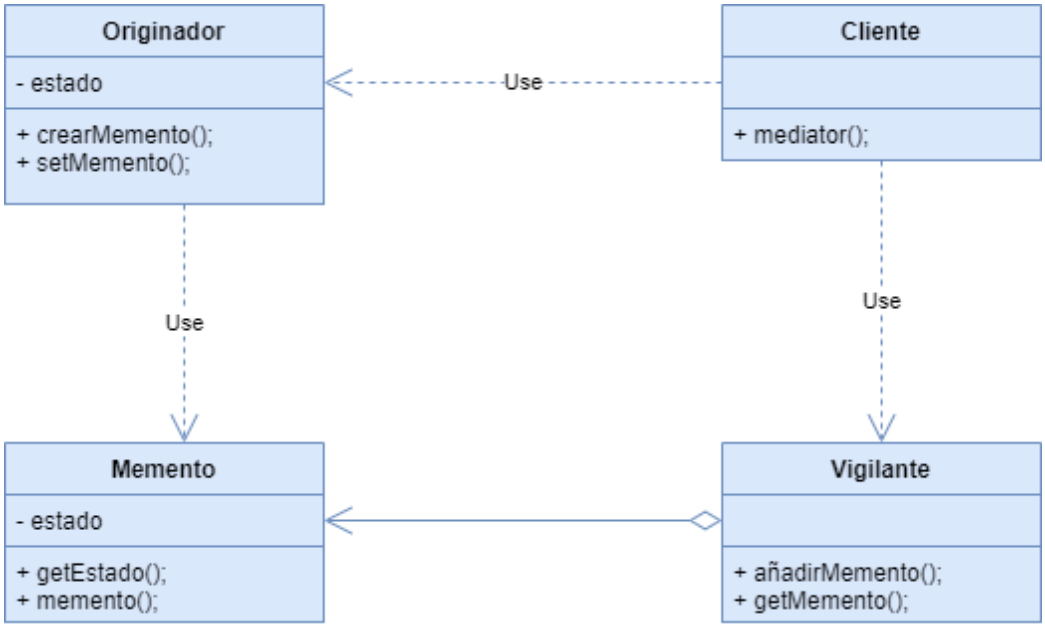

Figura 3.18: Diagrama UML Patrón Memento

3.3.7. Observer

- **Función:** Definir una dependencia de uno a muchos entre objetos, de forma que cuando uno cambie de estado se notifique y actualicen automáticamente todos los que dependen de el.

  - IObservable: Interface que deben de implementar todos los objetos que quieren ser observados, en ella se definen los métodos mínimos que se deben implementar.
  - ObservableConcreto: Clase que desea ser observada, ésta implementa IObservable y debe implementar sus métodos.
  - IObserver: Interfaces que deben implementar todos los objetos que desean observar los cambios de IObservable.
  - ObservableConcreto: Clase concreta que está atenta de los cambios de IObserver, esta clase hereda de IObserver y debe de implementar sus métodos.

  La estructura que cumple este patrón se muestra en la Figura 3.19.

- **Estructura:**

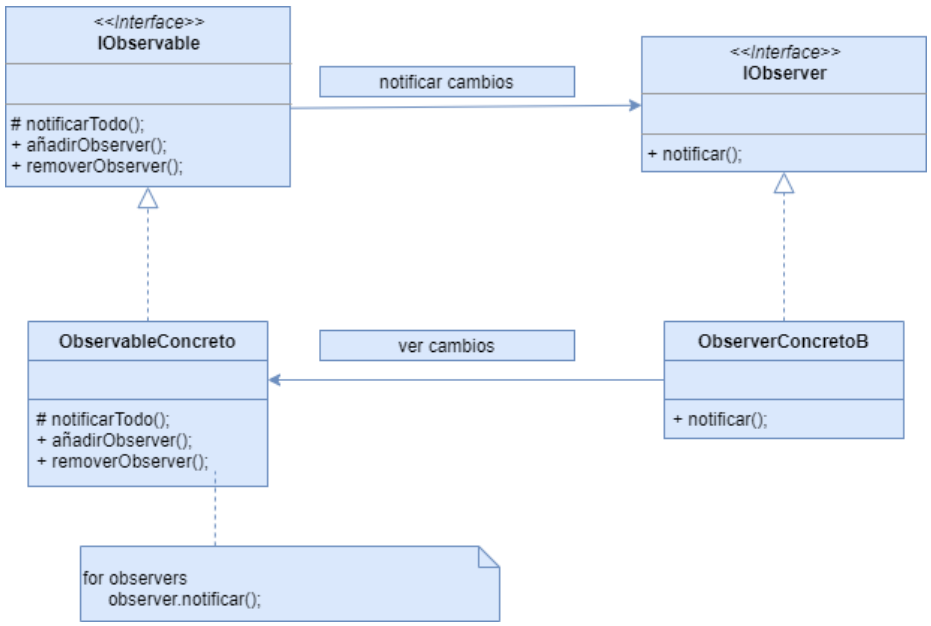

Figura 3.19: Diagrama UML Patrón Obsever

3.3.8. State

- **Función:** Alterar la conducta interna de un objeto.
- **Estructura:**
  - Contexto: Representa el componente que puede cambiar de estado, el cual tiene entre sus propiedades el estado actual. En el ejemplo de la máquina de refrescos, éste sería la máquina como tal.
  - EstadoAbstracto: Clase base para la generación de los distintos estados. Se recomienda que sea una clase abstracta en lugar de una interface debido a que podemos definir comportamientos por default y así afectar el funcionamiento de todos los estados.
  - EstadoConcreto: Cada uno de estos componentes representa un posible estado por el cual la aplicación puede pasar, por lo que tendremos un ConcreteState por cada estado posible. Esta clase debe de heredar de EstadoAbstracto.

La estructura que cumple este patrón se muestra en la Figura 3.20.

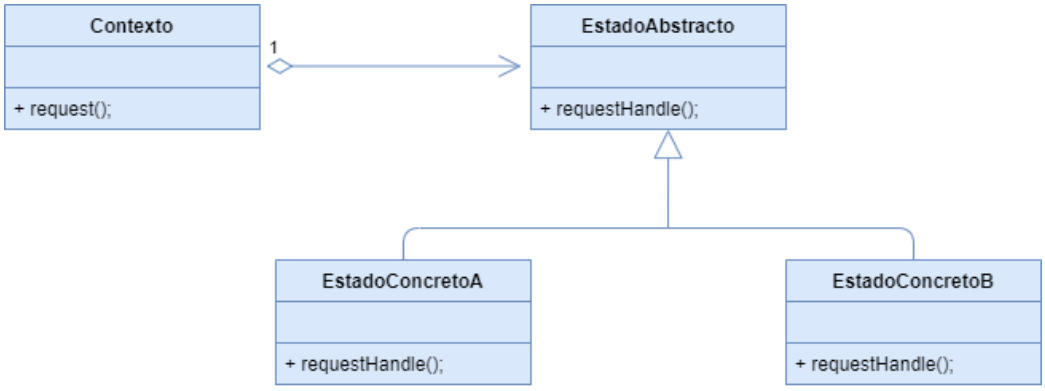

Figura 3.20: Diagrama UML Patrón State

3.3.9. Strategy

- **Función:** Definir una familia de algoritmos, encapsulados e intercambiables. La estrategia permite el algoritmo variar independientemente de los clientes que lo usan.

- **Estructura:**

  - Contexto: Componente que encapsula la estrategia a utilizar, tiene como característica que se puede establecer la estrategia a utilizar en tiempo de ejecución.
  - IStrategy: Interface en común que todas las estrategias deberán implementar. En esta interface se definen las operaciones que las estrategias deberán implementar.
  - StrategyConcreto: Representa las estrategias concretas, las cuales heredan de IStrategy.

La estructura que cumple este patrón se muestra en la Figura 3.21.

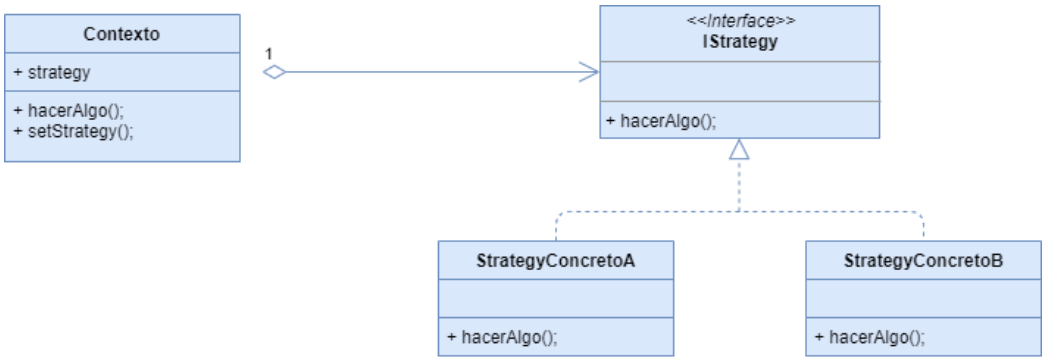

Figura 3.21: Diagrama UML Patrón Strategy

3.3.10. Template method

- **Función:** Redefinir subclases siguiendo ciertos pasos de un algoritmo sin cambiar la estructura del mismo
- **Estructura:**
  - Cliente: Es el componente que acciona la ejecución del templete.
  - AbstractTemplete: Clase abstracta con una serie de operaciones que definen los pasos para llevar a cabo la ejecución del algoritmo. La clase tiene el método templeteMethod que ejecuta en orden los métodos paso1, paso2, paso3.
  - Implementacion: Clase que representa un temple concreto, para lo cual deberá de heredar de AbstractTemplete e implementar los métodos de ésta.

La estructura que cumple este patrón se muestra en la Figura 3.22.

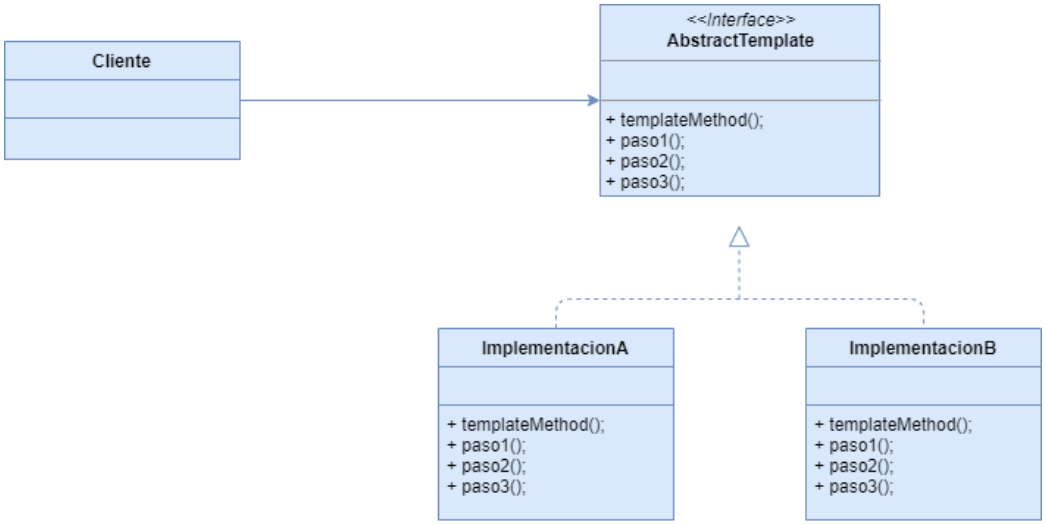

Figura 3.22: Diagrama UML Patrón Template method

3.3.11. Visitor

- **Función:** Representar un funcionamiento a ser realizado por los elementos de la estructura del objeto, además permite definir un nuevo funcionamiento sin cambiar las clases de los elementos en que opera.

- **Estructura:**

  - Cliente: Componente que interactúa con la estructura (elemento) y con el Visitante, éste es responsable de crear los visitantes y enviarlos al elemento para su procesamiento.
  - Elemento: Representa la raíz de la estructura, en forma de árbol, sobre la que utilizaremos el Visitante. Este objeto por lo general es una interface que define el método accept y deberán implementar todos los objetos de la estructura.
  - ElementConcreto (A,B): Representa un hijo de la estructura compuesta, la estructura completa puede estar compuesta de un gran número de estos objetos y cada uno deberá implementar el método accept.
  - IVisitor: Interface que define la estructura del visitante, la interface deberá tener un método por cada objeto que se requiera analizar de la estructura (elemento).
  - VisitorConcreto (A,B): Representa una implementación del visitante, esta implementación puede realizar una operación sobre el element. Es posible tener todos los ConcreteVisitor necesarios para realizar las operaciones que necesitemos.

  La estructura que cumple este patrón se muestra en la Figura 3.23.

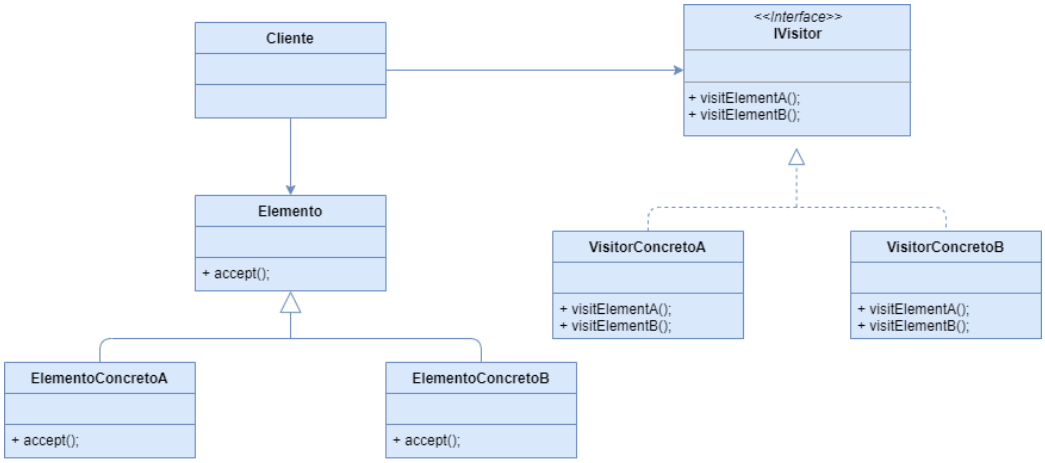

Figura 3.23: Diagrama UML Patrón Visitor

Capítulo 4

Ingeniería de software utilizando patrones de diseño

4.1. Beneficios del uso de patrones de diseño

Los patrones de diseño constituyen una poderosa arma para enfrentar los problemas del desarrollo de software, sin embargo, se deben manejar con cautela, pues la inclusión de patrones innecesarios puede llevar al incremento de la complejidad, por lo que para aprovechar al máximo su uso se debe tener muy en cuenta qué patrones se necesita utilizar, y cuáles están asociados con el funcionamiento de éstos, para evitar la introducción de patrones incompatibles entre sí.

La idea del uso de patrones de diseño es la de poder establecer soluciones a problemas recursivos en el desarrollo de software, de manera que se los pueda utilizar siempre que se presente el mismo problema; para poder cumplir con esto, las soluciones deberán ser bien definidas.

"Nuestro diseño deberá ser específico para el problema que tenemos en frente, pero deberá ser también lo suficientemente generalizado para poder orientarlo a la solución de futuros problemas y requerimientos" (Campo 2009)

Los patrones de diseño aportan en varios aspectos a mejorar el proceso de desarrollo de software, entre los aportes más importantes debemos destacar:

- **Ofrecen mayor utilidad que los lenguajes de programación** Para un gran porcentaje de los problemas que se presentan en el desarrollo, los patrones de diseño proveen mayor utilidad que los lenguajes de programación, ya que al ser empleados desde el proceso de diseño, la lógica de la solución está claramente descrita y no da lugar a ambigüedades. Además los patrones son fácilmente aplicables a cualquier lenguaje orientado a objetos.

- **Proveen una solución más equilibrada** Debido a que el empleo de patrones requiere un análisis profundo, se puede encontrar el punto de equilibrio entre las restricciones y objetivos de la solución, obteniendo como resultado un si

- **Ofrecen un alto nivel de cohesión** Los patrones de diseño permiten que se creen clases de cooperación mutua, pero al mismo tiempo las mantienen lo suficientemente independientes del funcionamiento de otras, por lo cual se obtiene un bajo acoplamiento de los componentes del software y una alta cohesión entre los módulos.

- **Fácil identificación de objetos** Permiten identificar abstracciones menos obvias pero más efectivas que aquellas que saltan fácilmente a la vista al momento de diseñar una solución.

- **Reutilización de software** Este beneficio se deriva de la definición misma de los patrones de diseño, y constituye en una fortaleza para aquellos que los dominan, pues consiste en optimizar el diseño de una solución a un problema de desarrollo determinado, la garantía del uso de los patrones de diseño radica en que se respaldan en la experiencia de la reutilización de los mismo, al enfrentarlos varias veces a problemas similares, pero en entornos diferentes.

- **Reusabilidad, extensibilidad y mantenimiento** Esto gracias a que cada patrón permite controlar el tamaño y ubicación de los elementos a ser implementados, garantizando un alto nivel de calidad y simplificando el trabajo en grupo por el vocabulario estándar que se maneja en cada componente, facilitando principalmente el mantenimiento adaptativo (integración de nuevos requerimientos

o cambios de los existentes).

- **Ofrecen soluciones concretas** Esta podría ser la más importante ventaja del uso de patrones de diseño, ya que estos proveen una solución concreta a un problema haciendo frente a casi todas las metodologías o tecnologías empleadas y además considerando las consecuencias, beneficios y viabilidad de la solución propuesta; esto se debe a que funcionan como una guía para resolver problemas comunes de programación.

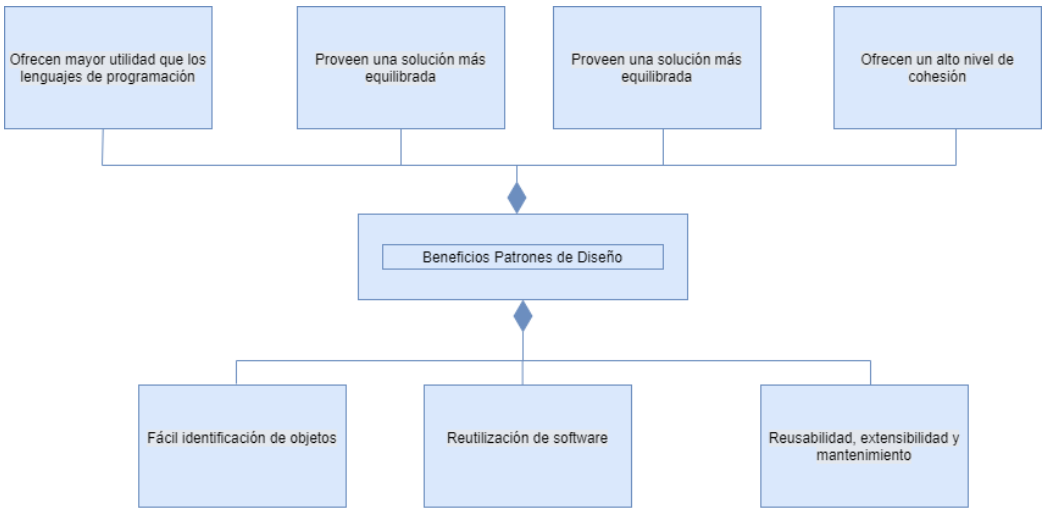

Figura 4.1: Beneficios del uso de patrones de diseño vistos cómo composicion

4.2. Desarrollo de Software con Patrones de Diseño

El desarrollo de software con patrones de diseño se fundamenta en la mejora de la calidad del sistema generado y las fortalezas; esta forma de desarrollo se basa en las ventajas que ofrecen los patrones de diseño.

Pero cabe recalcar que a pesar de los grandes beneficios que proveen los patrones de diseño, también se deben considerar las debilidades, que si bien es cierto son pocas, estas son lo suficientemente considerables, al punto que pueden anular los beneficios que los patrones ofrecen.

Ventajas

- Fácil integración

  Los sistemas desarrollados con patrones de diseño, además de facilitar la integración de nuevos requerimientos o cambio de los existentes, permiten realizar esto sin afectar la estructura del sistema; la clave para lograrlo es anticiparse a los nuevos requisitos de tal manera que la selección del patrón idóneo asegure una evolución adecuada.

- Capturar las mejores prácticas

  Al desarrollar software, el análisis y entendimiento de los diseños exitosos y fallidos es la clave para facilitar la aplicación de los diferentes patrones en cada una de las etapas de desarrollo, ya que se capturan las mejores prácticas y se evita el cometer los mismos errores, reduciendo notablemente la cantidad de código y simplificando la lógica de la solución.

- Soluciones probadas

  El empleo de patrones de diseño en el desarrollo de Software permite obtener soluciones probadas y validadas a un problema particular.

- Proceso de diseño serio

  Emplear patrones en el proceso de desarrollo es el primer paso para iniciar la etapa de diseño de Software serio; en el que se tiene estándares de uso, se facilita el aprendizaje de los componentes ya existentes, todo esto gracias a la documentación de calidad y vocabulario común.

- Incremento de la eficiencia

  Al usar patrones de diseño en el desarrollo de software, el incremento de eficiencia en el diseño es muy notable, ya que ante un problema no es necesario generar una nueva solución: basta con analizar las existentes y tomar las más adecuada con la tranquilidad de saber que estas soluciones son probadas con anterioridad y el riesgo de fallo es prácticamente nulo.

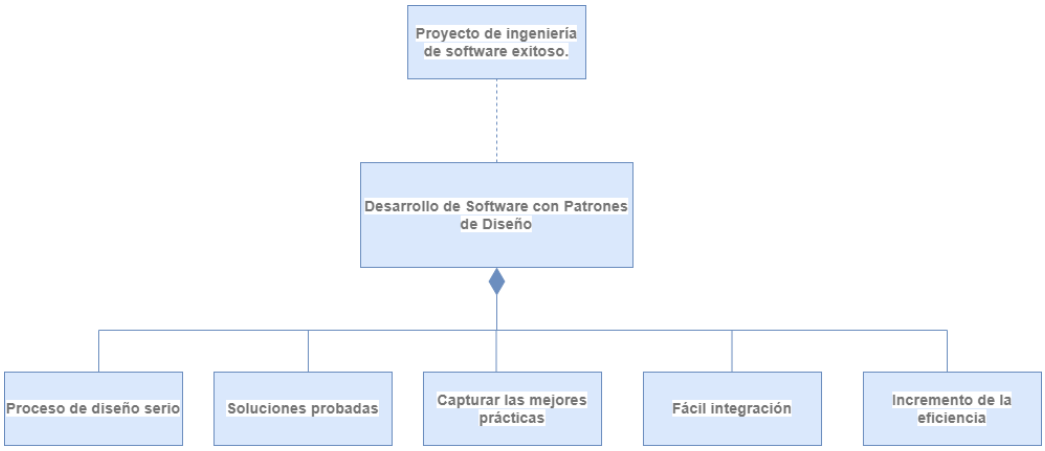

Figura 4.2: Las ventajas que nos ofrecen los patrones de diseño nos garantizan un proyecto de ingeniería de sotfware exitoso

Desventajas

- Disminución de la capacidad de iniciativa

  Al usar componentes, previamente desarrollados, se minimiza la capacidad de iniciativa de las personas y les restan creatividad, ya que se basan en la utilización de patrones conocidos y no se realiza un esfuerzo en buscar nuevas soluciones.

- Mapeo e integración de soluciones

  Si bien es cierto que los patrones de diseño describen paso a paso como cubrir un requerimiento o solucionar un problema específico, se debe tener presente que 26 ningún patrón explica como se lo debe implementar, ni cómo se lo debe integrar al funcionamiento de otros patrones. Al no contar con una guía o referencia que indique como realizar esto, se puede perder más tiempo y recursos tratando de realizar dicho mape

- Incremento de la complejidad debido a la falta de experiencia

  Es necesario tener experiencia para reconocer el problema y dar una adecuada solución; es decir que se necesita experiencia en el uso de patrones de diseño para poder utilizarlos efectivamente. En caso de no conocer y determinar correctamente el patrón, se estaría simplemente agregando complejidad al sistema, o peor aún se puede llegar a incluir patrones in-

compatibles entre sí, lo cual haría imposible integrar los módulos de un sistema, y que se deba desarrollar nuevamente la aplicación desde cero.

4.3. Desarrollo de Software sin Patrones de Diseño

Ventajas

- Reducción del riesgo

  Dado que se emplea conceptos conocidos, se optimizan recursos; además se minimizan el tiempo de desarrollo empleando prácticas conocidas y ya probadas con anterioridad en otros proyectos.

- Falta de experiencia

  La falta de experiencia no es un obstáculo para desarrollar sistemas de calidad, eficaces y eficientes.

Desventajas

- Mantenimiento

  Dada la falta de documentación adecuada de un sistema se dificulta el mantenimiento adaptativo y correctivo de la solución.

- Uso de estándares

  Dado que en el proceso de desarrollo de Software común, no se obliga a usar estándares resulta compleja la reutilización de soluciones previas en sistemas similares.

4.4. Comparación

Se debe tener presente que para asegurar el éxito del desarrollo con patrones, la mayoría de ocasiones será necesario capacitar sobre el uso de patrones de

diseño, a los desarrolladores de software; lo cual implica necesariamente incrementos en el presupuesto y plazos del proyecto, es decir mayor riesgo que puede conducir a errores e incluso al fracaso, es por esto que se recomienda iniciar la aplicación de patrones de diseño en proyectos desarrollados por versiones cuya complejidad se incremente de manera controlada, aplicando patrones simples y fáciles de entender por todos los miembros del equipo.(Guerrero, Suárez, and Gutiérrez 2013)

Además es sumamente importante que los desarrolladores de mayor experiencia documenten de manera detallada sus soluciones para que estas puedan ser empleadas por los demás miembros del equipo, facilitando el proceso de análisis y selección de patrones.

Capítulo 5

Cómo usar un patrón de diseño

Para poder explicar a detalle cómo se utiliza cada patrón de diseño ha sido necesaria la investigación detallada de los patrones, sus orígenes, aplicaciones, ventajas y desventajas, de tal manera que se garantice un uso sencillo sin necesidad de contar con una amplia experiencia en el manejo de patrones de diseño.

5.1. Objetivos

- Comprender y tomar decisiones adecuadas, para el uso de patrones de diseño en el desarrollo de software.
- Describir el funcionamiento de los principales patrones de diseño y ubicarlos, de acuerdo a su aporte, en la solución de problemas de las diferentes etapas del proceso de desarrollo de Software.
- Brindar una serie de pasos que garanticen el éxito al aplicar los patrones de diseño y faciliten el desarrollo de aplicaciones.

5.2. Alcance

El siguiente capítulo describe el funcionamiento y uso de los patrones de diseño en la implementación de aplicaciones de software, de manera clara y precisa; además,

provee una serie de pasos a seguir para la implementación exitosa de los patrones en sistemas de Software, tratando de obtener soluciones reusables.

No se excluye la posibilidad de enfoques alternativos que ofrezcan más ventajas o facilidades en el proceso de aprendizaje y uso de los patrones de diseño para solucionar problemas puntuales del desarrollo de software.

5.3. Definiciones

1. **Campo de acción "Objetos":** Es una entidad provista de métodos o mensajes a los cuales responde (comportamiento); atributos con valores concretos (estado); y propiedades (identidad). (Bahit 2011)

2. **Campo de acción "Clases":** Una clase es un modelo que se utiliza para crear objetos que comparten un mismo comportamiento, estado e identidad. (Bahit 2011)

3. **Método:** Es el algoritmo asociado a un objeto que indica la capacidad de lo que éste puede hacer. (Bahit 2011)

4. **Acoplamiento:** Es una medida de la fuerza con que una clase está conectada a otras clases, con que las conoce y con que recurre a ellas. (Visconti and Astudillo 2012)

5. **Cohesión:** Es una medida de cuán relacionadas y enfocadas están las responsabilidades de una clase. (Visconti and Astudillo 2012)

6. **Escalabilidad:** Es la capacidad del sistema para crecer ante un incremento de la concurrencia de usuarios, sin aumentar su complejidad ni disminuir su rendimiento. El crecimiento de un sistema distribuido puede introducir cuellos de botella y latencias que degradan su rendimiento. La escalabilidad puede referirse a diversos parámetros del sistema, entre ellos, cantidad de tráfico de red

adicional, facilidad para agregar más capacidad de almacenamiento y cantidad de transacciones adicionales que se pueden procesar. (Del Pilar)

7. **Mantenimiento:** Esto incluye modificaciones y actualizaciones para prevenir futuros problemas de software. Su objetivo es asistir a problemas, que no son significativos en este momento, pero puede causar problemas serios en el futuro. (Del Pilar)

8. **Reutilización de código:** Se refiere al comportamiento y a las técnicas que garantizan que una parte o la totalidad de un programa informático existente se pueda emplear en la construcción de otro programa. De esta forma se aprovecha el trabajo anterior, se economiza tiempo, y se reduce la redundancia. (Autores 2011)

5.4. Guía Para el Uso de Patrones de Diseño

En esta sección se describen los pasos para poder utilizar patrones de diseño en el desarrollo de software, las secciones incluidas contienen un catálogo de patrones y recomendaciones para realizar el desarrollo de software utilizándolos correctamente.

5.4.1. Abstract Factory

Objetivo: Permitir la creación de grupos de objetos relacionados o dependientes, sin especificar sus clases concretas.

Aplicaciones: El uso del patrón Abstract Factory se recomienda en los siguientes casos:

- El sistema a desarrollarse debe ser independiente de la forma en que se crean sus objetos.

- El sistema debe ser configurado con uno o varios conjuntos de productos posibles.

- Un grupo de objetos relacionados es diseñado para usarse en conjunto, y es necesario reforzar esta restricción.

Patrones de Diseño Colaboradores:

- Una fábrica abstracta siempre es un objeto de instancia única, correspondiente al patrón Singleton.
- Se puede implementar una fábrica abstracta como un patrón Factory Method o un patrón Prototype.

Problematica: Para implementar grupos de objetos es necesario especificar sus nombres exactos en la clase cliente, es decir que en caso de que se necesite modificar un objeto también se debe cambiar la clase que lo instancia.

Solución: El patrón de diseño Abstract Factory aísla a las clases "clientes" de los nombres y definiciones de las clases "producto" (soporta la creación implícita de objetos); de manera que, la única forma de conseguir un "producto" es a través de una clase Abstract Factory, de esta manera el conjunto de productos pueden modificarse fácilmente sin necesidad de actualizar cada clase cliente.

Campo de acción: Aplicado a nivel de objeto.

Diagrama o Implementación:

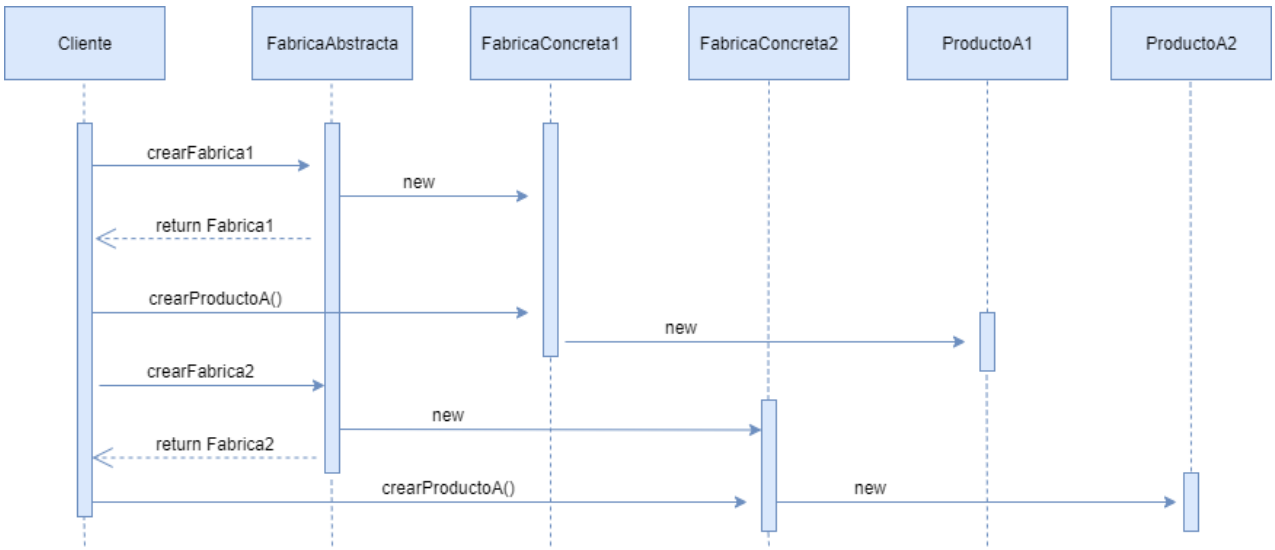

Figura 5.1: Diagrama de secuencia Abstract Factory

- Clase cliente solicita la creación del FabricaConcreta1 a la clase AbstractFactory.
- Clase AbstractFactory crea una instancia del FabricaConcreta1 y la regresa.
- El cliente le solicita al FabricaConcreta1 la creación de un ProductoA.
- La clase FabricaConcreta1 crea una instancia del ProductoA1 el cual es parte de la familia1 y lo regresa.
- Clase cliente está vez solicita la creación del FabricaConcreta2 a la clase AbstractFactory.
- La clase AbstractFactory crea una instancia del FabricaConcreta2 .
- El cliente le solicita a FabricaConcreta2 la creación de un ProductoA .
- La clase FabricaConcreta2 crea una instancia del ProductoA2 el cual es parte de la familia2 y lo regresa.

5.4.2. Builder

Objetivo: Separar la construcción de un objeto complejo de su representación, de manera que se puedan crear diferentes representaciones de dicho objeto utilizando el mismo proceso de construcción.

Aplicaciones:

- Se necesita tener un control minucioso sobre el proceso de creación de un objeto.
- El algoritmo de creación de objetos complejos debe ser independiente de las partes que conforman al objeto y de la forma en que ellas son ensambladas.
- El proceso de construcción debe permitir diferentes representaciones del objeto que está siendo construido.

Patrones de Diseño Colaboradores: El objeto construido mediante un patrón Builder generalmente es un objeto Composite.

Campo de acción: Aplicado a nivel de objeto.

Problematica: Para emplear un mismo proceso de construcción para diferentes objetos, se debe realizar modificaciones el algoritmo o estrategia de construcción; lo cual incrementa la complejidad y reduce la modularidad del sistema.

Solución: El patrón de diseño Builder permite ubicar la lógica de construcción de un objeto fuera del mismo y programarla en una clase independiente (Constructor), la cual devolverá al cliente una representación determinada del objeto sin necesidad de que éste o la clase cliente sepan la secuencia de creación de dicho objeto.

Diagrama o Implementación:

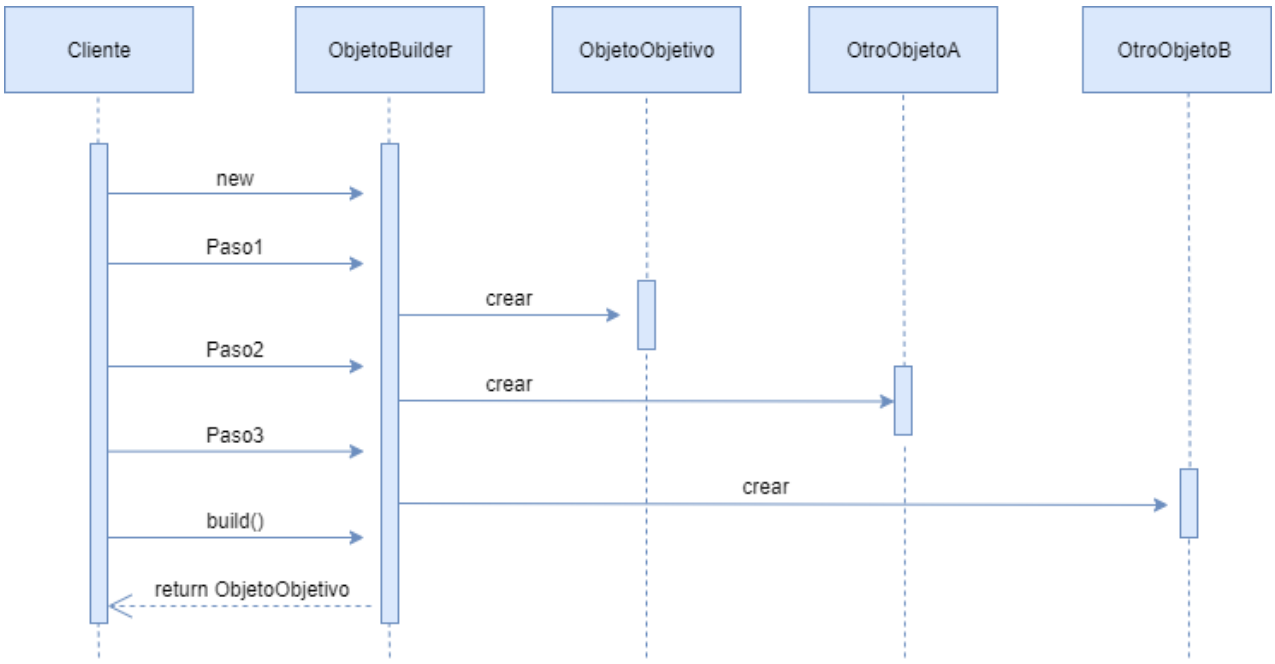

Figura 5.2: Diagrama de secuencia Builder

A continuación se explica el diagrama de la figura 5.2

- Clase cliente crea una instancia del ObjectoBuilder .
- La clase cliente cliente ejecuta el paso 1 de la creación en el ObjectoBuilder.
- Internamente el ObjectoBuilder crea al ObjetoObjetivo.
- Clase cliente ejecuta el paso 2 de la creación en el ObjectoBuilder.
- Internamente el ObjectoBuileder crea una instancia de OtroObjetoA.
- El cliente ejecuta el paso 3 de la creación en el ObjectoBuilder.
- Internamente el ObjectoBuilder crear la instancia de OtroObjetoB.
- Clase cliente solicita a la clase ObjectoBuilder la creación del ObjetoObjetivo, éste toma todos los objetos creados anteriormente, los asocia al ObjetoObjetivo y lo regresa.

5.4.3. Factory Method

Objetivo: Definir una interfaz para la creación de un objeto, permitiendo a las subclases decidir cual clase instanciar; es decir que el patrón de diseño Factory Method permite a una clase asumir la instanciación de una subclase.

Aplicaciones:

- Es importante la flexibilidad del sistema.
- Una clase es incapaz de anticipar que objetos debe implementar.
- Se requiere manejar varios objetos de manera similar, pero que los resultados del uso de los mismos sea diferente.

Patrones de Diseño Colaboradores:

- Un patrón Factory Method generalmente implementa el patrón Abstract Factory.
- Generalmente son llamados dentro de un patrón Template Method.

Campo de acción: Aplicado a nivel de clases.

Problematica: Para permitir que una clase instancie una subclase se debe especificar claramente los nombres de los objetos a ser creados dependiendo del estado de la aplicación; esto incrementa la complejidad y dificulta el mantenimiento de la aplicación

Solución: El patrón de diseño Factory Method permite separar la lógica de creación de objetos de la clase cliente; de tal manera que los parámetros enviados por dicha clase son los que definen el objeto; es decir que se puede crear objetos con funcionalidad diferente a través de la misma interfaz.

Diagrama o Implementación:

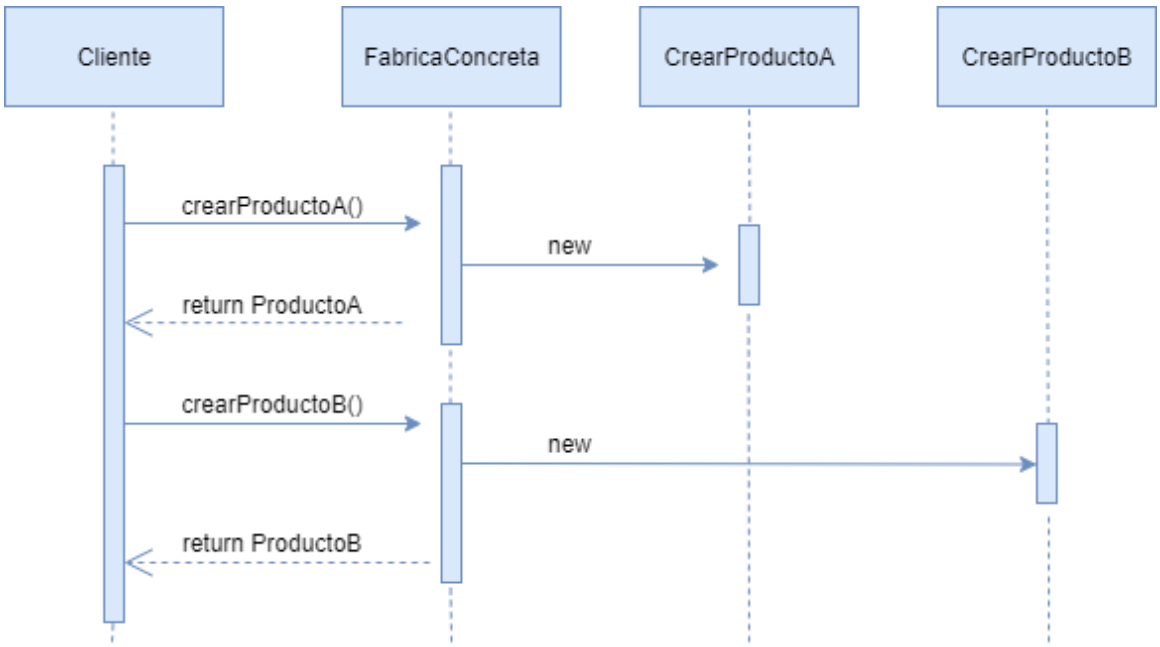

Figura 5.3: Diagrama de secuencia de Factory Method

A continuación se explica el digrama de la figura 5.3

- Clase cliente le solicita a la clase FabricaConcreta la creación del ProductoA .
- FabricaConcreta localiza la implementación concreta de ProductoA y crea una nueva instancia.
- La FabricaConcreta regresa crearProductoA (ProductoConcreto) creado.
- El cliente le solicita a la FabricaConcreta la creación del ProductoB.
- La FabricaConcreta localiza la implementación concreta del ProductoB y crea una nueva instancia.
- FabricaConcreta regresa crearPoruductoB (ProductoConcreto) creado.

5.4.4. Singleton

Objetivo: Garantizar la creación de una instancia única para una clase dada.

Aplicaciones:

- El sistema exige exactamente una instancia de una clase, la cual debe ser accesible para los clientes desde un punto de acceso bien definido.
- La única instancia debe ser extendida a subclases y los clientes deben ser capaces de usarla sin modificar su código.

Patrones de Diseño Colaboradores: Un patrón Singleton es frecuentemente relacionado y empleado en la implementación del patrón Abstract Factory, sobre todo cuando se trata de una fábrica concreta.

Campo de acción: Aplicado a nivel de objetos y clases.

Problematica: Para garantizar la creación de una sola instancia se requiere altos niveles de validación y una variable global; la cual es difícil de controlar cuando se presentan múltiples instanciaciones de objetos.

Solución: El patrón Singleton crea una clase que instancia el único objeto que será responsable de la creación, inicialización y acceso; dicha instancia debe ser un dato privado, ya que se debe ocultar la operación de creación de la instancia. Además se requiere una función estática pública que se encargue del encapsulamiento de la inicialización y proporcione un punto de acceso global. En el proceso de implementación del patrón se debe garantizar que cada clase es responsable de controlar que no se permita más de una instancia.

Diagrama o Implementación:

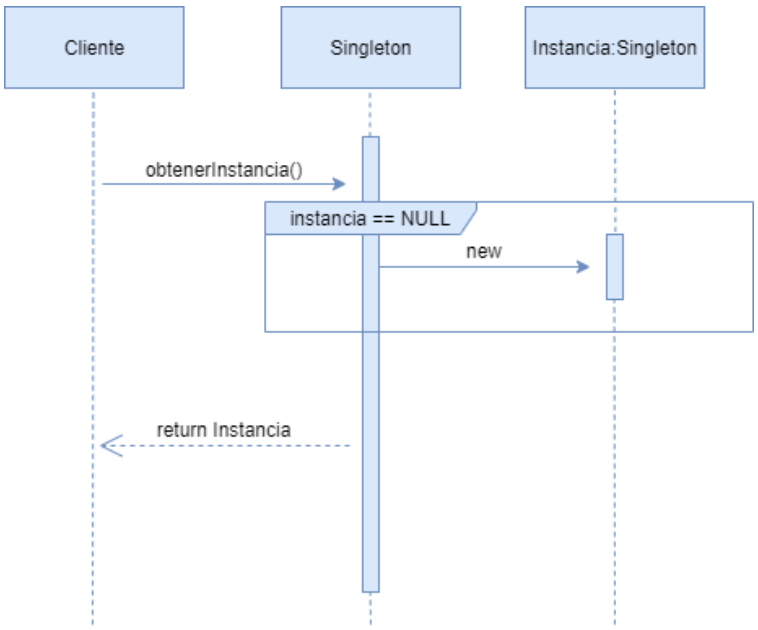

Figura 5.4: Diagrama de secuencia Singleton

A continuación se explica el diagrama de la figura 5.4

- Clase cliente solicita la instancia a la clase Singleton mediante el método estático obtenerInstancia().
- La clase Singleton validará si la instancia ya fue creada anteriormente, de no haber sido creada entonces se crea una nueva.
- Se regresa la instancia creada en el paso anterior o se regresa la instancia existente en otro caso.

Los patrones estructurales proporcionan ayuda en la combinación de clases y los objetos dando lugar a estructuras más complejas. (García Peñalvo et al. 1998)

5.4.5. Adapter

Objetivo: Permite adaptar o modificar, una interfaz existente de tal manera que a clases, que de otra forma serían incompatibles, puedan interactuar.

Aplicaciones: El uso del patrón Adapter se recomienda en los siguientes casos:

- Se requiere crear una clase reutilizable, que coopere con clases no relacionadas o no previstas; es decir, clases que no necesariamente tienen interfaces compatibles.
- Se necesita soportar diferentes conjuntos de métodos para diferentes propósitos.
- Se necesita utilizar una clase existente a través de una interfaz incompatible con dicha clase.
- Se desea permitir al usuario seleccionar entre diferentes GUI's (Interfaces Gráficas de Usuario), para explotar el mismo sistema de la forma que le resulte más cómoda al usuario.

Patrones de Diseño Colaboradores: Ningun patrón de diseño es colaborador.
Campo de acción: Aplicado a nivel de clases.

Problematica: Para hacer uso de un objeto, cuya interfaz resulta incompatible; es necesario crear una nueva o en caso contrario modificar la aplicación para que pueda integrarse dicha interfaz; en cualquiera de los dos caminos el incremento de la complejidad es notable.

Solución: El patrón de diseño Adapter permite adaptar la interfaz original de un objeto con una compatible con el objeto esperado por la clase cliente; es decir que el

patrón es visto como una capa delgada de código entre dos objetos llamados "sintácticamente incompatibles".

Diagrama o Implementación:

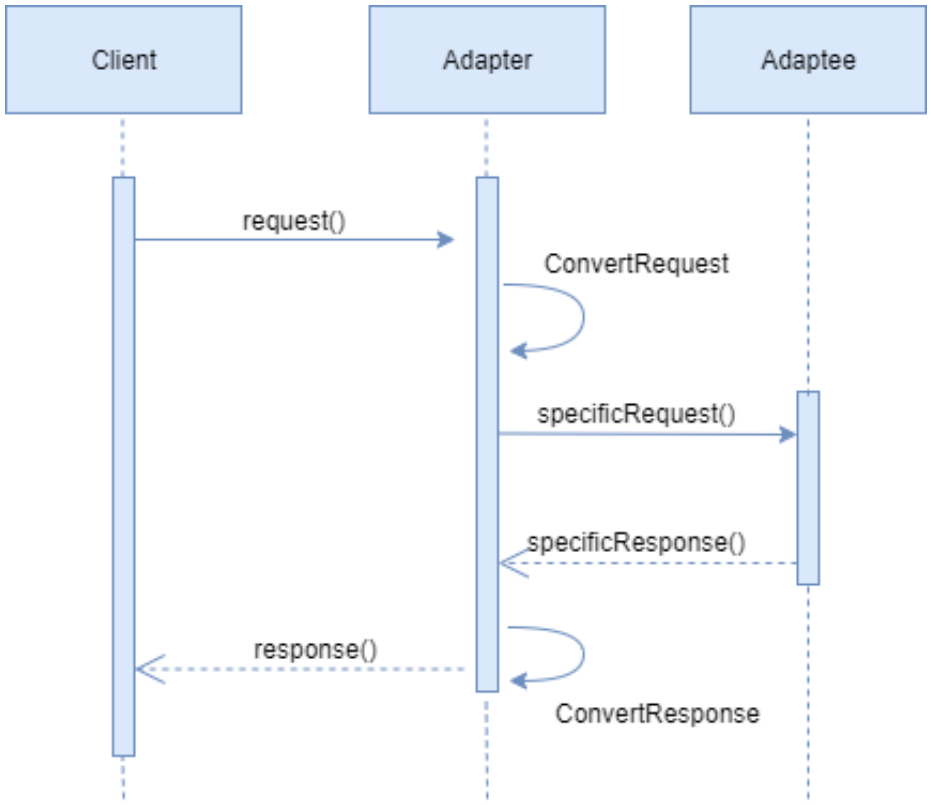

Figura 5.5: Diagrama de secuencia patrón de diseño Adapter

A continuación se explica el comportamiento del patrón Adapter mediante el diagrama de secuencia de la figura 5.5

- La clase Client invoca a la clase Adapter con parámetros genéricos.
- Clase Adapter convierte los parámetros genéricos en parámetros específicos del componente Adaptee.
- La clase Adapter invoca a Adaptee.
- Clase Adaptee responde.

- La clase Adapter convierte la respuesta de Adaptee a una respuesta genérica para la clase Client.
- La clase Adapter responde a Client con una respuesta genérica.

5.4.6. Bridge

Objetivo: Separar una abstracción de su implementación, de manera que las dos puedan variar independientemente.

Aplicaciones:

- Se desea compartir una implementación entre múltiples objetos, y que este hecho sea transparente para el cliente.
- Se necesite suprimir un enlace permanente entre una abstracción y su implementación, por ejemplo, cuando se debe enlazar o cambiar una implementación en tiempo de ejecución.
- La necesidad de implementar varias representaciones de una abstracción puede generar una proliferación de clases.
- La abstracción y la implementación deben ser extensibles mediante la agregación de subclases. En este caso el patrón de diseño bridge permite combinar las diferentes abstracciones e implementaciones y extenderlas independientemente.
- Los cambios en la implementación de una abstracción, no deben impactar a las clases clientes, es decir, su código no debe ser re-compilado.

Patrones de Diseño Colaboradores: El patrón Abstract Factory puede crear y configurar un patrón Bridge determinado.

Campo de acción: Aplicado a nivel de objetos.

Problematica: Para hacer uso de diferentes representaciones de una clase abstracta, se requiere implementar una herencia ya que una clase abstracta define la interfaz para dicha abstracción y las clases concretas, o herederas, implementan la interfaz de diferentes formas, sin embargo esta implementación es limitante, considerando que se enlaza permanentemente a la abstracción, lo cual dificulta el modificar, extender, y reutilizar las abstracciones y las implementaciones independientemente.

Solución: El patrón de diseño bridge permite conectar mediante un puente una abstracción de sus diferentes implementaciones, lo cual crea una jerarquía para las abstracciones y otra para las implementaciones, permitiendo gestionar ambas jerarquías de clases independientemente.

Diagrama o Implementación:

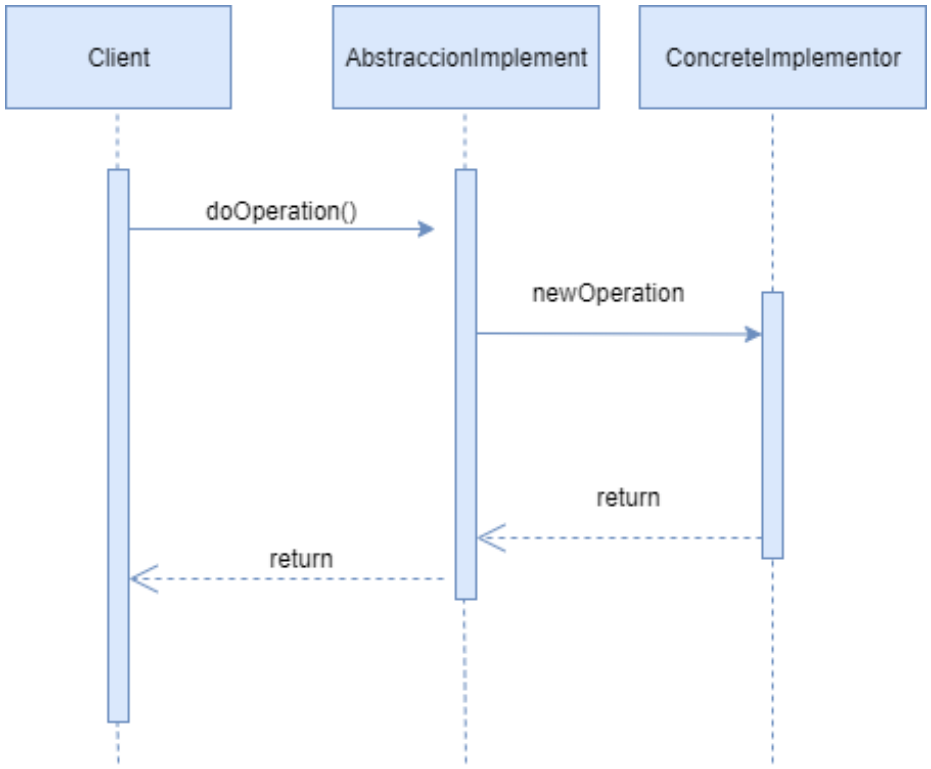

Figura 5.6: Diagrama de secuencia patrón de diseño Bridge

En la figura 5.6 se explica el comportamiento del patrón Bridge mediante un diagrama de secuencia.

- Clase client ejecuta una operación de AbstraccionImplement.
- La clase AbstraccionImplement replica la petición a ConcreteImplementor, en este paso la clase AbstraccionImplement pudiera realizar una conversión de los parámetros para ejecutar al ConcreteImplementor.
- ConcreteImplementor regresa los resultados a la clase AbstraccionImplement.
- Finalmente la clase AbstraccionImplement convierte los resultados del ConcreteImplementor para ser devueltos al cliente.

5.4.7. Composite

Objetivo: Conformar estructuras jerárquicas de manera que componentes individuales puedan ser tratados al igual que grupos de componentes. Las operaciones típicas para componentes incluyen agregar, eliminar, desplegar, encontrar y agrupar.

Aplicaciones: El uso del patrón Composite se recomienda en los siguientes casos:

- Se desea que las clases clientes manejen uniformemente a todos los objetos de una estructura jerárquica, desconociendo si están gestionando un objeto simple o uno compuesto.
- Se desea representar jerarquías completas de objetos o simplemente una parte de ellas

Patrones de Diseño Colaboradores:

- Un composite generalmente es ideal para responsabilidades Chain of Responsability.
- Un patrón decorator generalmente usa un composite.
- El patrón iterador puede ser utilizado para recorrer los composites.

Campo de acción: Aplicado a nivel de objetos.

Problematica: Para crear objetos, individuales o una agrupaciones, semejante a una estructura de árbol con nodos compuestos o simples; se debe hacer uso de listas enlazadas, sin embargo, la dificultad en estos casos es saber reconocer un objeto compuesto o uno simple, para poder determinar que operaciones aplicar a cada caso.

Solución: El patrón de diseño composite permite referenciar objetos individuales u hoja, empleando una interface "Componente" misma que ejecuta directamente la operación en caso de ser un objeto hoja, mientras que en caso de ser un objeto compuesto replica todas las operaciones a cada uno de los objetos componentes u hoja.

Diagrama o Implementación:

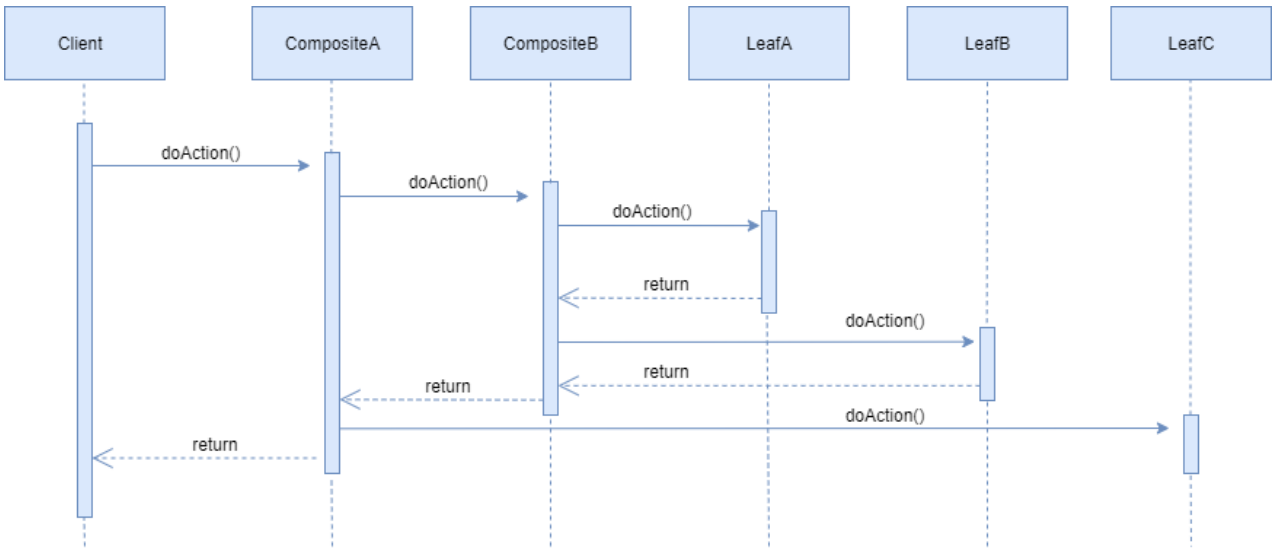

Figura 5.7: Diagrama de secuencia patrón Composite

En la figura 5.7 se explica el comportamiento del patrón Composite mediante un diagrama de secuencia.

- La clase client realizar una acción sobre la clase CompositeA.
- Clase CompositeA a su vez realiza una acción sobre clase CompositaB.

- Clase CompositeB realiza una acción sobre clase LeafA y clase LeafB y el resultado es devuelto a la clase CompositeA.
- La clase CompositeA propaga la acción sobre LeafC, el cual le regresa un resultado.
- La clase CompositeA obtiene un resultado final tras la evaluación de toda la estructura y la clase client obtiene un resultado.

5.4.8. Decorator

Objetivo: Implementar responsabilidades adicionales a un objeto, de manera dinámica, y proporcionar una alternativa flexible para extender la funcionalidad de una subclase.

Aplicaciones: Se recomienda el uso del patrón Decorator cuando:

- El sistema exige agregar responsabilidades dinámica y transparentemente a los objetos individuales; es decir sin afectar a los otros objetos.
- Se requiere retirar responsabilidades sin afectar el funcionamiento.
- La extensibilidad a subclase es impráctica. A veces un gran número de extensiones independientes es posible y produciría una explosión de subclases para dar soporte a cada combinación.

Patrones de Diseño Colaboradores:

- Un patrón Decorator es normalmente considerado un patrón Composite generado con solo un componente; ya que un Decorator añade responsabilidades adicionales.
- Los patrones Decorator y Strategy trabajan en conjunto, en el ámbito de variación de un objeto; el uno se encarga de su presentación mientras que el otro de su forma, respectivamente.

Campo de acción: Aplicado a nivel de objetos.

Problematica: Para agregar una conducta o estado específico a los objetos de manera individual y en tiempo de ejecución, se requiere implementar la herencia; sin embargo esta se aplica a toda una clase de manera estática y el cliente pierde el control.

Solución: El patrón Decorator conforma la interface del componente que trabaja como frontera entre una clase y sus respectivas subclases, misma que es transparente a los clientes, la clase decorator remite el pedido al componente y realiza acciones adicionales; mientras que la transparencia permite anidar recursivamente decoradores de tal manera que se satisfaga un número ilimitado de responsabilidades agregadas dinámicamente.

Diagrama o Implementación:

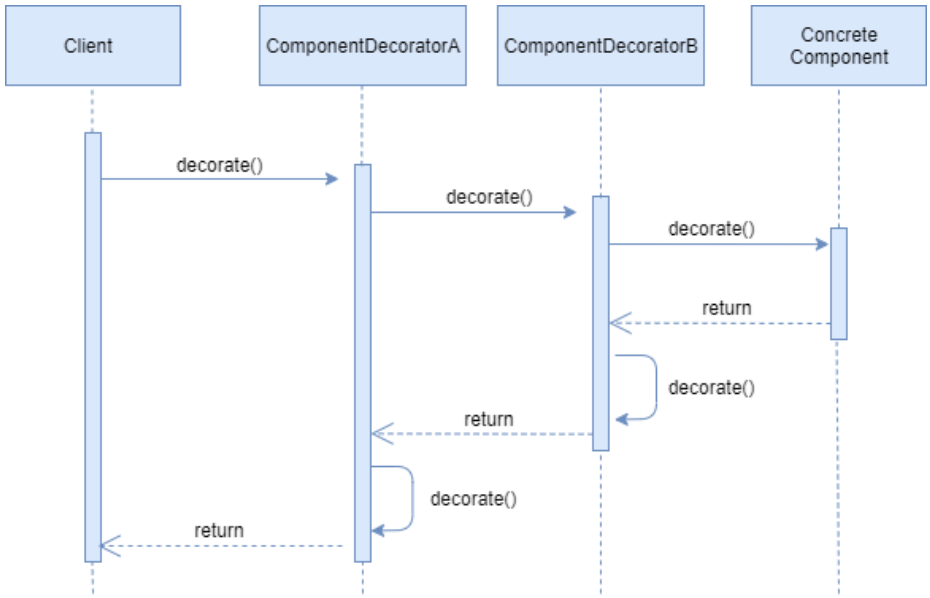

Figura 5.8: Diagrama de secuencia patrón de diseño Decorator

En la figura 5.8 se explica el comportamiento del patrón Decorator mediante un diagrama de secuencia.

- Clase Client realiza una operación sobre el componente DecoratorA.

- El componente DecoratorA realiza la misma operación sobre componente DecoradorB.
- La clase componente DecoradorB realiza una acción sobre ConcreteComponente.
- La clase componente DecoradorB ejecuta una operación de decoración.
- La clase componente DecoradorA ejecuta una operación de decoración.
- La clase Cliente recibe como resultado un objeto decorado por todos los Decoradores, los cuales encapsularon el Component en varias capas.

5.4.9. Facade

Objetivo: Proveer una interface unificada para un conjunto de interfaces de un subsistema, facilitando el uso de dicho subsistema.

Aplicaciones: Se recomienda el uso del patrón Facade cuando:

- Se presentan demasiadas dependencias entre los clientes y la implementación de clases, facilitando la portabilidad.
- El objetivo es definir un punto de entrada para cada subsistema, para simplificar la interacción entre subsistemas mediante sus respectivas implementaciones del patrón Facade
- Evitar la creciente complejidad de los subsistemas debido a su proceso de evolución, mediante la implementación de una interface simple

Patrones de Diseño Colaboradores:

- Los patrones Facade y Abstract Factory pueden trabajar en conjunto para proveer una interface para crear los objetos del subsistema de una manera independiente.

- Los patrones Facade y Mediator son similares en que abstraen la funcionalidad de clases existentes, considerando que el propósito de Mediator es resumir la comunicación entre los objetos colegas.

Campo de acción: Aplicado a nivel de objetos.

Problematica: La implementación de una solución que simplifique el uso de un subsistema complejo se la lleva a cabo mediante un proceso de encapsulamiento, mismo que en muchas ocasiones resulta no ser el mejor camino.

Solución: El patrón Facade introduce un objeto que proporciona una única interface simplificada minimizando la comunicación y dependencias entre los subsistemas. El objeto realiza una tarea compleja ya que se debe encargar de traducir la interface de cada subsistema a un lenguaje común para permitir la interacción entre ellas.

Diagrama o Implementación:

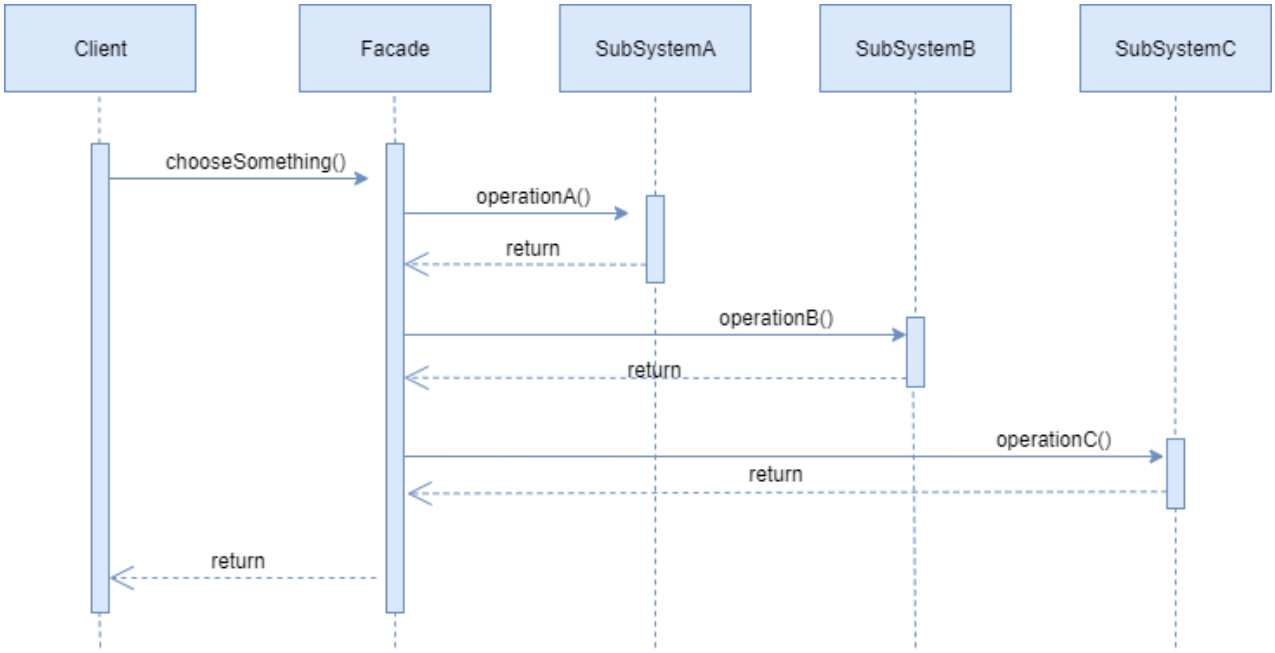

Figura 5.9: Diagrama de secuencia patrón Facade

En la figura 5.9 se explica el comportamiento del patrón Facade mediante un

diagrama de secuencia.

- La clase client invoca una operación de la clase Facade.
- La clase Facade se comunica con el componente SubsystemA para realiar una operación.
- La clase Facade se comunica con el componente SubsystemB para realiar una operación.
- La clase Facade se comunica con el componente SubsystemC para realiar una operación.
- La clase Facade responde a la clase client con el resultado de la operación.

5.4.10. Flyweight

Objetivo: Eliminar o reducir la redundancia cuando hay gran cantidad de objetos que contienen información idéntica, además lograr un equilibrio entre flexibilidad y rendimiento.

Aplicaciones: Se recomienda el uso del patrón Flyweight cuando:

- El sistema no depende de la identidad del objeto.
- El sistema requiere el uso de grandes números de objetos; lo cual produce altos costos de almacenamiento.
- La mayoría de grupos de objetos pueden ser reemplazados por pocos objetos compartidos.

Patrones de Diseño Colaboradores:

- El patrón Flyweight trabaja en conjunto con Composite, para representar una estructura jerárquica mediante un gráfico de nodos.

- Tanto los patrones State y Strategy trabajan de manera idónea con objeto Flyweight.

Campo de acción: Aplicado a nivel de objetos.

Problematica: La aplicación requiere instanciar objetos iguales, con funcionalidad diferente, lo cual es altamente costoso en términos de memoria.

Solución: El patrón Flyweight permite compartir objetos de tal manera que se emplee una granularidad fina sin los altos costos de implementación de la misma; para lo cual cada objeto es dividido en dos partes: estado-independiente o intrinsic, almacenada en el objeto Flyweight; y estado-dependiente o extrinsic, almacenada en los objetos clientes que se encargan de pasar a Flyweight sus operaciones cuando estas son invocadas.

Diagrama o Implementación:

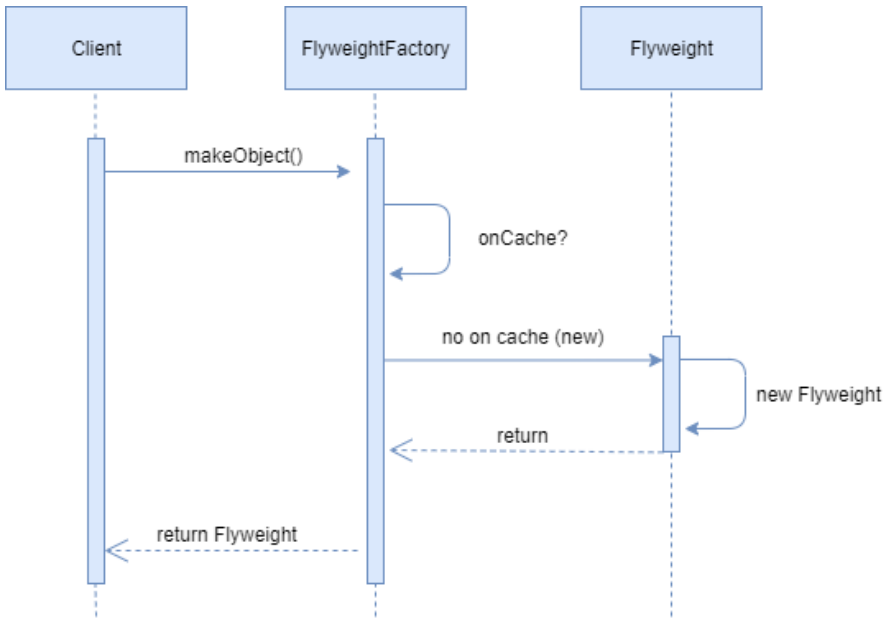

Figura 5.10: Diagrama de secuencia patron Flyweight

En la figura 5.10 se explica el comportamiento del patrón Flyweight mediante un diagrama de secuencia.

- La clase client solicita al componente Factory la creación de un objeto Flyweight.
- La clase Factory antes de crear el objeto, valida si ya existe un objeto idéntico al que se le está solicitando. De ser así, regresa el objeto existente; de no existir, crea el nuevo objeto y lo almacena en caché para ser reutilizado más adelante.
- El objeto Flyweight se crea o es tomado del caché y es devuelto al client.

5.4.11. Proxy

Objetivo: Proveer un sustituto para un objeto de tal manera que se pueda controlar el acceso a este.

Aplicaciones: Se recomienda el uso del patrón Proxy cuando:

- El sistema requiere una representación remota para un objeto en un diferente lugar.
- Se desea ocultar la complejidad de un objeto representándolo con una simple que no requiera mayor conocimiento para de esta manera facilitar su uso.
- Los objetos deben tener diferentes puntos de acceso, controlando el acceso al objeto original, sin que esto signifique instanciar el objeto en diferentes lugares.

Patrones de Diseño Colaboradores:

- Tanto el patrón Proxy como el Decorator tienen aplicaciones similares; sin embargo se debe tener cuidado ya que tienen propósitos diferentes, es decir que un Decorator agrega una o más responsabilidades a un objeto mientras que un Proxy controla el acceso al objeto.

Campo de acción: Aplicado a nivel de objetos.

Problematica: Para dar soporte a los objetos en el instante en que son requeridos, se debe instanciar previamente a todos los objetos de tal manera que estos se

encuentren listos para que el cliente haga uso de los mismos; sin embargo implica un consumo alto de recursos.

Solución: El patrón Proxy permite controlar el acceso a un objeto instanciándolo en el momento en el que el objeto sea usado realmente; para lo cual emplea un objeto imagen "proxy" que siempre está presente como representación del original, y se encarga de instanciarlo solo cuando es requerido; de tal manera que se pone a disposición una referencia más versátil y sofisticada que un simple punto de acceso a un objeto; mediante tres opciones: un "proxy remoto" ocultando el hecho de que un objeto reside en un espacio diferente o "proxy virtual" optimizando la creación de un objeto en el momento que es requerido o de ambas maneras empleando referencias inteligentes realizando tareas adicionales cuando el objeto es accedido.

Diagrama o Implementación:

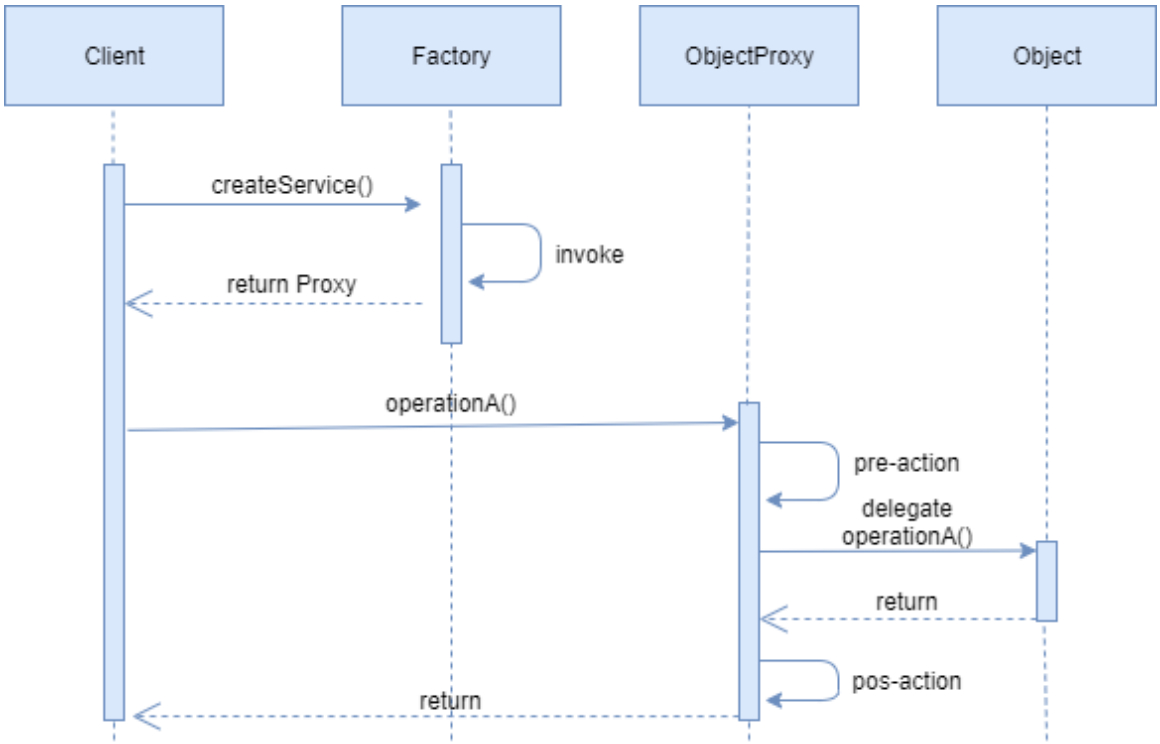

Figura 5.11: Diagrama de secuencia patrón Proxy

En la figura 5.11 se explica el comportamiento del patrón Proxy mediante un diagrama de secuencia.

- La clase cliente solicita al componente Factory un Objeto.
- Componente Factory crea un Proxy que encapsule al Object.
- Clase cliente ejecuta el Proxy creado por el componente Factory.
- La clase Proxy realiza una o varias acciones previas a la ejecución del Object.
- La clase Proxy delega la ejecución al componente Object.
- La clase Proxy realiza una o varias acciones después de la ejecución del Object.
- La clase Proxy regresa un resultado.

Estos patrones de diseño están relacionados con algoritmos y asignación de responsabilidades a los objetos.

5.4.12. Chain of responsibility

Objetivo: Evitar acoplar a la clase solicitante de una acción con la clase ejecutor de la misma para permitir a más de un objeto manejar la solicitud. Encadenar los objetos receptores y pasar la demanda a lo largo de toda la cadena hasta que un objeto la maneje.

Aplicaciones: Se recomienda el uso del patrón Chain of responsability cuando:

- Varios objetos deben manejar una solicitud, y determinar quien lo haga no es prioritario pues el objeto que la maneje es determinado automáticamente.
- Se desea realizar una solicitud a uno de entre varios objetos sin especificar explícitamente quien debe recibir la solicitud.
- El conjunto de objetos capaces de manejar una solicitud deben definirse dinámicamente.

Patrones de Diseño Colaboradores:

- El patrón chain of responsability generalmente es implementado junto con el patrón composite, pues un componente padre puede ser un sucesor.

Campo de acción: Aplicado a nivel de objetos.

Problematica: Una aplicación necesita ejecutar una acción independientemente del estado de la misma, convencionalmente es posible cubrir este requerimiento, pero para lograrlo, cada objeto debe instanciar el método respectivo que se encarga de ejecutar la solicitud y conocer además los objetos capaces de ejecutar la misma acción e invocarlos en un orden determinado, lo cual genera un alto acoplamiento entre objetos.

Solución: El patrón de diseño Chain of Responsability presenta una solución más óptima para cubrir este requerimiento, pues propone preparar cada objeto para que sea capaz de manejar una solicitud por su parte o a su vez de pasarle la solicitud a

otro objeto definido dinámicamente en tiempo de ejecución, lo cual genera una cadena de objetos que son potenciales ejecutores de una solicitud, en caso de no poder ejecutar la solicitud, el objeto la pasa al siguiente eslabón de la cadena.

Diagrama o Implementación:

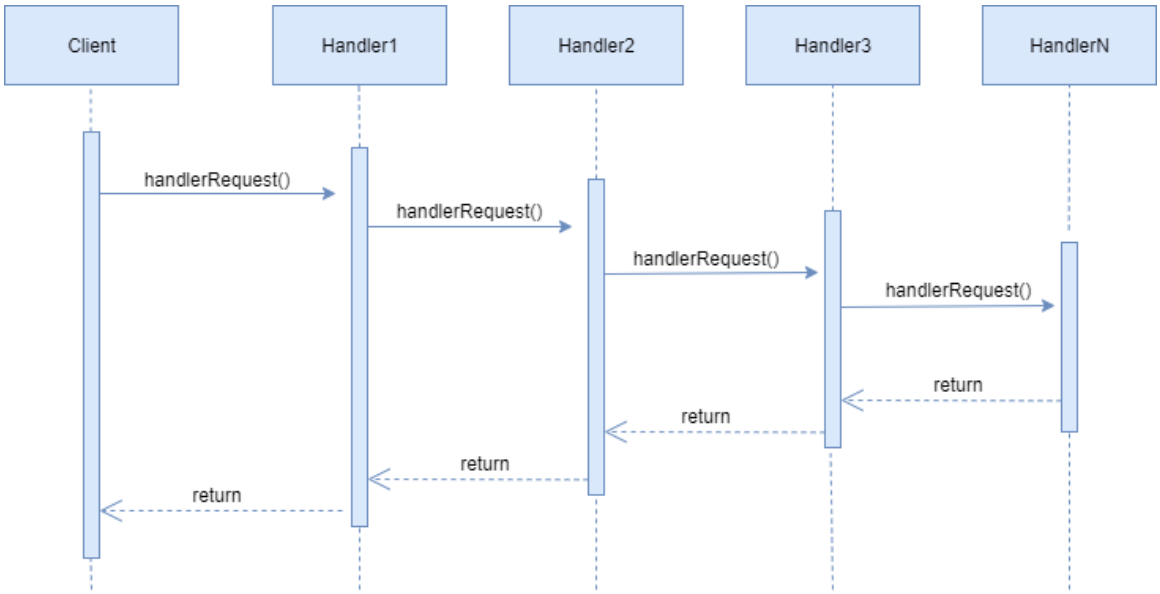

Figura 5.12: Diagrama de secuencia patrón Chain Of Responsability

En la figura 5.12 se explica el comportamiento del patrón Chain Of Responsability mediante un diagrama de secuencia.

- La clase client solicita el procesamiento de una solicitud a una cadena de responsabilidad (Chain Of Responsability).
- El primer componente Handler intenta procesar el mensaje, sin embargo, no es capaz de procesarlo por alguna razón y envía el mensaje al siguiente componente handler de la cadena.
- El segundo componente Handler intenta procesar el mensaje sin éxito, por lo que envía el mensaje al siguiente componente Handler de la cadena.
- El tercer componente Handler también intenta procesar el mensaje sin éxito y envía el mensaje al siguiente Handler de la cadena.

- El componente HandlerN (Algun handler de la secuencia) por fin es capaz de procesar el mensaje exitosamente y regresa una respuesta (opcional) de tal forma que la respuesta es replicada por todos los componentes Handlers pasados hasta llegar a la clase client.

5.4.13. Command

Objetivo: Encapsular una solicitud en un objeto facilitando la parametrización de las clases clientes con diferentes consultas.

Aplicaciones:

Se recomienda el uso del patrón Command cuando:

- Se tienen comandos que diversos receptores pueden manejar de diferentes maneras.
- Se dispone de un conjunto de comandos de alto nivel que son implementados mediante operaciones primitivas.
- Se desea especificar, apilar y ejecutar comandos a diversos instantes.
- Es necesario disponer de la opción de deshacer la ejecución de comandos.
- Es necesario manejar auditorias y registros de todos los cambios mediante comandos.

Patrones de Diseño Colaboradores:

- Un patrón composite puede ser usado para implementar comandos compuestos o transacciones, el uso del patrón memento conserva el estado que un comando requiere para poder deshacer los efectos de su ejecución. El uso del patrón prototype permite copiar un comando antes de ser grabado en un registro o histórico de acciones.

Campo de acción: Aplicado a nivel de objetos.

Problematica: Para la interacción entre objetos se manejan referencias directas entre los mismos, un objeto solicitante referencia los métodos de uno receptor; por medio de sentencias condicionantes (if o switch, case), se incrementa la complejidad cuando se trata de agregar una nueva clase receptora; ya que se debe modificar la clase invocadora aumentando la sentencia condicionante que discrimina la nueva clase; lo cual contradice los dos principios de la programación orientada a objetos.

Solución: El patrón de diseño command soluciona esta contradicción mediante la creación de una clase abstracta llamada comando, la cual puede ser instanciada en una clase concreta, permitiendo parametrizar un objeto comando a ejecutar para la clase cliente.

Diagrama o Implementación:

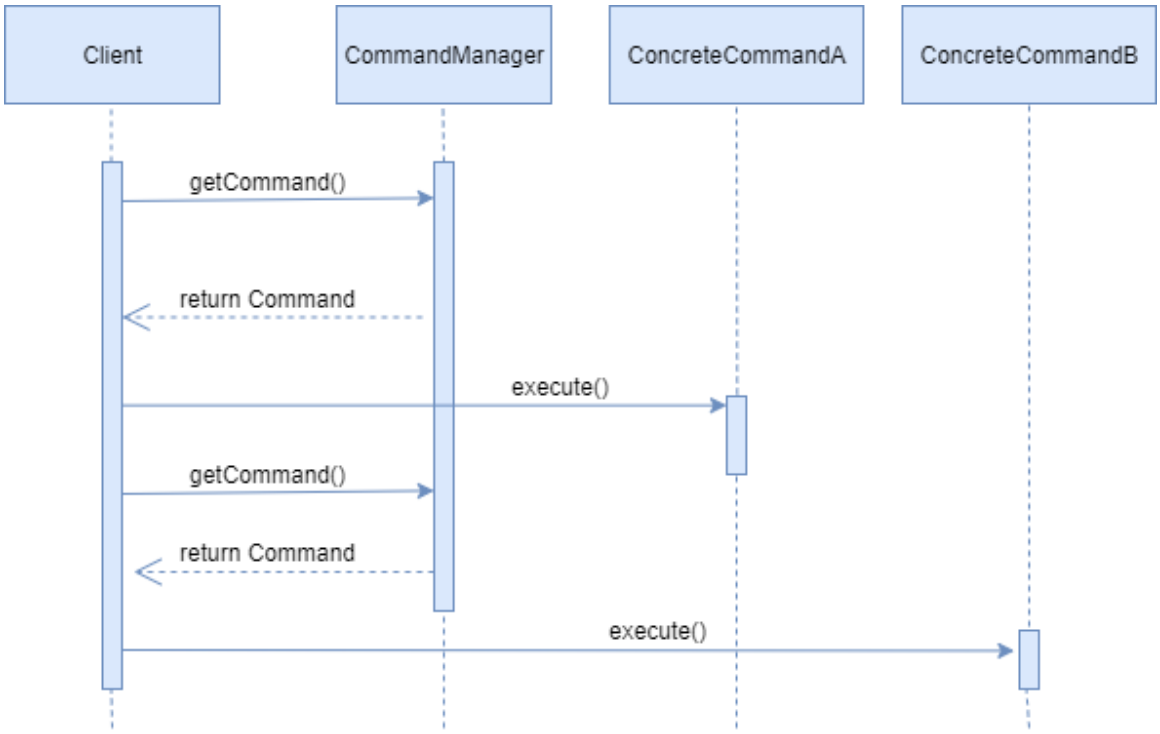

Figura 5.13: Diagrama de secuencia patrón Command

En la figura 5.13 se explica el comportamiento del patrón Command mediante un

diagrama de secuencia.

- El componente invoker obtiene un Comando de la clase CommandManager.
- El componente invoker ejecuta el comando.
- El componente invoker obtiene otro Comando de la clase CommandManager.
- El componente invoker ejecuta el comando.

5.4.14. Iterator

Objetivo: Proveer una forma de acceder secuencialmente a los elementos de un objeto agregado, sin exponer su representación subyacente.

Aplicaciones:

Se recomienda el uso del patrón Iterator cuando:

- Se requiere manejar una colección de objetos y existen diferentes formas de navegar a través de ella.
- Existen diferentes colecciones de objetos para la misma lógica de navegación.
- Se pueden aplicar diferentes filtros y algoritmos de ordenamiento.

Patrones de Diseño Colaboradores:

- Los patrones Iterator generalmente se aplican a estructuras recursivas, por lo que el uso del patrón composite es muy común.
- El patrón factory method permite instanciar el iterador apropiado para una colección.
- Generalmente se utiliza el patrón memento para conservar internamente el estado de cada iteración.

Campo de acción: Aplicado a nivel de objetos.

Problematica: Para manejar diversos objetos contenidos en una colección se debe la debe instanciar con su algoritmo de navegación; lo cual implica la creación de varios objetos, garantizando que la clase cliente conozca las implementaciones de los mismos, sobrecargando la memoria y provocando un alto acoplamiento entre los objetos y las clases que los instancian.

Solución: El patrón de diseño Iterator implementa una interfaz para navegar a través de la colección de objetos, permitiendo que cada uno discrimine la secuencia a seguir; es decir que basta con instanciar la interfaz y cada objeto la implementará indicando el objeto actual y el siguiente, de acuerdo a algún criterio de ordenamiento de los elementos de la colección.

Diagrama o Implementación:

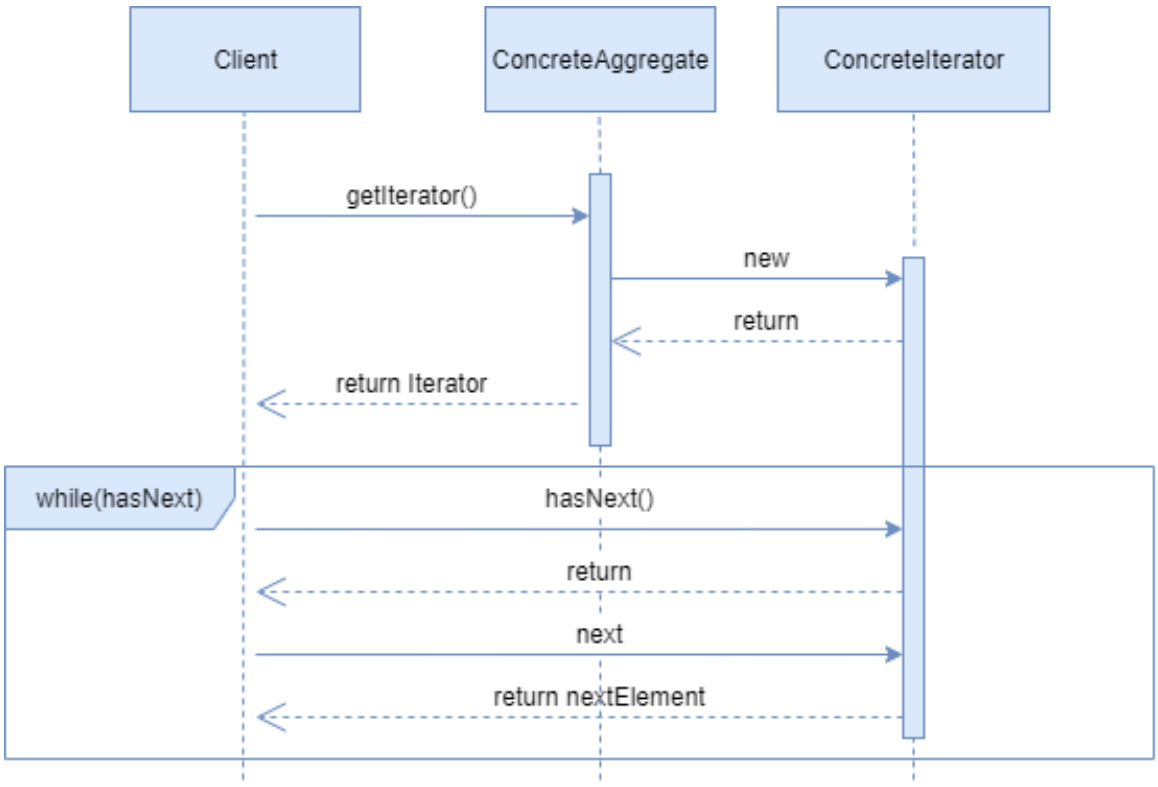

Figura 5.14: Diagrama de secuencia patrón Iterator

En la figura 5.14 se explica el comportamiento del patrón Iterator mediante un diagrama de secuencia.

- La clase client solicita al componente ConcreteAggregate la creación de un iterador.
- El componente ConcreteAggregate crea un nuevo Iterador.
- La clase client, para recorrer los elementos, entra en un ciclo hasta que no existen más elementos en el iterador, el método hasNext le indicará cuándo se ha llegado al final.
- La clase client solicita el siguiente elemento al iterador mediante el método next.
- Si existen más elementos nos regresamos al paso tres, esto se repite hasta finalizar el recorrido.

5.4.15. Interpreter

Objetivo: Dado un lenguaje de programación, define una representación con un intérprete que es usado para las sentencias.

Aplicaciones:

Se recomienda el uso del patrón Interpreter cuando:

- Se desea interpretar una gramática, y esta no es muy compleja.
- La eficiencia esperada de la aplicación no es exigente.

Patrones de Diseño Colaboradores:

- Las sentencias con estructura de árbol generalmente se implementan como un composite.
- El patrón flyweight muestra como compartir símbolos terminales dentro de la sintaxis de árbol abstracta.
- Se puede utilizar un patrón Iterator para recorrer la estructura.
- El patrón visitor puede ser utilizado para mantener el comportamiento de cada nodo en la estructura de árbol.

Campo de acción: Aplicado a nivel de clases.

Problematica: Para ejecutar diversos comandos o instrucciones, estas son definidas en una estructura básica para que la aplicación las interprete de manera correcta; mediante un objeto que interpreta las sentencias por medio de condicionantes if o Switch, case: lo cual complica considerablemente la programación.

Solución: El patrón de diseño Interpreter define a las instrucciones como objetos, lo cual permite que cada uno de estos que se implemente se identifique a sí mismo como una sentencia o a su vez a otra mediante un análisis de estructuras de árbol; todo este proceso mediante una interfaz.

Diagrama o Implementación:

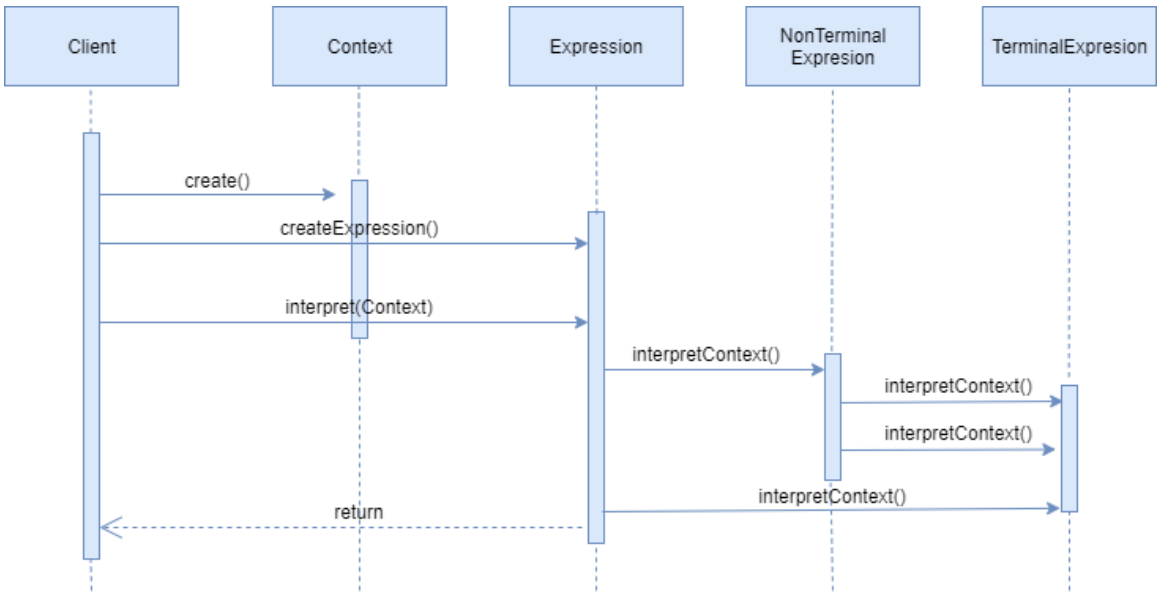

Figura 5.15: Diagrama de secuencia patrón interpreter

En la figura 5.15 se explica el comportamiento del patrón interpreter mediante un diagrama de secuencia.

- La clase client crea el contexto para la ejecución del componente interpreter.
- La clase client crea u obtiene la expresión a evaluar.
- La clase client solicita la interpretación de la expresión al componente interpreter y le envía el contexto.
- La Expresión manda llamar a las Expresiones No Terminales que contiene.
- La Expresión No Terminal manda llamar a todas las Expresiones Terminales.
- La Expresión Raíz solicita la interpretación de una Expresión Terminal.
- La expresión se evalúa por completo y se tiene un resultado de la interpretación de todas las expresiones terminales y no terminales.

5.4.16. Mediator

Objetivo: Definir un objeto que encapsule la interacción entre otros, promoviendo un bajo acoplamiento; ya que evita que los objetos se referencien explícitamente entre ellos variando independientemente de su interacción.

Aplicaciones: Se recomienda el uso del patrón Mediator cuando:

- Los objetos de la aplicación se comunican de manera bien estructura pero potencialmente complejas.
- Las identidades de los objetos deben protegerse aun cuando estos se comuniquen entre sí
- El comportamiento de algunos objetos puede ser agrupado y personalizado.
- La reutilización de un objeto es complicada porque este referencia y se comunica con muchos otros objetos.

Patrones de Diseño Colaboradores:

- Las clases Mediator frecuentemente utilizan el patrón de diseño Observer para recibir notificaciones de las diferentes solicitudes de las clases que interactúan.
- puede utilizar el patrón Adapter para independizar la clase Mediator de las clases concretas que gestiona.

Campo de acción: Aplicado a nivel de objetos.

Problematica: Todas las aplicaciones orientadas a objetos basan su funcionamiento en la interacción entre estos, si no existe un mecanismo claramente definido de interacción se puede generar una aplicación con un alto acoplamiento contradiciendo los principios de la programación orientada a objetos.

Solución: El patrón de diseño Mediator, coordina interacciones entre objetos relacionados; centralizando en una clase la lógica que realiza los cambios de estados de los objetos, de manera que ofrece una forma sistematizada de aumentar la cohesión (se centraliza la lógica) y reducir el acoplamiento entre clases (se reducen la dependencias entre ellas).

Diagrama o Implementación:

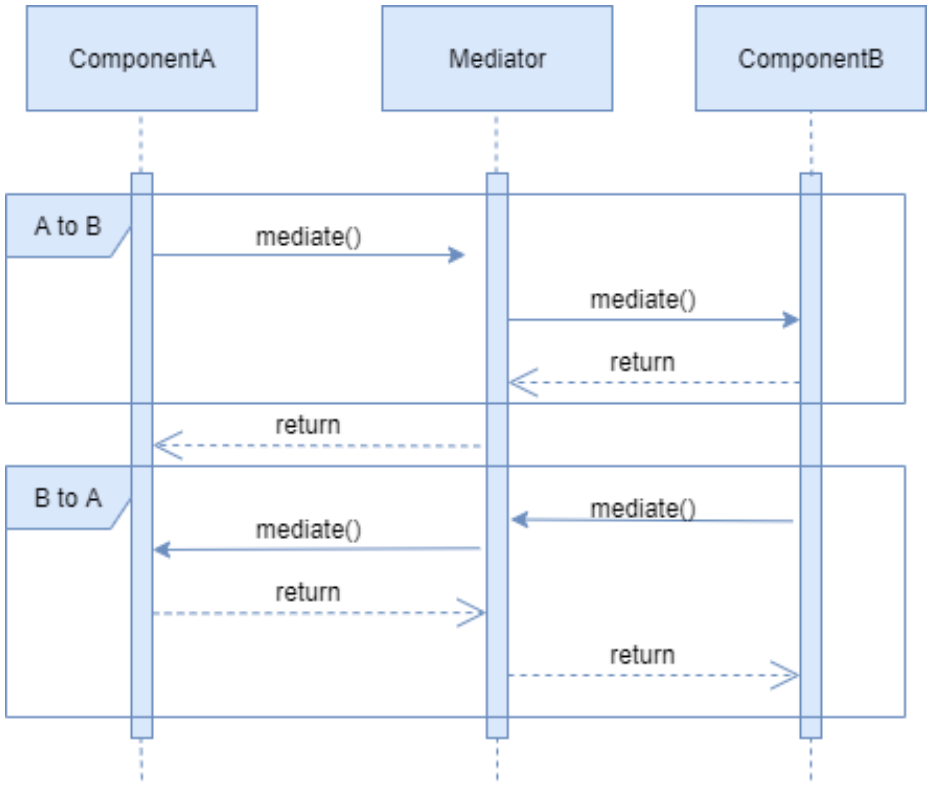

Figura 5.16: Diagrama de secuencia patrón Mediator

En la figura 5.16 se explica el comportamiento del patrón Mediator mediante un diagrama de secuencia.

- La clase ComponenteA desea comunicarse con el ComponenteB y le envía un mensaje por medio del mediador.
- El componete mediator puede analizar el mensaje con fines de depuración, seguimiento o para canalizar el mensaje al destinatario.
- El mensaje es entregado al destinatario y regresa una respuesta al componente

mediator.

- El mediator recibe la respuesta y la reedirecciona al ComponenteA.
- De igual forma, el proceso se puede repetir del ComponentB al ComponentA repitiendo los pasos anteriores logrando una comunicación bidireccional.

5.4.17. Memento

Objetivo: Almacenar el estado interno de un objeto en un entorno externo al mismo, de manera que dicho estado pueda ser restaurado posteriormente.

Aplicaciones: Se recomienda el uso del patrón Memento cuando:

- Es necesario conservar el estado de un objeto a lo largo del tiempo (persistencia), y al mismo tiempo encapsular los valores de los atributos de dicho estado del objeto a objetos externos.

Campo de acción: Aplicado a nivel de objetos.

Problematica: Una aplicación desea implementar la funcionalidad de deshacer acciones realizadas sobre objetos, lo que implica guardar sus estados; lo cual requiere que se implemente la funcionalidad de guardar y recuperar el estado en el objeto mismo, saturando la memoria.

Solución: El patrón de diseño Memento sugiere separar al objeto de las funciones de guardar y recuperar un estado determinado, pero conservando al mismo tiempo el encapsulamiento de los valores de los atributos del objeto, mediante la restricción de acceso de otros objetos a la persistencia de los estados del objeto.

Diagrama o Implementación:

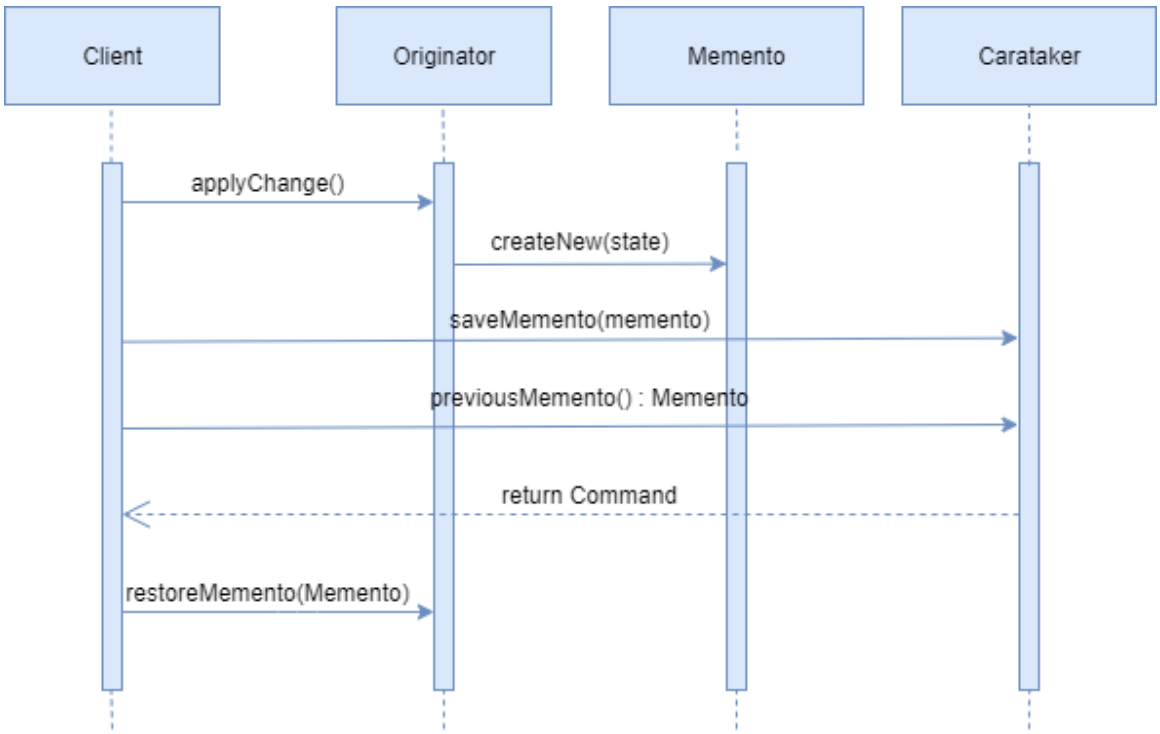

Figura 5.17: Diagrama de secuencia patrón Memento

En la figura 5.17 se explica el comportamiento del patrón Memento mediante un diagrama de secuencia.

- La clase client aplica un cambio sobre el componente Originator.
- El componente Originator crea un nuevo Memento que representa su estado actual.
- La clase client guardar el Memento en el componente Caretaker para posteriormente poder cambiar entre los estados del componente Originator.
- Después de un tiempo, La clase client solicita al componente Caretaker el estado previo del componente Originator.
- La clase client restaura el estado del Originator mediante el Memento obtenido del componente Caretaker .

5.4.18. Observer

Objetivo: Definir una o más dependencias entre objetos de tal manera que si un objeto cambia su estado, dichas relaciones sean notificadas y actualizadas automáticamente.

Aplicaciones: Se recomienda el uso del patrón Observer cuando:

- Una abstracción requiere dos aspectos, uno dependiente del otro; los cuales necesitan ser encapsulados en los objetos separados para que pueden variar y ser usados de manera independiente.
- El cambio de un objeto requiere el cambio de otros, y no se conoce con exactitud cuántos.
- Un objeto debe notificar a otros sin necesidad de conocer quiénes son estos de manera específica.

Patrones de Diseño Colaboradores:

- El patrón Mediator puede actuar como mediador entre los sujetos y los observadores, encapsulando la semántica de actualización compleja.
- El patrón Observer puede hacer uso del patrón Singleton para hacer único y globalmente accesible al sujeto.

Campo de acción: Aplicado a nivel de objetos.

Problematica: La aplicación requiere que se cree dependencias entre objetos que permitan actualizar las relaciones entre los mismos en el momento en que uno varíe, en el desarrollo convencional este proceso implica código redundante y consumo amplio de recursos.

Solución: El patrón Observer describe cómo establecer estas relaciones; empleando

dos objetos esenciales: “sujeto” y “observador” que realizan el proceso conocido como publicar-subscribir; es decir sujeto puede manejar cualquier número de observadores dependientes, los que son notificados mediante una “publicación” cuando un sujeto sufre algún tipo de cambio en su estado, en ese momento observador le consultará para sincronizar su estado con el de el sujeto.

Diagrama o Implementación:

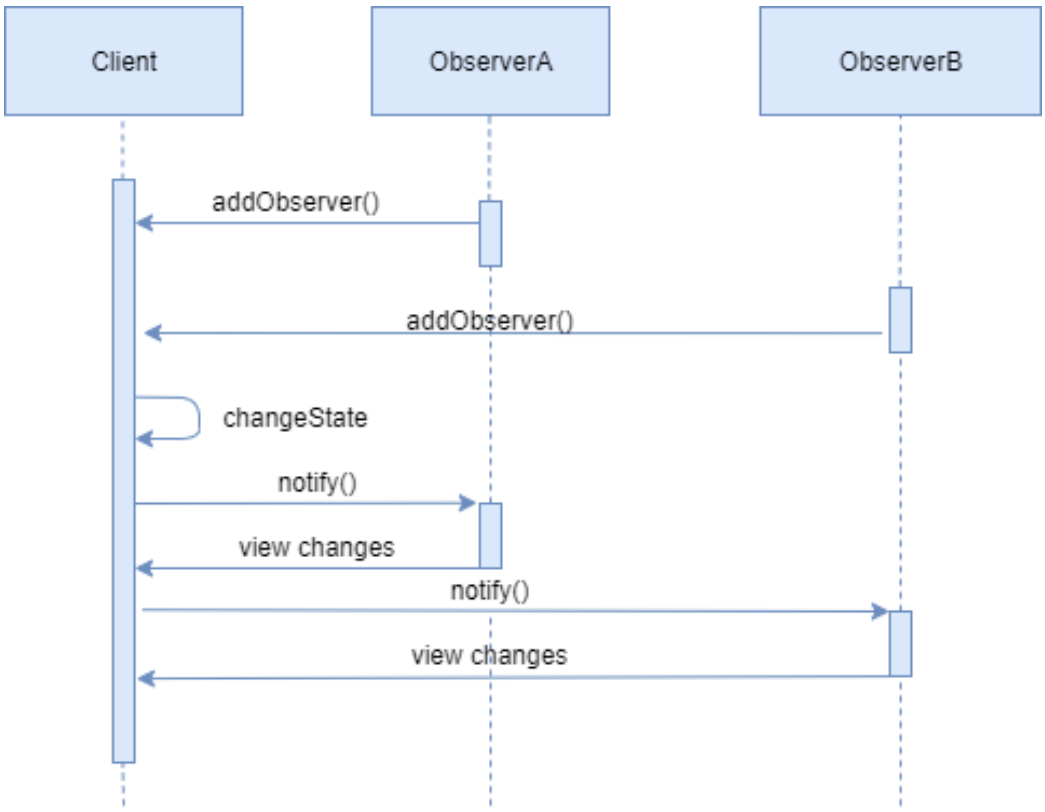

Figura 5.18: Diagrama de secuencia patrón Observer

En la figura 5.18 se explica el comportamiento del patrón Observer mediante un diagrama de secuencia.

- El componente ObserverA se registra con el objeto client para ser notificado de algún cambio.
- El componente ObserverB se registra con el objeto client para ser notificado de algún cambio.
- Ocurre algún cambio en el estado del client.
- Todos los Observers son notificados con el cambio ocurrido.

5.4.19. State

Objetivo: Permitir a un objeto alterar su conducta cuando sus estados internos cambian; aparentemente parecerá que cambia sus clases.

Aplicaciones:

Se recomienda el uso del patrón State cuando:

- La conducta de un objeto depende totalmente de su estado, la cual debe cambiar en tiempo de ejecución de acuerdo a dicho estado.
- Las operaciones a ser realizadas tienen grandes declaraciones con muchas partes condicionantes que dependen del estado del objeto; el cual normalmente es representado por una o varias constantes numeradas

Patrones de Diseño Colaboradores:

- El patrón Flyweight colabora en la implementación del patrón State; ya que explica cuando y donde deben ser compartidos los objetos States.

Campo de acción: Aplicado a nivel de objetos.

Problematica: La aplicación requiere que se controle en tiempo de ejecución la variación de un objeto monolítico; dado que este es definido en función de su estado, si este varía el objeto también debe hacerlo; lo que implica crear una clase de declaraciones dentro de la cual se determina que conducta se debe llevar a cabo, empleando un proceso poco práctico.

Solución: El patrón State permite tener una clase que varía considerando las numerosas clases relacionadas; es decir que el patrón cambia entre las diferentes clases internas de tal manera que el objeto adjuntado parece cambiar de clase. Cuando se presentan declaraciones condicionantes el patrón coloca cada rama del condicionante en una clase separada tratando el estado del objeto como variable independiente de otros.

Diagrama o Implementación:

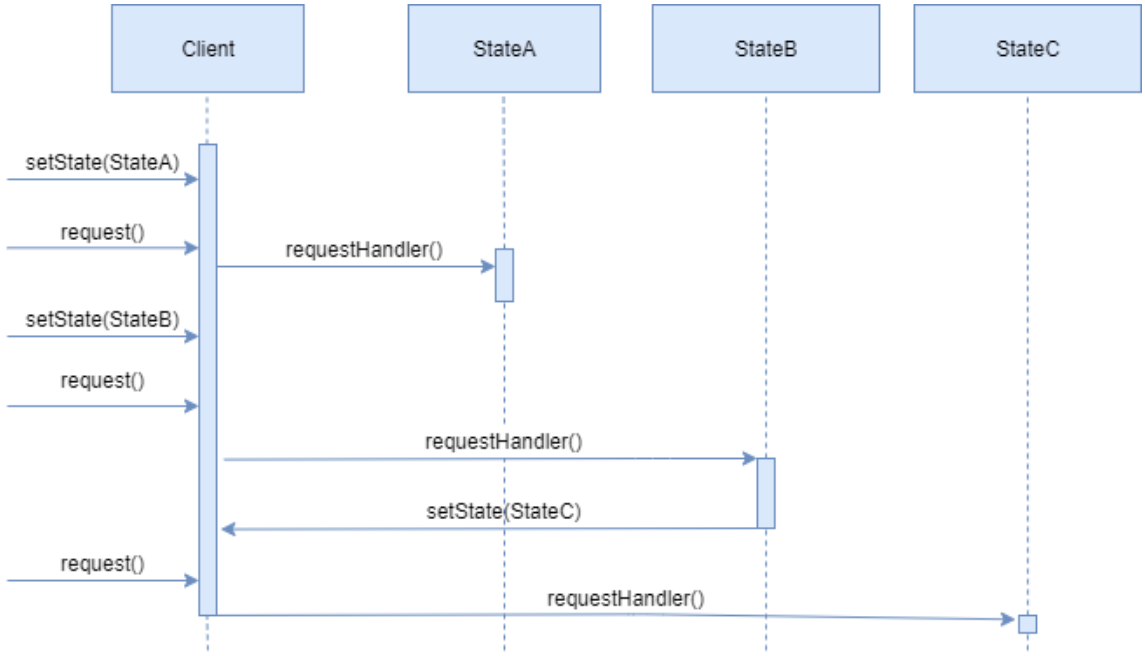

Figura 5.19: Diagrama de secuencia patrón State

En la figura 5.19 se explica el comportamiento del patrón State mediante un diagrama de secuencia.

- Se establece un estado por default la clase Client, el cual es StateA.
- Se ejecuta la operación request sobre la clase Client, la cual delega la ejecución al estado actual (StateA).
- la clase Client cambia del estado A al estado B.
- Se ejecuta nuevamente la operación request sobre la clase Client que delega la ejecución al estado actual (StateB).
- La ejecución del StateB da como resultado un cambio de estado al StateC.
- Se ejecuta nuevamente la operación request sobre la clase Client que delega la ejecución al estado actual (StateC).

5.4.20. Strategy

Objetivo: Definir un grupo de algoritmos, encapsulando cada uno y haciéndolo intercambiable; esto permite que el algoritmo varíe independientemente de las clases clientes que lo usen.

Aplicaciones: Se recomienda el uso del patrón Strategy cuando:

- Varias clases relacionadas únicamente difieren en su conducta; ya que las estrategias proporcionan una manera de configurar una clase con una de muchas conductas.
- La aplicación necesita diferentes variantes del algoritmo; es decir que se podría definir un algoritmo que refleje los intercambios de una variable.
- Se requiere que un algoritmo use datos que los clientes no deben conocer; ya que el patrón evita exponer las estructuras complejas de los datos.
- Una clase debe definir muchas conductas; las cuales aparecen como múltiples declaraciones condicionantes en sus funcionamientos y el patrón se encarga de relacionarlas en ramas en su propia clase.

Patrones de Diseño Colaboradores:

El patrón Strategy eventualmente puede desarrollar adecuados objetos Flyweight.

Campo de acción: Aplicado a nivel de objetos.

Problematica: La aplicación requiere que se emplee uno de los principales conceptos de diseño "abrir-cerrar"; para alcanzar este objetivo se encapsula los detalles de las interfaces; pero esto no cubre los aspectos de cambios de clases y el impacto de la implementación de clases derivadas.

Solución: El patrón Strategy crea un conjunto de algoritmos encapsulados y relacionados entre sí en una clase conductora llamada "Context"; el programa cliente

puede seleccionar uno de los algoritmos o en algunos casos Context se encarga de seleccionar el más idóneo de acuerdo a la situación. El punto clave es que permite cambiar fácilmente entre algoritmos.

Diagrama o Implementación:

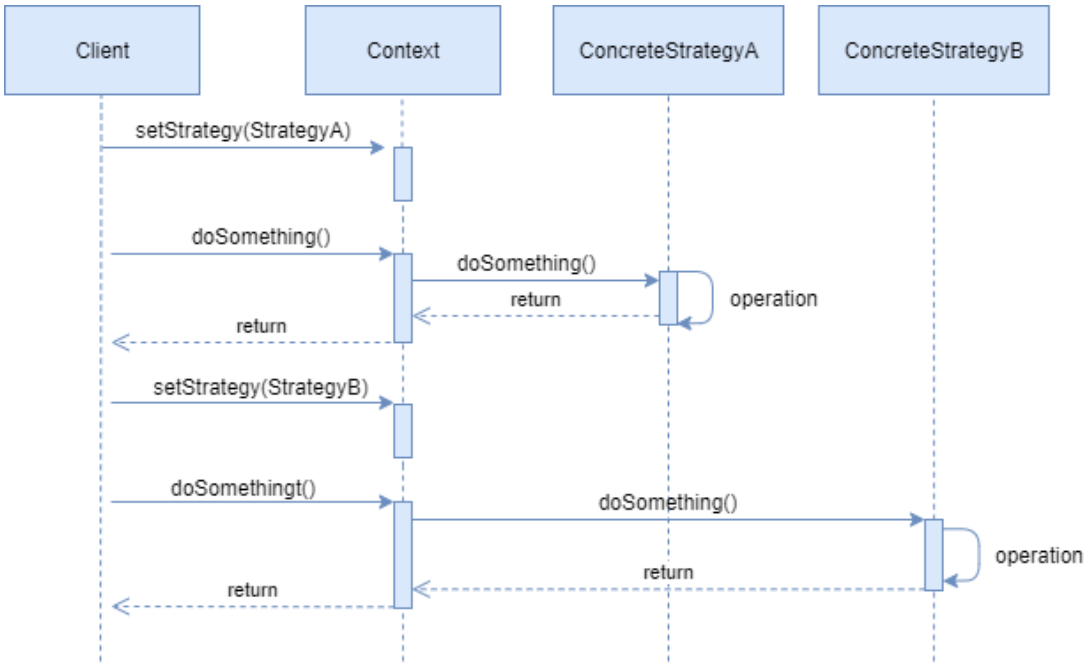

Figura 5.20: Diagrama de secuencia patrón Strategy

En la figura 5.20 se explica el comportamiento del patrón Strategy mediante un diagrama de secuencia.

- La clase client crea un nuevo componente contexto y establece la estrategia A.
- La clase client ejecuta la operación doSomething.
- El componente Context a su vez delega esta responsabilidad a ConcreteStrategyA.
- ConcreteStrategyA realiza la operación y regresa el resultado.
- El componente Context toma el resultado y lo regresa la clase client.
- La clase client le cambia la estrategia al componente Context en tiempo de ejecución.

- La clase client ejecuta nuevamente la operación doSomething.
- El componente Context a su vez delega esta responsabilidad a ConcreteStrategyB.
- ConcreteStategyB realiza la operación y regresa el resultado.
- El componente Context toma el resultado y lo regresa la clase client.

5.4.21. Template method

Objetivo: Definir la estructura de un algoritmo en una operación permitiendo a las subclases redefinir ciertos pasos sin cambiar la estructura del mismo.

Aplicaciones: Se recomienda el uso del patrón Template cuando:

- Se requiere implementar la parte invariable de un algoritmo en la clase y delegar las partes variables a las subclases.
- Una conducta común entre subclases debe factorizarse y localizarse en una clase común evitando código duplicado.
- La aplicación necesita controlar las extensiones de las subclases.

Patrones de Diseño Colaboradores:

- El patrón Template presenta un funcionamiento semejante al patrón Strategy; usando la herencia para variar la parte de un algoritmo.

Campo de acción: Aplicado a nivel de clases.

Problematica: Es necesario emplear un mayor esfuerzo para poder proporcionar una interface común a componentes que han llegado a tener similitudes significantes; ya que implicaría duplicar el esfuerzo para poder detallar al máximo la relación entre dichos componentes.

Solución: El patrón Template escribe una clase padre en la que se asignan uno o más métodos a ser implementados a las clases derivadas, formalizando la idea de definir un algoritmo en una clase; pero delegando los detalles de la implementación a las subclases.

Diagrama o Implementación:

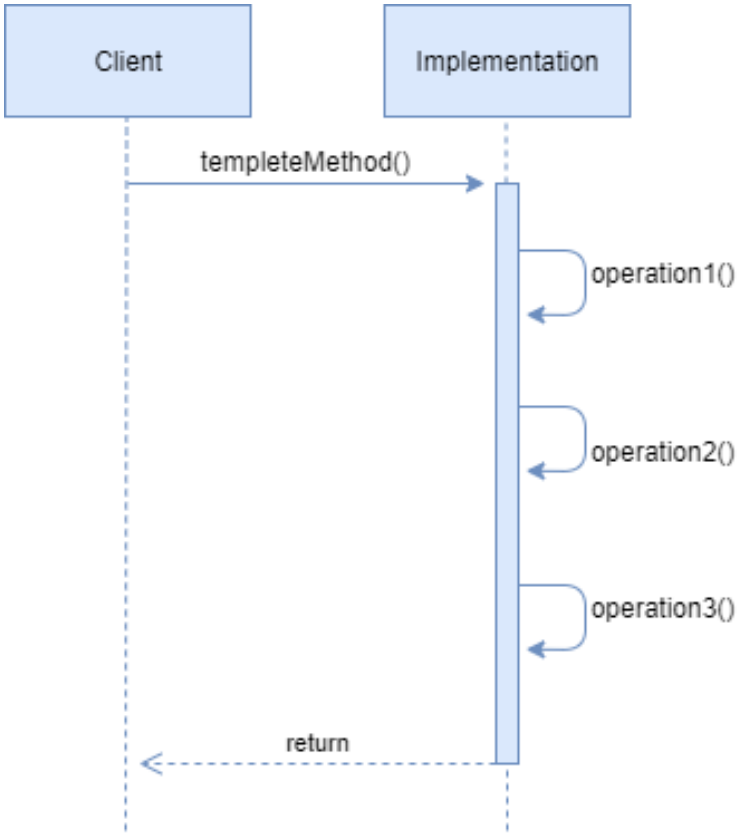

Figura 5.21: Diagrama de secuencia patrón Template Method

En la figura 5.21 se explica el comportamiento del patrón Template Method mediante un diagrama de secuencia.

- La clase cliente crea u obtiene una instancia de una implementación del templete.
- La clase cliente ejecuta el método público templete method del templete.
- La clase implementation por default del método templete method ejecuta en orden los métodos step1, step2, step3.

- La clase implementation retorna un resultado.

5.4.22. Visitor

Objetivo: Representar una operación a ser realizada en los elementos de una estructura de objetos; permitiendo definir un nuevo funcionamiento sin cambiar las clases de los elementos que operan.

Aplicaciones: Se recomienda el uso del patrón Visitor cuando:

- Una estructura de objetos contiene muchas clases de objetos que difieren en su interface y el objetivo es combinar su funcionamiento de sus clases concretas.
- Muchas operaciones distintas y no relacionadas necesitan ser ejecutadas en objetos y estructuras de objetos; el objeto Visitor permite mantener unido el funcionamiento relacionado definiéndolo en una clase.
- Las clases definen estructuras de objetos que rara vez cambian; pero se necesita definir requerimientos altamente cambiantes.

Patrones de Diseño Colaboradores:

- El patrón Visitor puede ser usado para aplicar una operación sobre una estructura de objetos definida por un patrón Composite.
- Los objetos Visitor pueden ser aplicados para la interpretación de objetos Interpreter.

Campo de acción: Aplicado a nivel de objetos.

Problematica: La aplicación requiere que se realicen muchas operaciones distintas que no se relacionen, en el objeto nodo de una estructura heterogénea agregada; tomando en cuenta que no se puede permitir llenar la clase nodo con este tipo de operaciones; incrementando la complejidad al analizar cada nodo hasta encontrar el

adecuado para lanzar el indicador para realizar el funcionamiento deseado.

Solución: El patrón Visitor es el modelo orientado a objetos que permite la creación de una clase externa que actúa sobre los datos de las otras clases, este proceso es altamente útil; ya que facilita el empleo únicamente de las instancias de las clases que se necesita para solventar las necesidades del momento sin necesidad de involucrar o afectar a las que no son útiles en determinada operación.

Diagrama o Implementación:

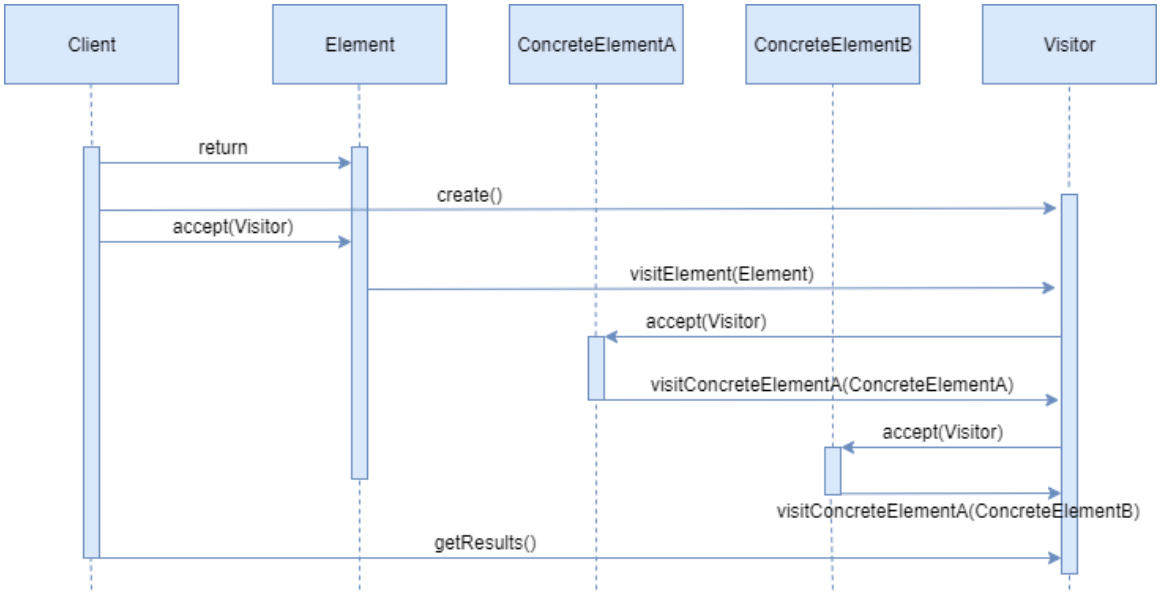

Figura 5.22: Diagrama de secuencia patrón Visitor

En la figura 5.22 se explica el comportamiento del patrón Visitor mediante un diagrama de secuencia.

- La clase client crea la estructura (Element).
- La clase client crea la instancia del Visitor a utilizar sobre la estructura.
- La clase client ejecuta el método accept de la estructura y la envía al Visitor.
- El component Element le dice al Visitor con que método lo debe procesar. El Visitor deberá tener un método para cada tipo de clase de la estructura.

- El Visitor analiza al Element mediante su método visitElement y repite el proceso de ejecutar el método accept sobre los hijos del Element. Nuevamente el Visitor deberá tener un método para procesar cada clase hija de la estructura.

- El componente ConcreteElementA le indica al Visitor con qué método debe procesarlo, el cual es visitElementA.

- La visitante continúa con los demás hijos de Element y esta vez ejecuta el método accept sobre el componente ConcreteElementB.

- El ConcrteElementB le indica al Visitor con qué método debe procesarlo, el cual es visitElementB.

- Finalmente el Visitor termina la operación sobre la estructura cuando ha recorrido todos los objetos, obteniendo un resultado que es solicitado por la clase client mediante el método getResults (el resultado es opcional ya que existen operaciones que no arrojan resultados).

Capítulo 6

Aplicación de patrones de diseño en un caso de estudio

En el presente capítulo se describe la aplicación de patrones de diseño al desarrollo de software en un sistema que servirá para gestionar una plataforma enfocada a patrones de diseño de software.

6.0.1. Descripción del caso de estudio

Para el caso de estudio se va a aplicar la metodología Proceso Unificado (UP) (Rosas 2020), pues este proceso de desarrollo es totalmente orientado a objetos y presenta gran flexibilidad a la hora de implementarlo, lo cual permite optimizar los tiempos invertidos en el desarrollo de una aplicación.
A lo largo del capítulo, a manera de ejemplo, se presentarán extractos de modelos y diagramas del módulo de seguridades del sistema.

6.0.2. Identificación del caso de estudio

La selección del caso de estudio partió de la necesidad de desarrollar un sistema con las suficientes prestaciones para poder aplicar ampliamente patrones de diseño, y de esta manera retroalimentar las actividades descritas en la guía de patrones de diseño en la cual explicamos el uso de cada uno de ellos, al aplicarla en un caso real

de desarrollo de software.

El caso de estudio a desarrollar debe manejar diferentes tipos de conexión a bases de datos por medio del patrón de diseño factory, de manera que pueda demostrarse mejor el uso de patrones de diseño para controles de seguridad, adicionalmente, el caso de estudio debe corresponder a una aplicación orientada a objetos.

6.0.3. Aplicación de patrones de diseño en la implementación del caso de estudio

A continuación se implementarán paso a paso las actividades descritas en la guía para el uso de patrones de diseño en el desarrollo de software, estas actividades son:

- Modelado de la solución.
- Análisis detallado.
- Arquitectura de la solución.
- Diseño de la aplicación.
- Codificación del sistema.
- Análisis del uso de la guía en el caso de estudio.

6.0.4. Caracterización del sistema

Objetivo

La caracterización del sistema constituye un proceso de entendimiento de la solución por parte del equipo de desarrollo. Para el caso de estudio se ha optado por desarrollar un sistema que automatice el proceso de desarrollo de software por medio de una plataforma de gestión de patrones de diseño .

Requerimientos del sistema

A continuación se describirán los requisitos de desarrollo del sistema de software; cuales funciones deberá cumplir y cuales funciones no están consideradas para su desarrollo.

Aquellos requerimientos que definirán la funcionalidad del sistema:

Responsabilidades

- Registrar, modificar y buscar patrones de diseño con las siguientes consideraciones:
    - Registro: La creación de un patrón de diseño debe incluir la siguiente información correspondiente a:
        - Nombre
        - Función
        - Estructura
        - Objetivo
        - Aplicaciones
        - Patrones de diseño colaboradores
        - Campo de acción
        - Problematica
        - Solución
        - Diagrama o implementación
    - La búsqueda en el foro de discusión de patrones de duiseño puede ser por cualquiera de los atributos de cada patrón de diseño.
- Registrar, modificar, eliminar y buscar artículos sobre patrones de diseño con las siguientes consideraciones:

- Registro: La creación de un artículo sobre patrones de diseño debe incluir la siguiente información correspondiente a:
  - Título
  - Imágenes
  - Código de programación con soporte a Python, Java y PHP
  - Descripción
- La búsqueda de información sobre artículos de patrones de diseño se podrá realizar mediante el buscador del foro de discusión.
- Sólo el usuario que es autor del artíuclo puede editarlo o eliminarlo.
- Sólo los usuarios que previamente se han registrado en la plataforma pueden comentar o agregar a sus favoritos algún artículo que este disponible en la plataforma.

- Sistema de mensajeria instantánea, los usuarios registrados en la plataforma tienen la posibilidad de mandar mensajes de texto a otros usuarios de la plataforma por medio su perfil y con las siguientes consideraciones:
  - Título
  - Descripción del mensaje
- Registro y edición de usuarios, el registro de un usuario deberá contener la información personal de dicho usuario, cómo son:
  - Nombre
  - Apellidos
  - Correo electrónico
  - Contraseña
  - Foto de perfil

Exclusiones

- El sistema no manejará información contable de ningún tipo, considerando que el costo de suscripción.
- El sistema no se encargará de administrar los registros d los usuarios.
- El sistema no manejará información correspondiente a proveedores de publicaciones de ningún tipo (donantes, distribuidoras, etc.).

6.0.5. Requerimientos no fucionales

Los siguientes requisitos no funcionales serán implementados para cubrir requerimientos complementarios del usuario.

- Seguridad: Control de acceso a la aplicación por usuarios y perfiles de usuario.
- Mantenibilidad: La aplicación deberá ser capaz de soportar cambios a lo largo de su vida útil.
- Confiabilidad: La aplicación no deberá presentar errores en su ejecución.

6.0.6. Modelado de la solución

El sistema estará compuesto por los siguientes módulos: (Gamma, Helm, and Johnson 1995)

- Gestor de artículos.
- Gestor de Likes.
- Gestor de usuarios.
- Gestor de mensajes.
- Gestor de patrones de diseño.
- Gestor de comentarios.

- Gestor de foro de discusión.
- Gestor de seguridad.

6.0.7. Análisis detallado

En esta sección se ilustrará la solución a desarrollarse en un entorno técnico.

Funcionalidades

6.0.8. Identificación de problemas

Los posibles problemas en el desarrollo de la aplicación son:

- Problemas de creación de objetos (Instanciación)
  - Complejidad en la creación de objetos, debido a la necesidad de dinamizar la aplicación y su funcionamiento.
  - Limitada escalabilidad y mantenimiento complejo de la aplicación en caso de que se acoplen los objetos creadores con los objetos creados.
  - Sobrecarga de memoria en la creación de objetos similares (publicaciones)
  - Falta de control en la creación y acceso a objetos .
- Problemas de la estructura de los objetos
  - Complejidad de desarrollo, escalabilidad y mantenimiento de la aplicación debido al uso de objetos compuestos por uno o varios objetos jerárquicos, como perfiles de usuario y publicaciones.

6.0.9. Diagramas de clases de la aplicación

Para esta sección se ha descrito la aplicación de patrones de diseño en la implementación del caso de estudio, en el cual se puede apreciar claramente la inclusión de patrones de diseño, para observar toda la aplicación.

6.0.10. Diagrama del Módulo de seguridades

Los patrones de diseño incluidos en el diagrama de clases mostrado en la Figura 6.1, son: Composite y Command.

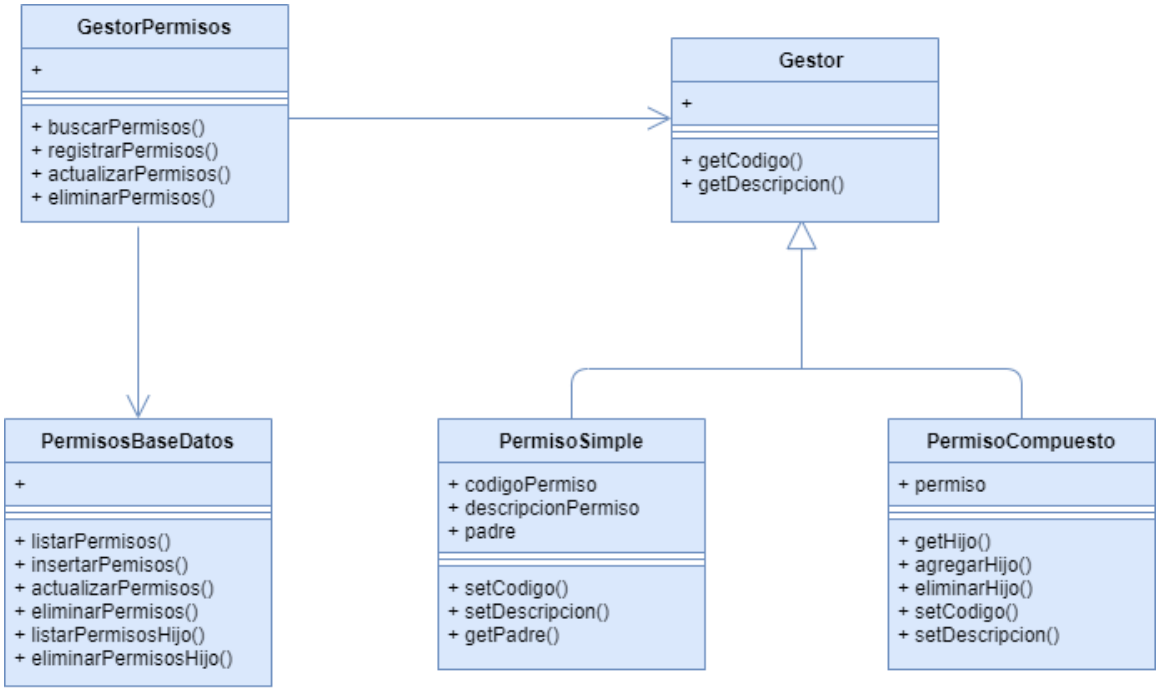

Figura 6.1: Implementación del requerimiento de gestión de permisos de perfiles de usuario

- **Gestor:** De tipo composite, permite gestionar fácilmente la jerarquía de permisos sin necesidad de redundar código para gestionar elementos simples y compuestos, adicionalmente, en caso de que se desee incorporar nuevas características a un elemento de tipo composite, simplemente se agregarán estas características en las clases concretas (PermisoSimple y PermisoCompuesto), y al

referenciarlos mediante su clase abstracta, esta actualización de características no implicará una actualización de las clases clientes del patrón.

- **GestorPermisos:** De tipo command, permite separar la implementación de un objeto, de las operaciones que lo gestionan, sujetándonos al caso anterior, en caso de requerir nuevas operaciones de gestión de los objetos, se podrá actualizar este objeto, y hacer uso de la nueva funcionalidad, sin que los objetos que utilizaban la funcionalidad anterior se vean afectados.

6.0.11. Análisis del uso de patrones de diseño en la implementación de un proyecto de software

El uso de patrones de diseño permitió la exitosa inclusión de 4 patrones de diseño a la aplicación seleccionada para el framework de gestión de patrones de diseño, con lo cual se mejoraron principalmente los aspectos relacionados con Escalabilidad, Seguridad, Modularidad, y Calidad del sistema; además permitió optimizar los tiempos de desarrollo de la misma al reciclar la lógica de programación de los patrones de diseño implementados.

La escalabilidad de la aplicación se mejora gracias a que se utilizan clases abstractas e interfaces a la hora de relacionar objetos, de esta manera se puede agregar nuevos objetos, que implementen dichas interfaces, sin afectar a las clases que manipulan esos objetos.

La seguridad de la aplicación se mejoró gracias a la inclusión del patrón Singleton, en este caso el patrón gestiona completamente el acceso a la base de datos, evitando que existan conexiones paralelas, el uso de una capa de acceso a datos permite controlar la parametrización de consultas y transacciones SQL, evitando la inyección de código SQL.

El uso de patrones de diseño mejora la modularidad del sistema, permite mejorar

la cohesión entre objetos, reduciendo notablemente su acoplamiento, haciendo factible la modificación de funcionalidad existente, la inclusión de nueva funcionalidad y el reemplazo de la misma por una nueva. Es importante indicar que el caso de estudio presento las prestaciones necesarias para poder aplicar la guía desarrollada, sin embargo, ésta guía presenta además opciones útiles a la hora de estudiar los patrones de diseño, individual y colectivamente.

Capítulo 7

Resultados

7.0.1. Vista general al caso de estudio plataforma de Gestión de Patrones de Diseño

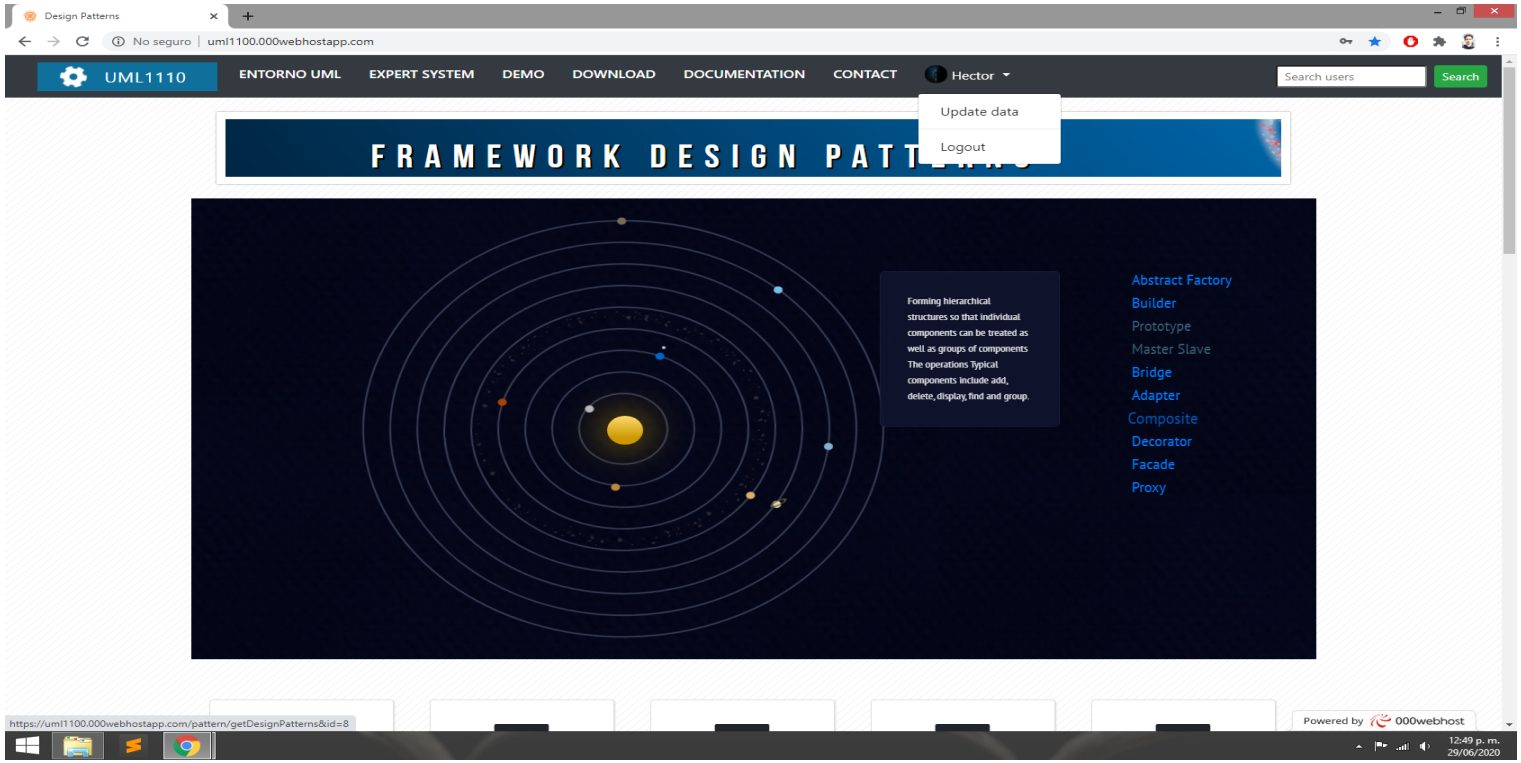

Figura 7.1: GUI Página principal

En la figura 7.1, se puede apreciar la página principal de la plataforma, en ella se enceuntra implementada una galaxia dinámica de patrones de diseño de software, la cual nos redirecciona al patrón de diseño que seleccionamos.

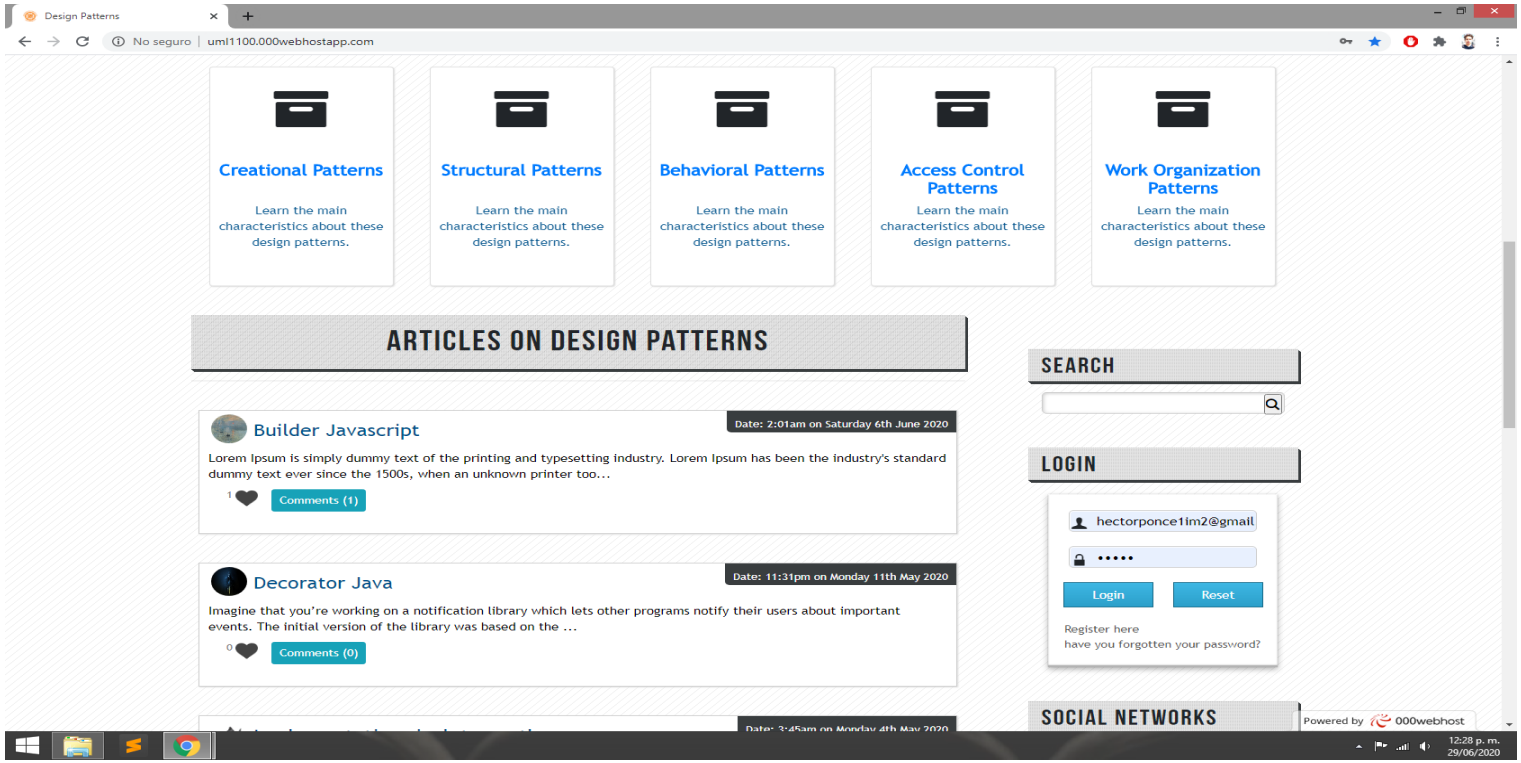

Figura 7.2: GUI Categorias de patrones de diseño

En la figura 7.2, se puede apreciar un panel de selección con las diferentes categorias de patrones de diseño que se encunetran en la plataforma, ademas de un breve vistazo al foro de discusión de patrones de diseño y lo paneles de búsqueda e inicio de sesión.

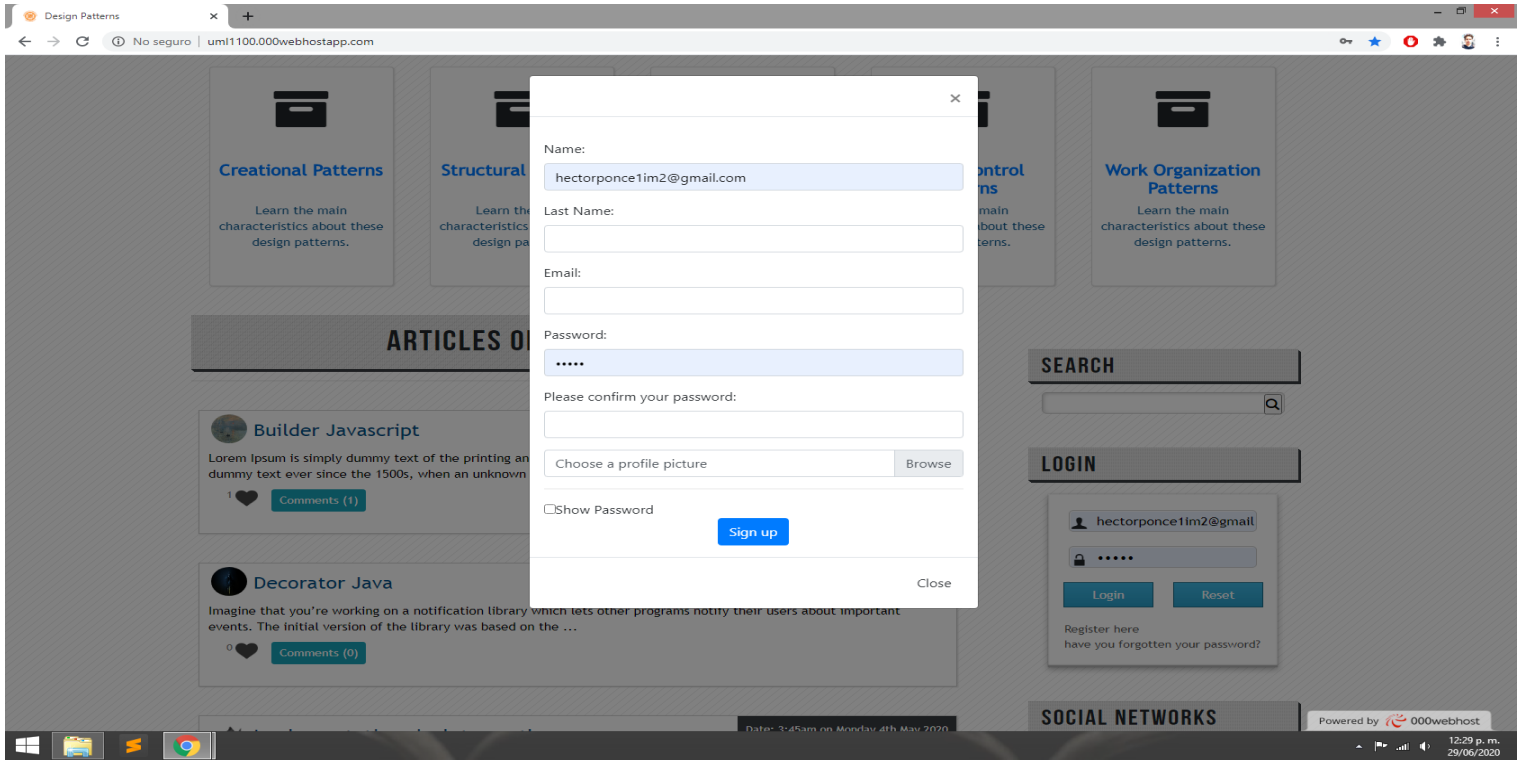

Figura 7.3: GUI Registro de usuarios

Para que un usuario pueda hacer uso de las diferentes funcionalidades que se encunetran disponibles en la plataforma, cómo son; entorno de desarrollo UML, foro de discusión y sistema de mensajeria, debe registarme por medio de un formulario, ver figura 7.3, proporcionando datos personales, cómo nombre, apellidos, contraseña y foto de perfil.

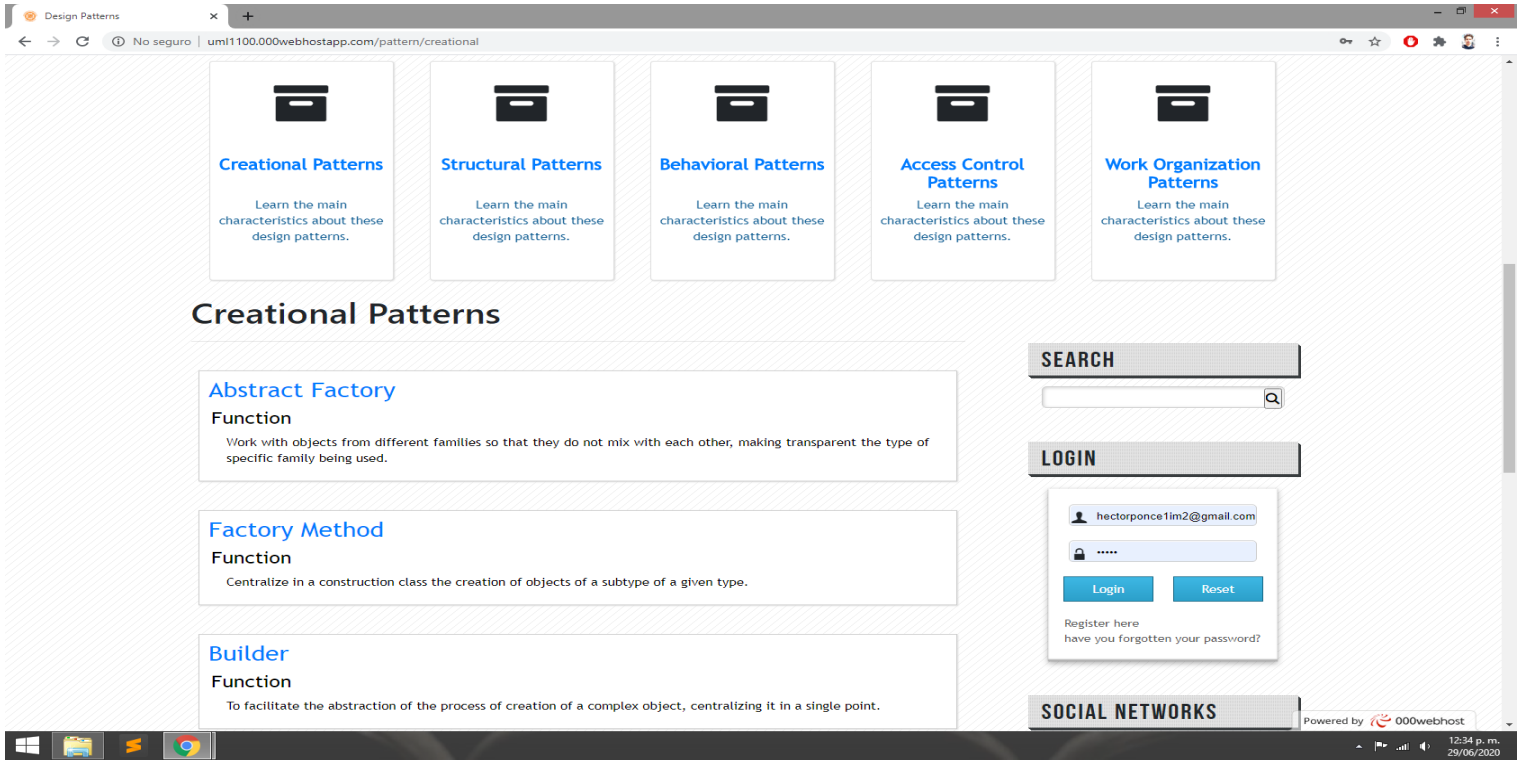

Figura 7.4: GUI Categoría patrones de diseño

Cuándo seleccionamos una categoría de patrones de diseño, nos despliega todos los patrones asociados a esa categoría, ver figura 7.4.

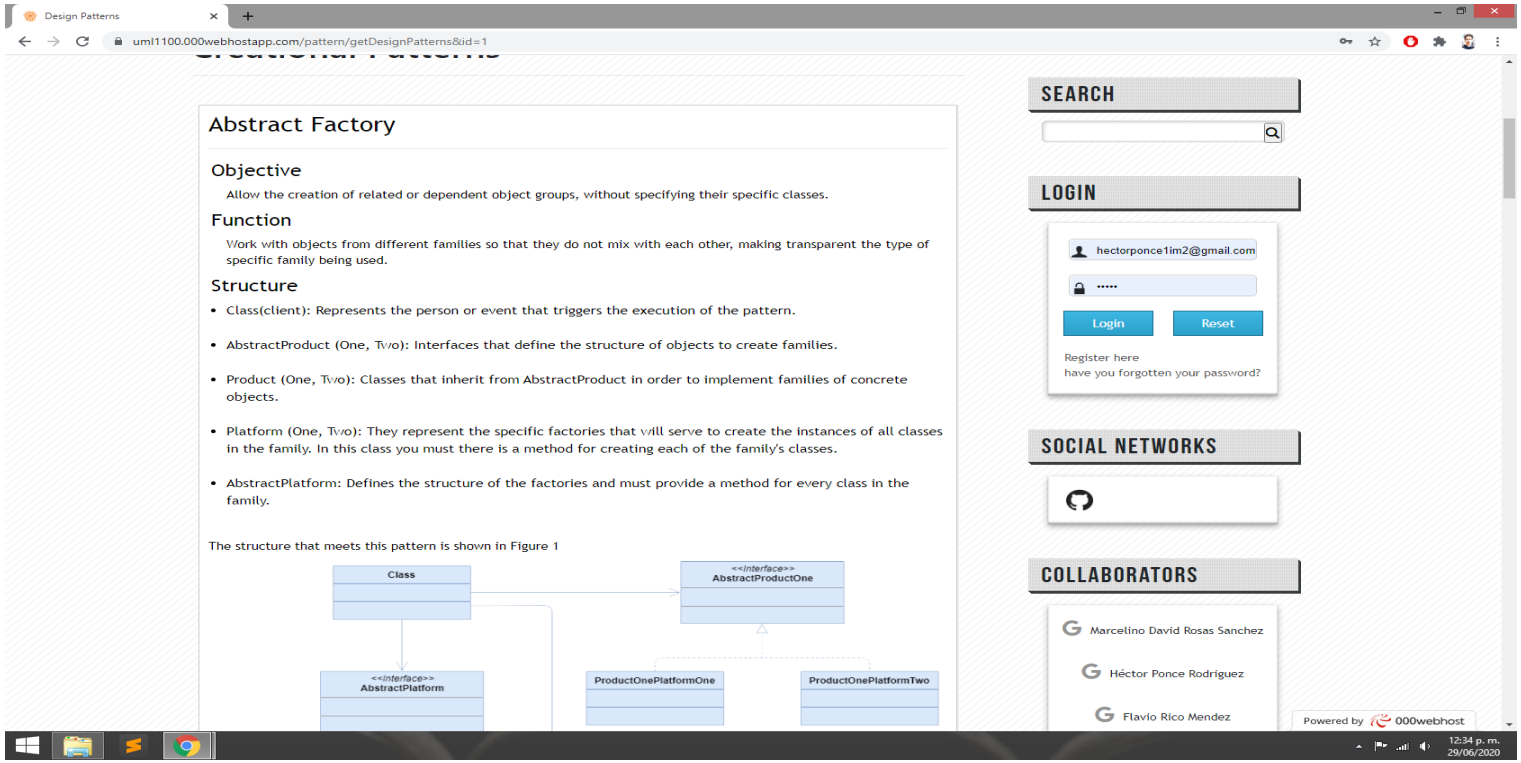

Figura 7.5: GUI Información de un patrón de diseño

En la figura 7.5, se puede apreciar que al momento de seleccionar un patrón de diseño, nos proporciona toda la información respecto a ese patrón, cómo son; título, estructura, función, aplicaciones, etc. Además de ilustrar dicha información con diagramas UML.

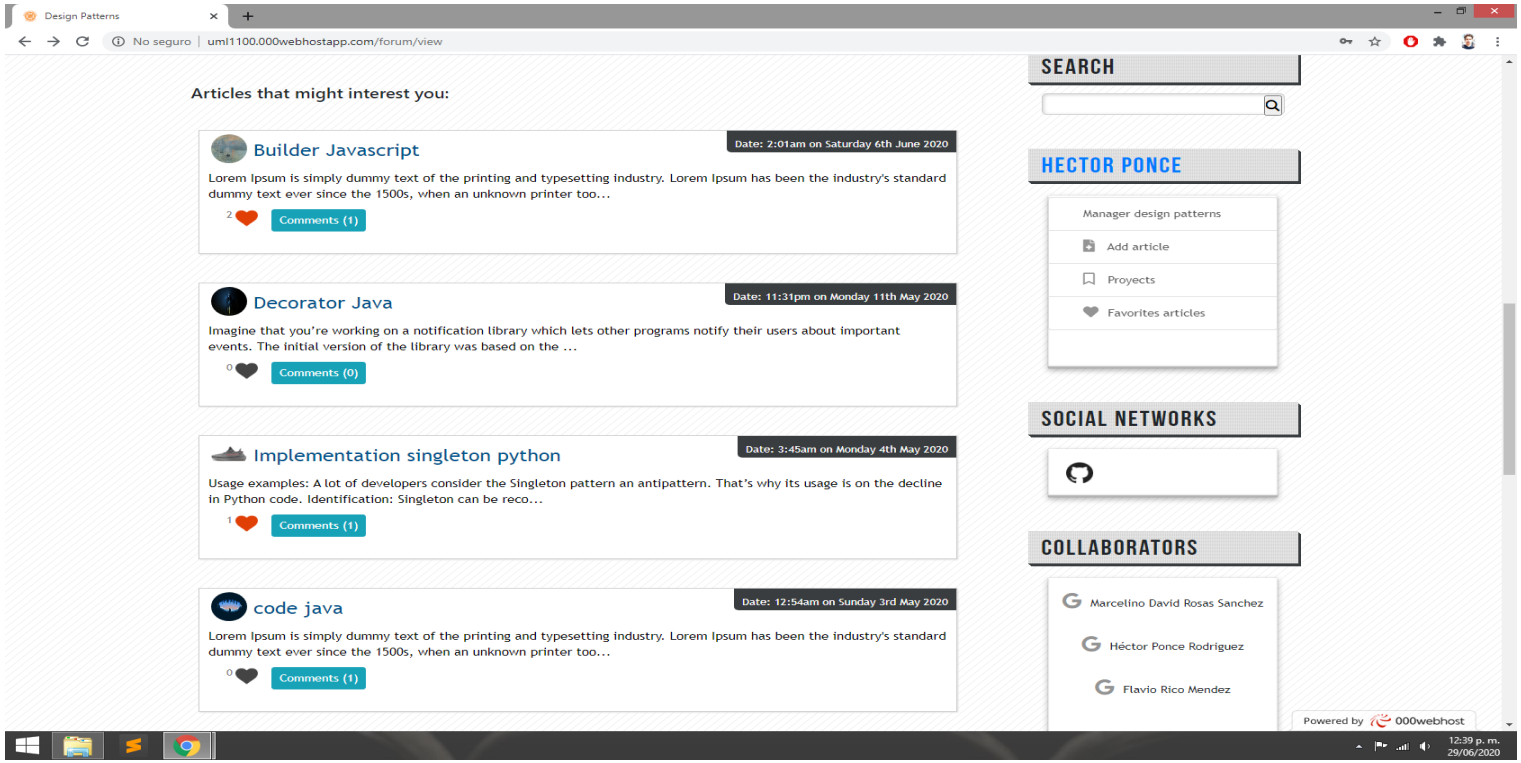

Figura 7.6: GUI Foro de discusión sobre patrones de diseño

La plataforma de gestión de patrones de diseño cuenta con un subsistema el cual consiste en un foro de discusión de patrones de diseño, ver figura 7.6, en la cual nos despliega los últimos artículos, dudas o propuestas de los usarios de la plataforma han subido.

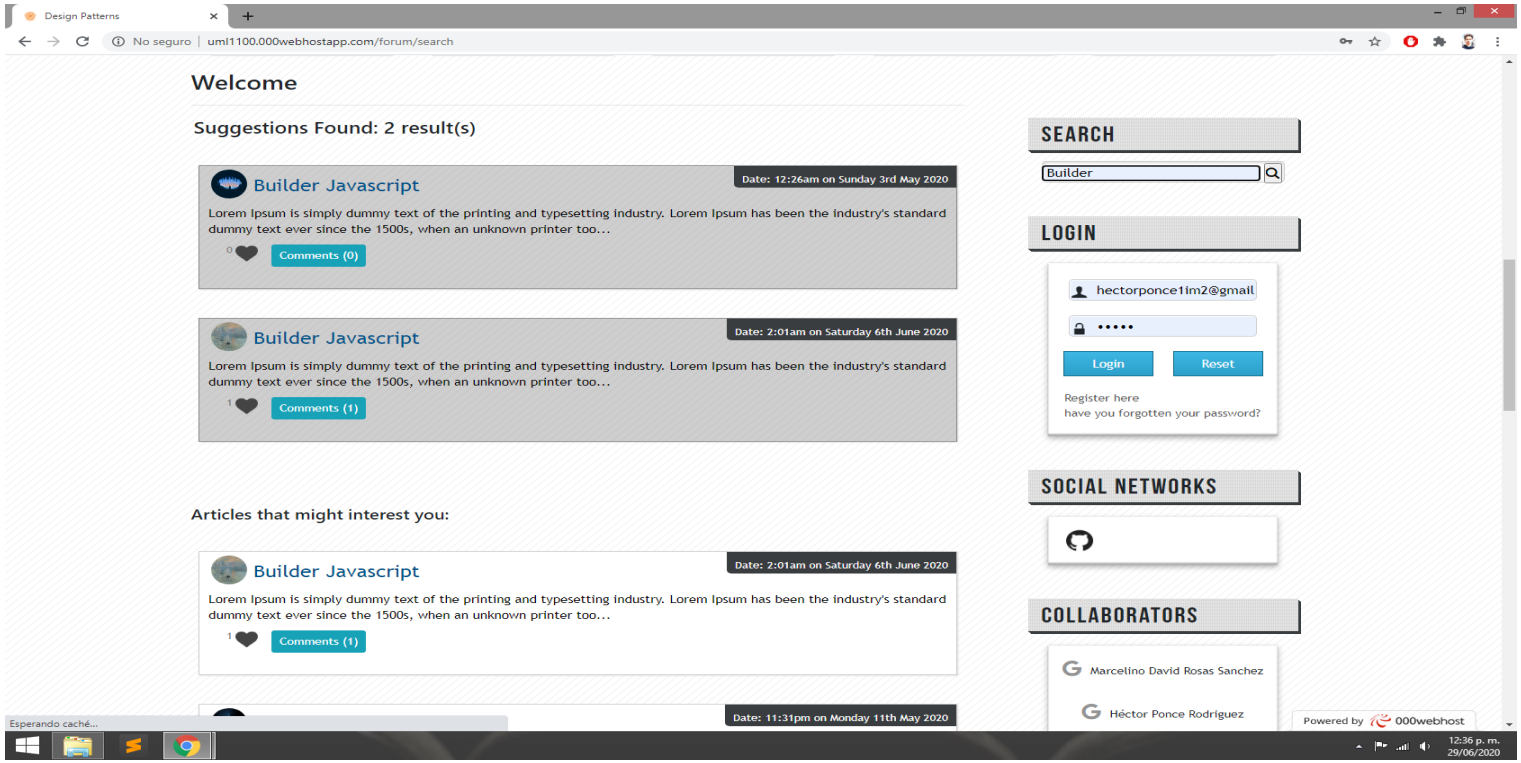

Figura 7.7: GUI Panel de búsqueda para artículos relacionados con patrones de diseño

Para que la experiencia de un usuario sea más por la plataforma, tiene a su disposición un panel de búsqueda, ver figura 7.7, para poder filtrar información relacionada con patrones de diseño.

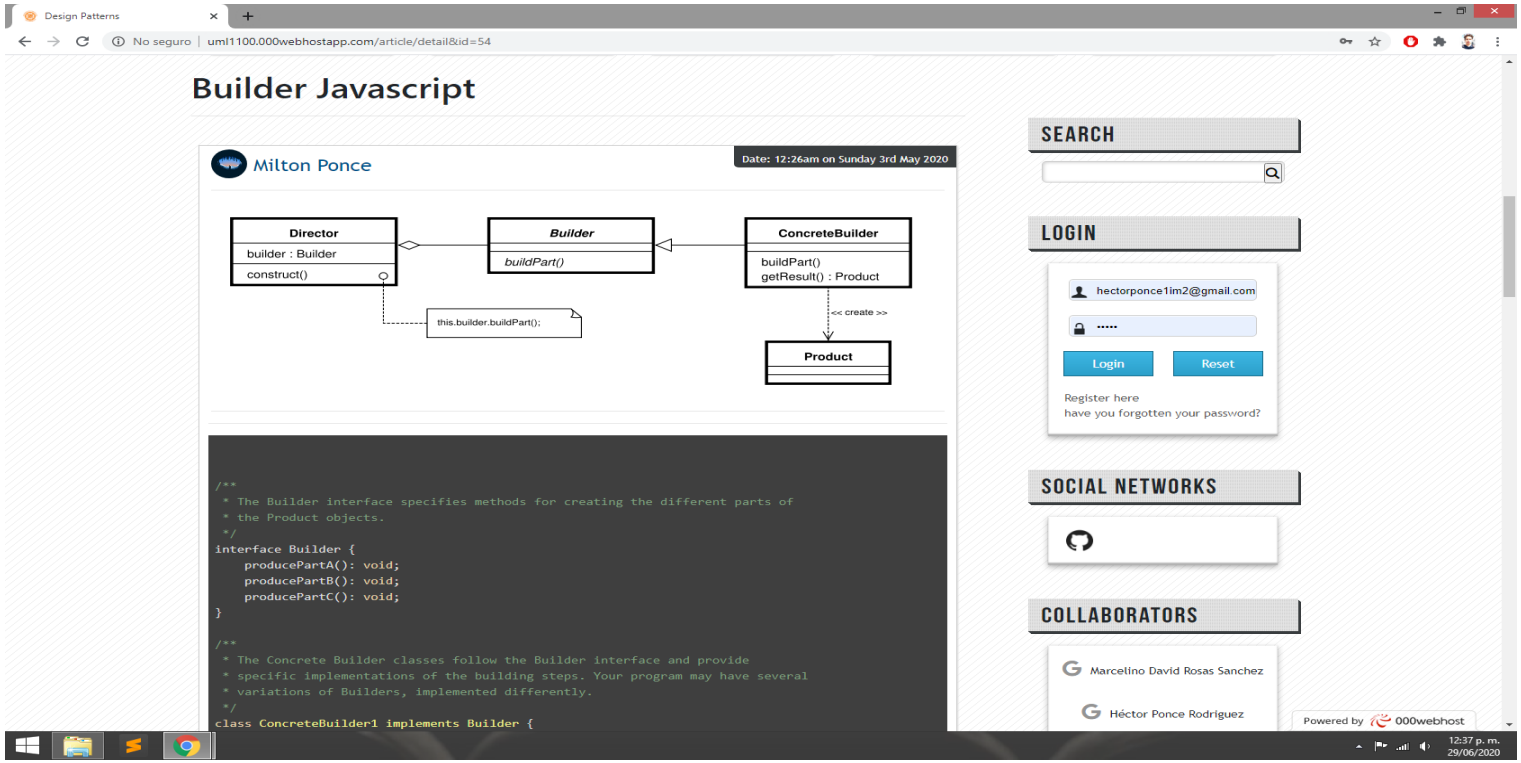

Figura 7.8: GUI Artículo del foro de discusión - Parte 1

Cuándo accedemos a un artículo que está disponible en la plataforma, ver figura 7.8, nos muestra información cómo son; título, usuario que dio de alta dicho artículo, imágines y código de alguna implementación si asi lo decidio el usuario.

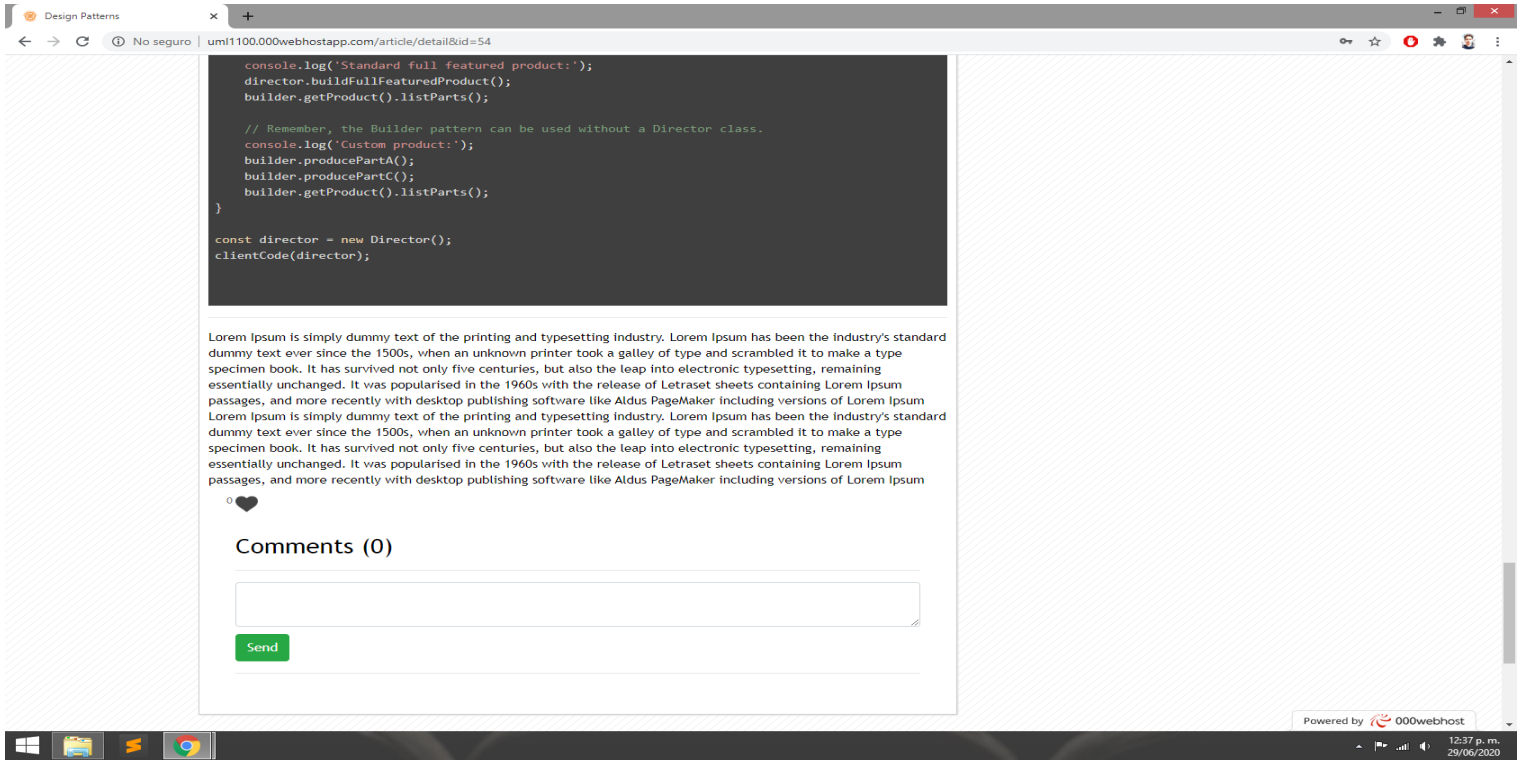

Figura 7.9: GUI Artículo del foro de discusión - Parte 2

En la figura 7.9 podemos apreciar la continuación de la GUI dónde consultamos algun artículo disponible en la plataforma, para hacer más interactiva la experiencias de los usuarios, tienen las opciones de hacer algun comentario sobre el artículo e indicar por medio del botón en forma de corazón si les gusto la información del artículo.

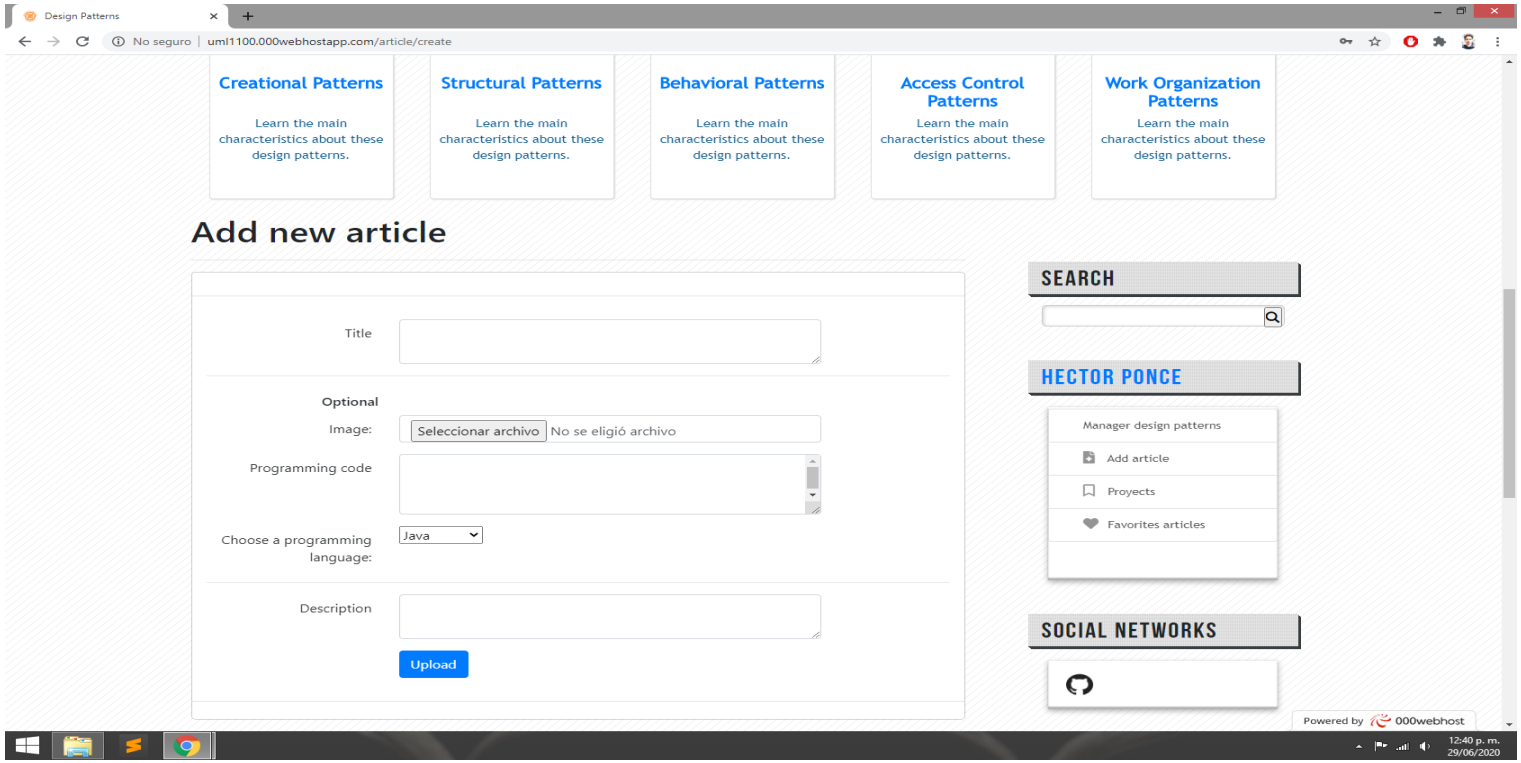

Figura 7.10: GUI Agregar artículo al foro de discusión sobre patrones de diseño

Cuándo un usuario tiene información la cual quiere contribuir en la plataforma, tiene la opción de agregar un artículo al foro de discusión por medio de un formulario, ver figura 7.10, en la cual puede incluir información cómo; un título, imágenes ó código de programación sobre alguna implementación si asi lo desea y una descripción.

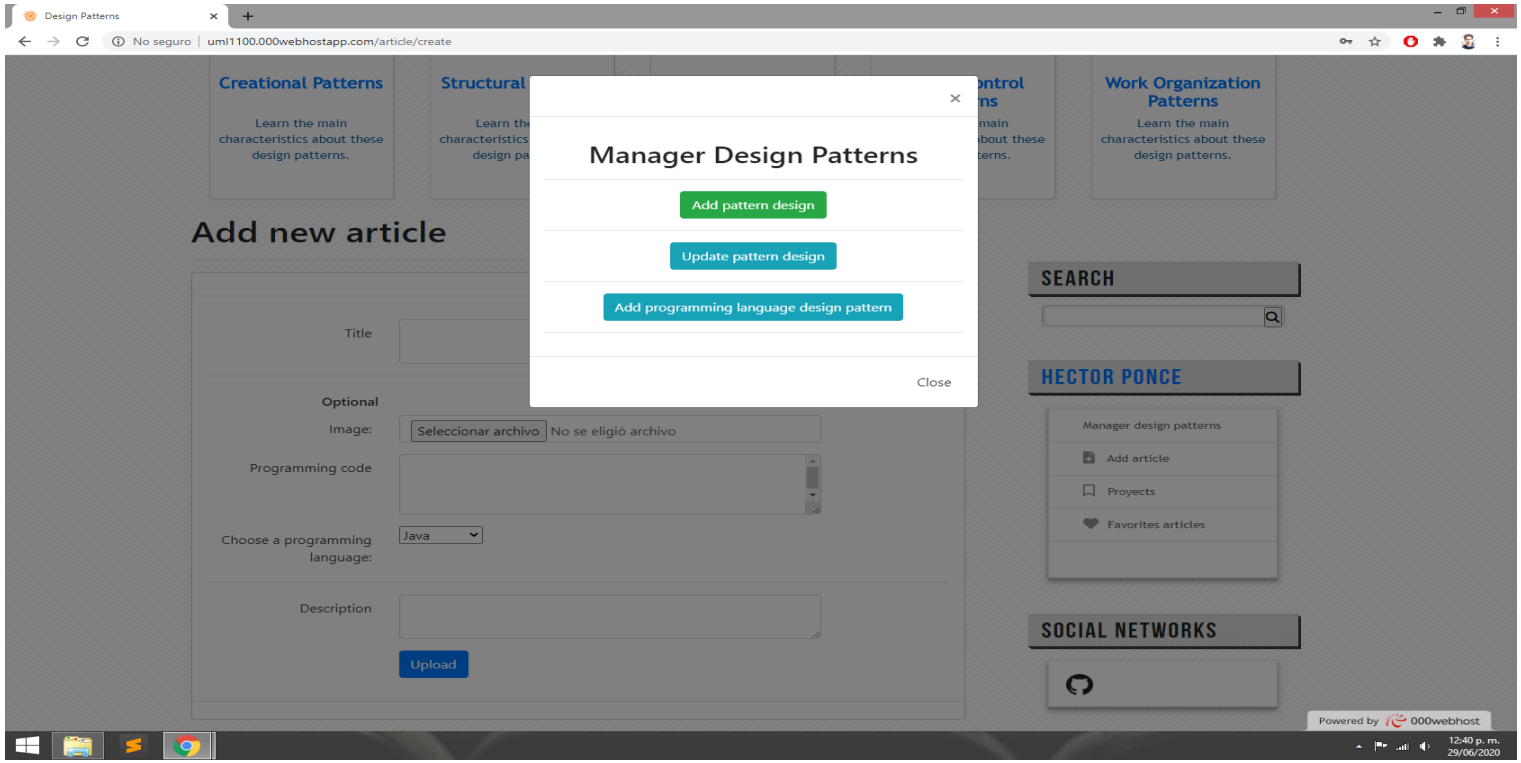

Figura 7.11: GUI Gestor de patrones de diseño

Solamente los usuarios que tengan las credenciales de super usuarios en la plataforma, pueden acceder al gestor de patrones de diseño, ver figura 7.11, con el cual pueden estar gestionando el contenido correspondiente a patrones de diseño de la plataforma.

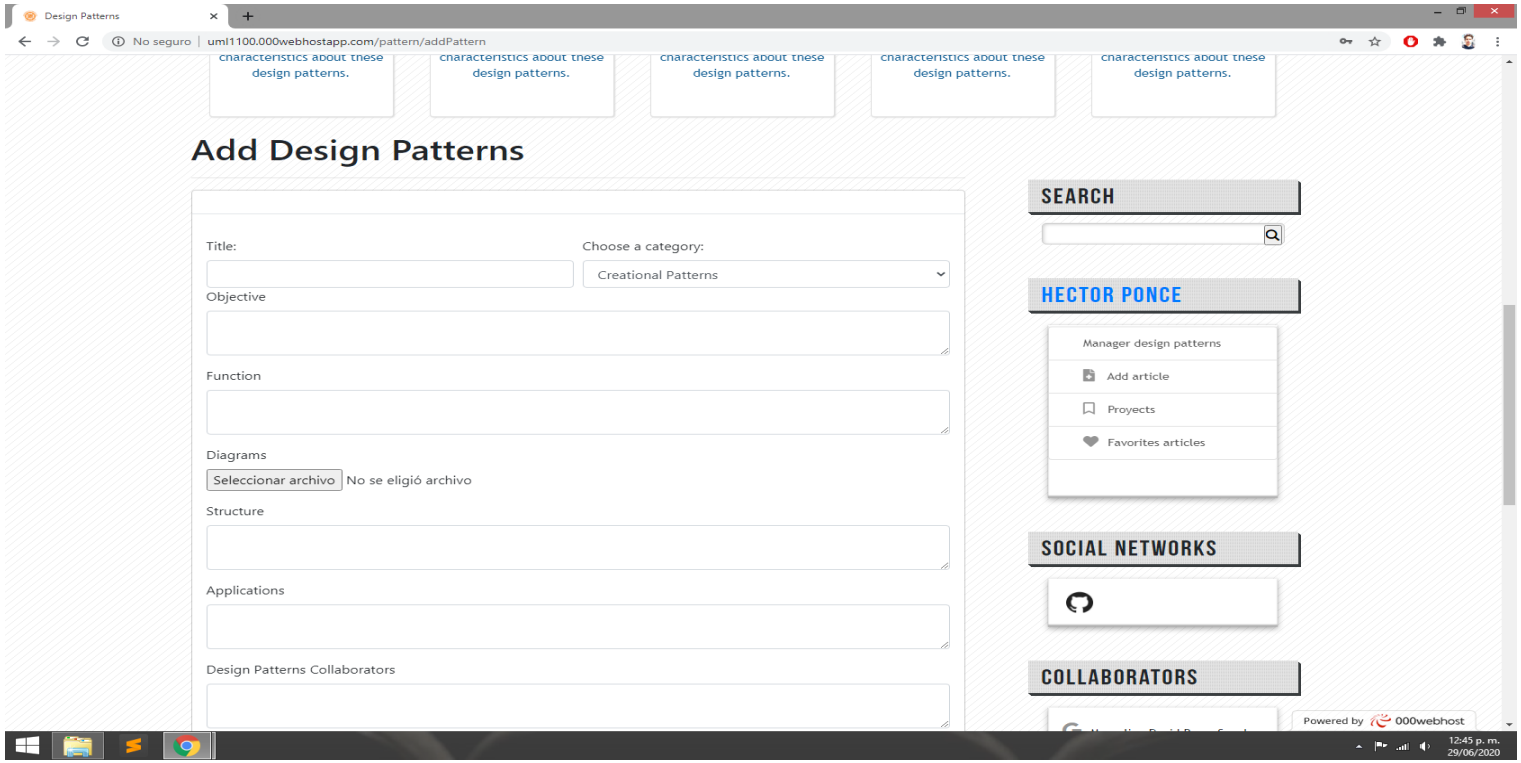

Figura 7.12: GUI Añadir/Modificar información sobre patrones de diseño

Los administradores de la plataforma que se encargan de gestionar el contenido correspondiente a patrones de diseño de la plataforma, tienen a su disposición el formulario que se puede apreciar en la figura 7.12, con el cual pueden estar gestionado la información de una manera más comoda.

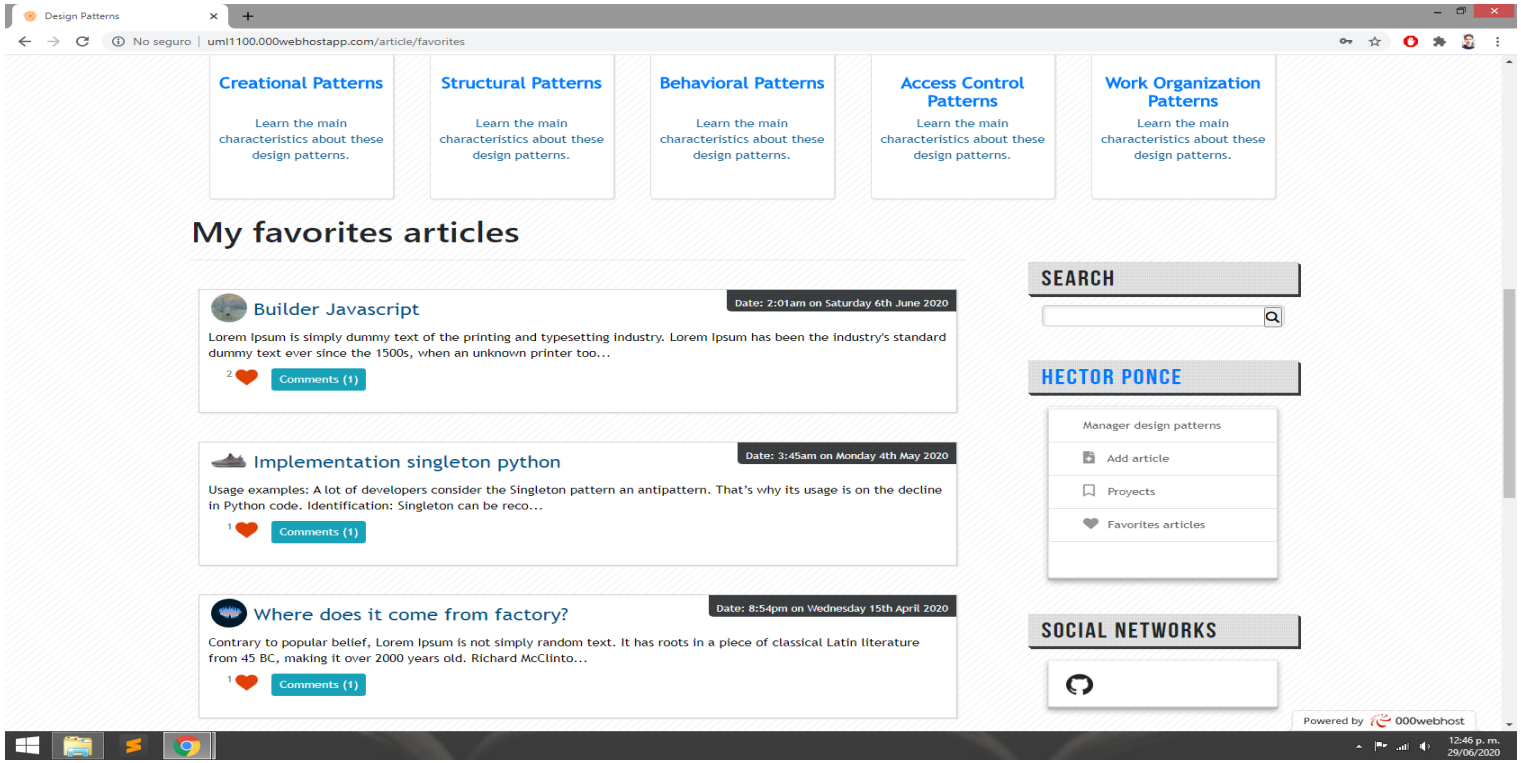

Figura 7.13: GUI Mis artículos favoritos

En la figura 7.13, se puede apreciar que cuándo un usuario preciona el botón de "me gusta.[en] forma de corazón sobre un artículo, este se agrega automáticamente a una sección de su perfil en dónde se listan todos estos artículos.

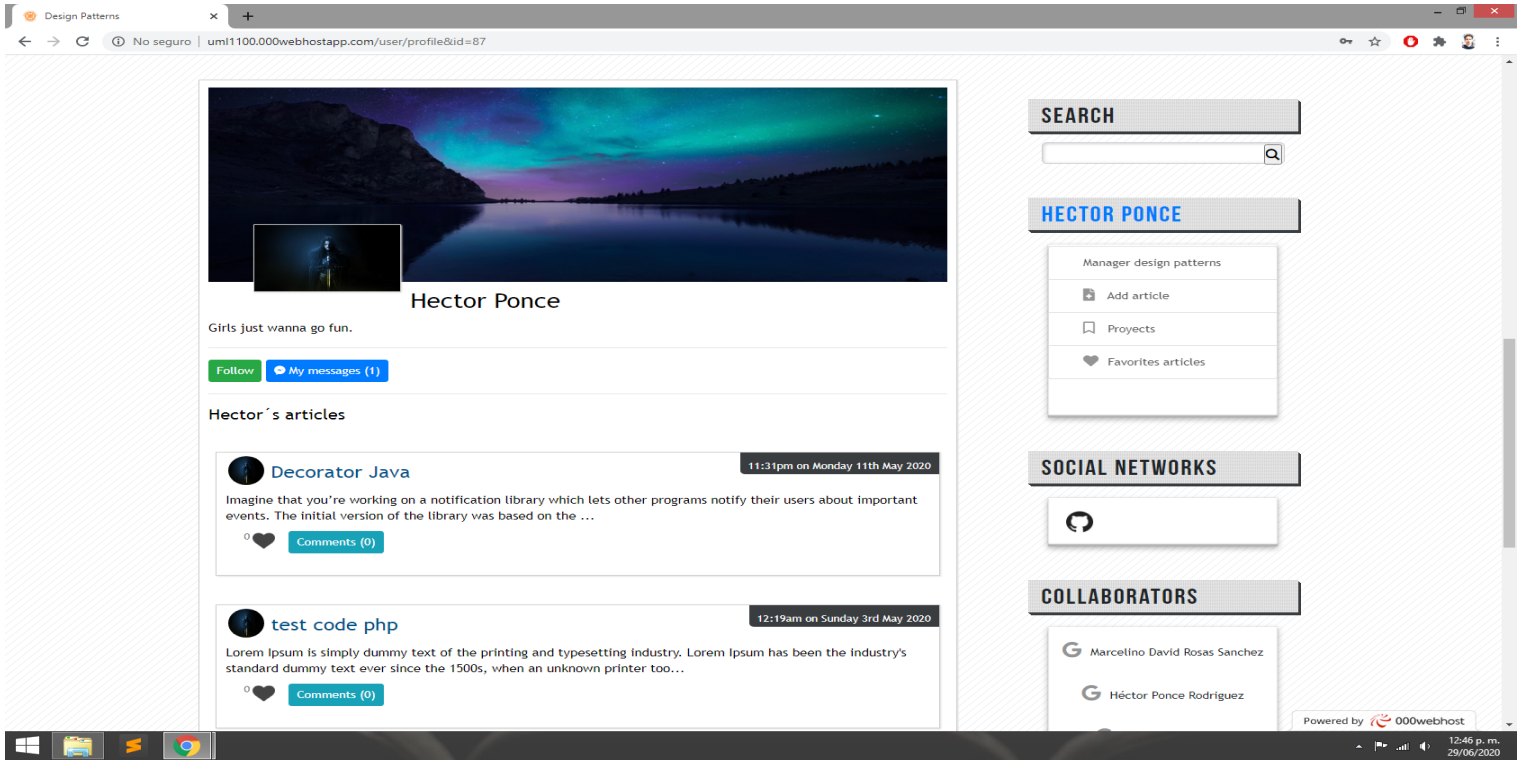

Figura 7.14: GUI Perfil de usuario

En la figura 7.14, se puede apreciar la GUI correspondiente al perfil de usario de un usuario que previamente se registro en la plataforma, en el cual se pueden apreciar secciones dónde se visualizan todos los artículos que previamente ha agregado en la plataforma y un botón con el nombre de "mis mensajes.[en] dónde le indica el número de mensajes que le faltan por leer o contestar.

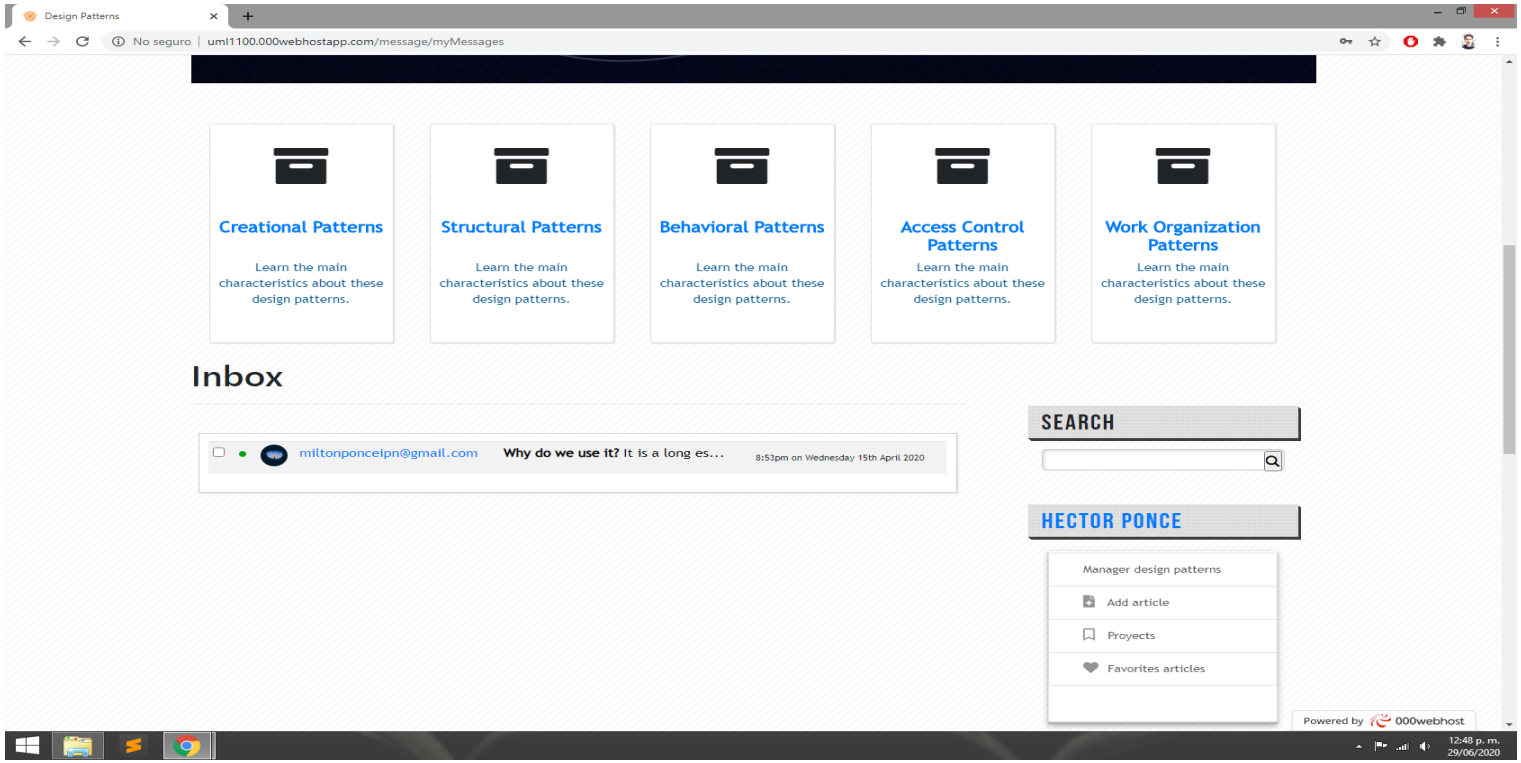

Figura 7.15: GUI Bandeja de entrada

En la figura 7.15, se puede apreciar la bandeja de entrada con los mensajes que previamente se mandarón a esa dirección de correo electronico.

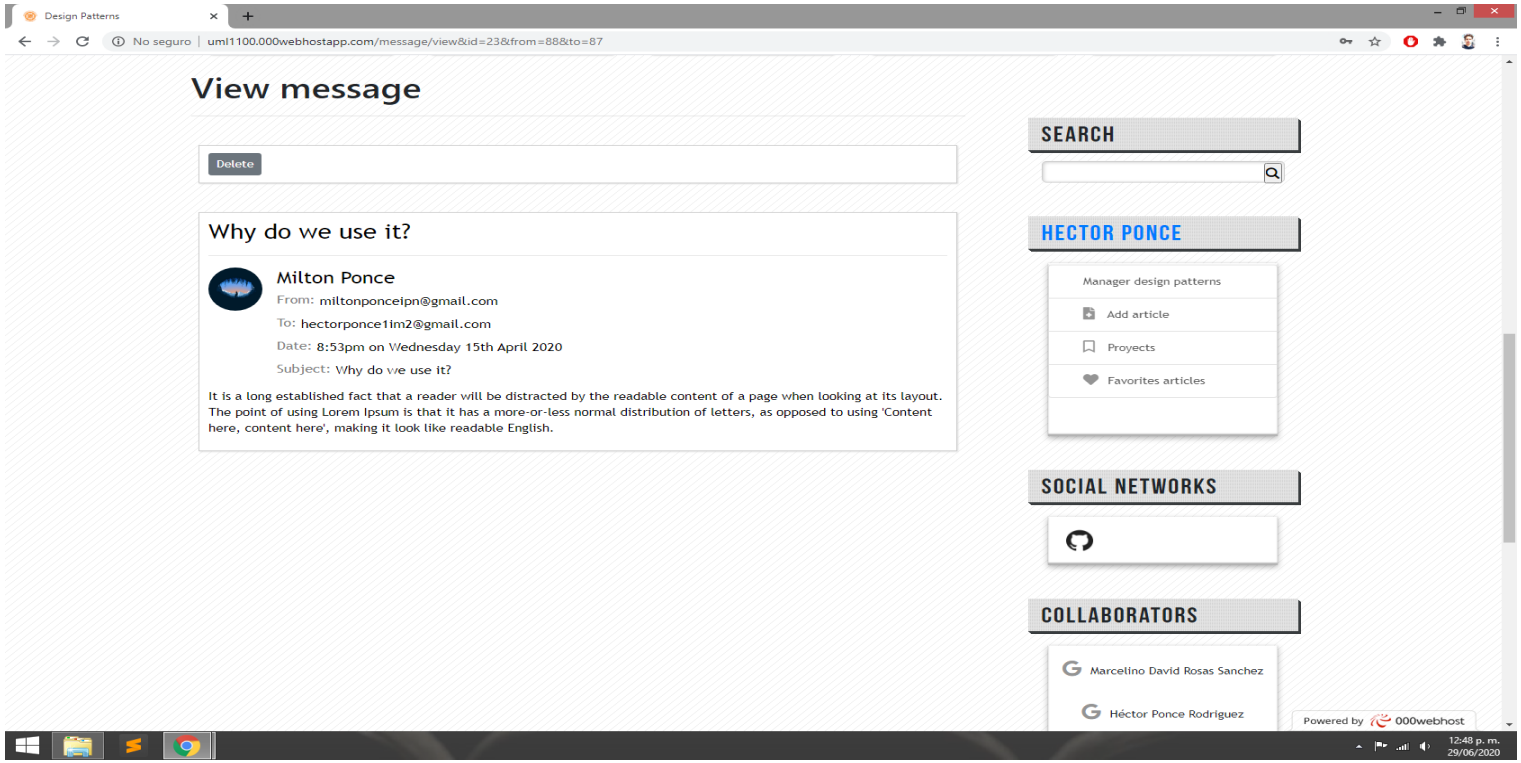

Figura 7.16: GUI Mensaje de texto

Cuándo un usuario selecciona un mensaje de su bandeja de entrada, este se visualizará cómo se puede apreciar en la figura 7.16.

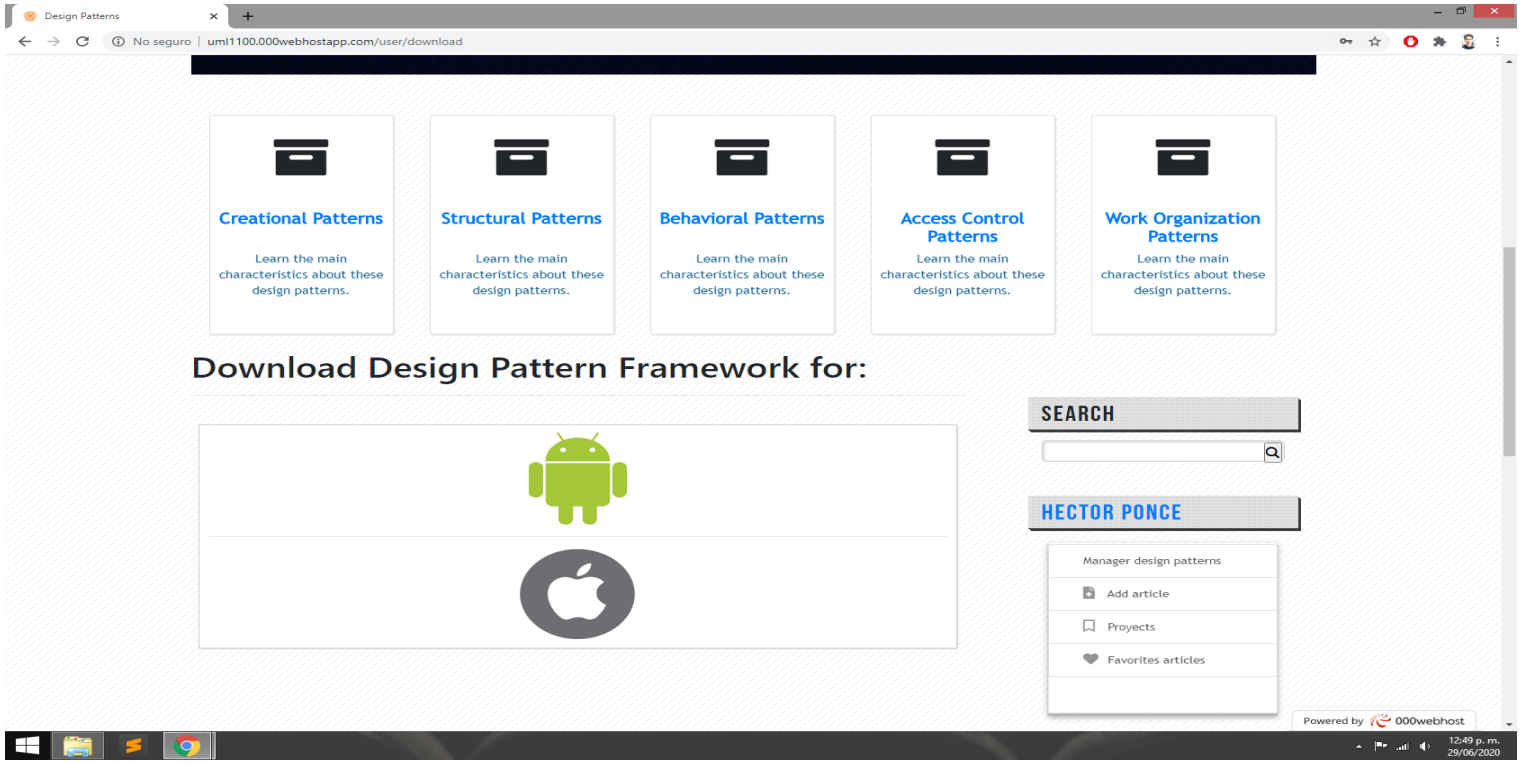

Figura 7.17: GUI Descargar plataforma de gestión de patrones de diseño

En la figura 7.17, se puede apreciar la página dónde estan las dos opciones de descarga de la plataforma de gestión de patrones de diseño para dispositivos móviles.

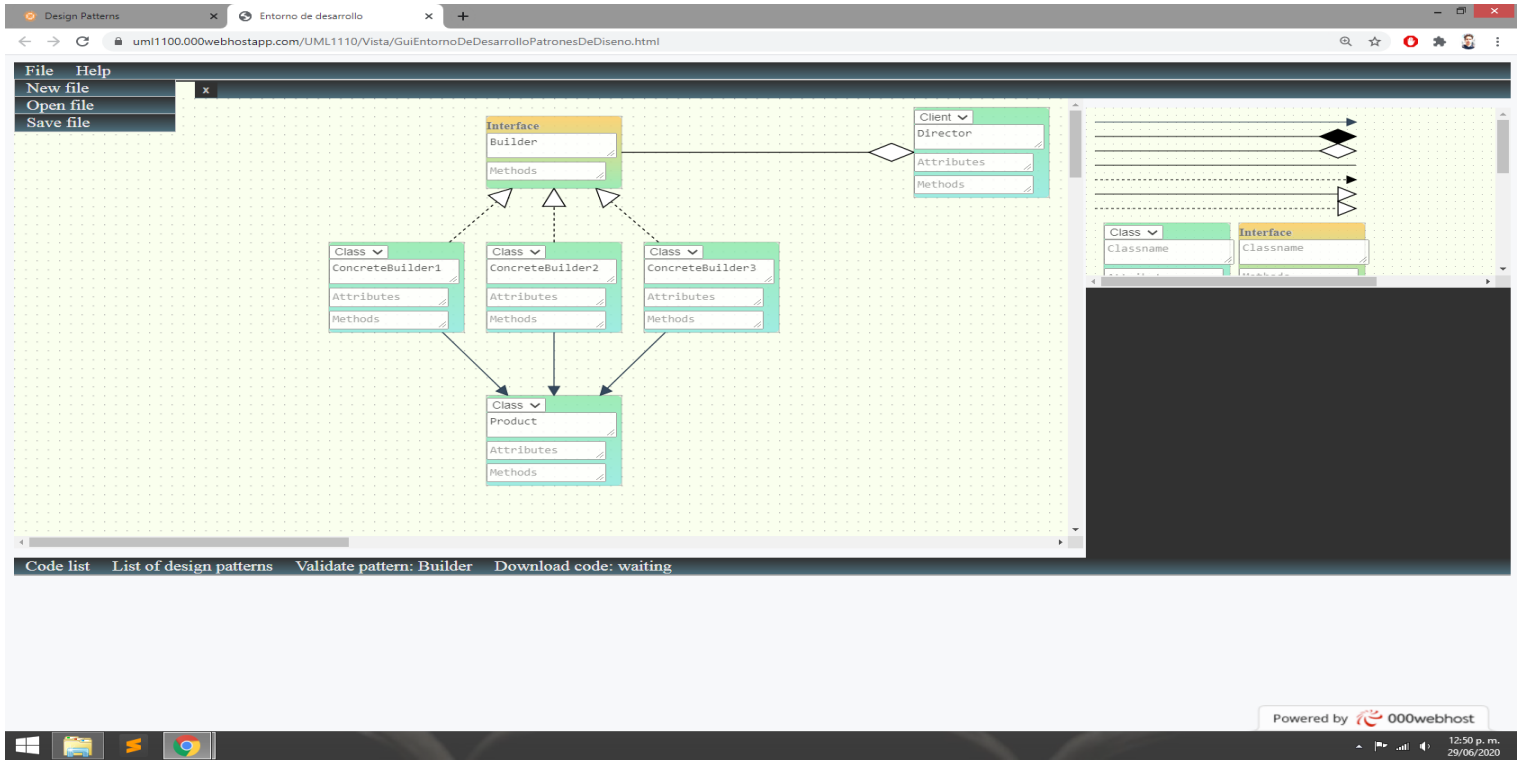

Figura 7.18: GUI

En la figura 7.18, podemos apreciar el subsistema Entorno de Desarrollo UML, en el cual podemos cargar diagramas genéricos de patrones de diseño para posteriormente poder descargar su implementación en código PHP, Python Y Java.

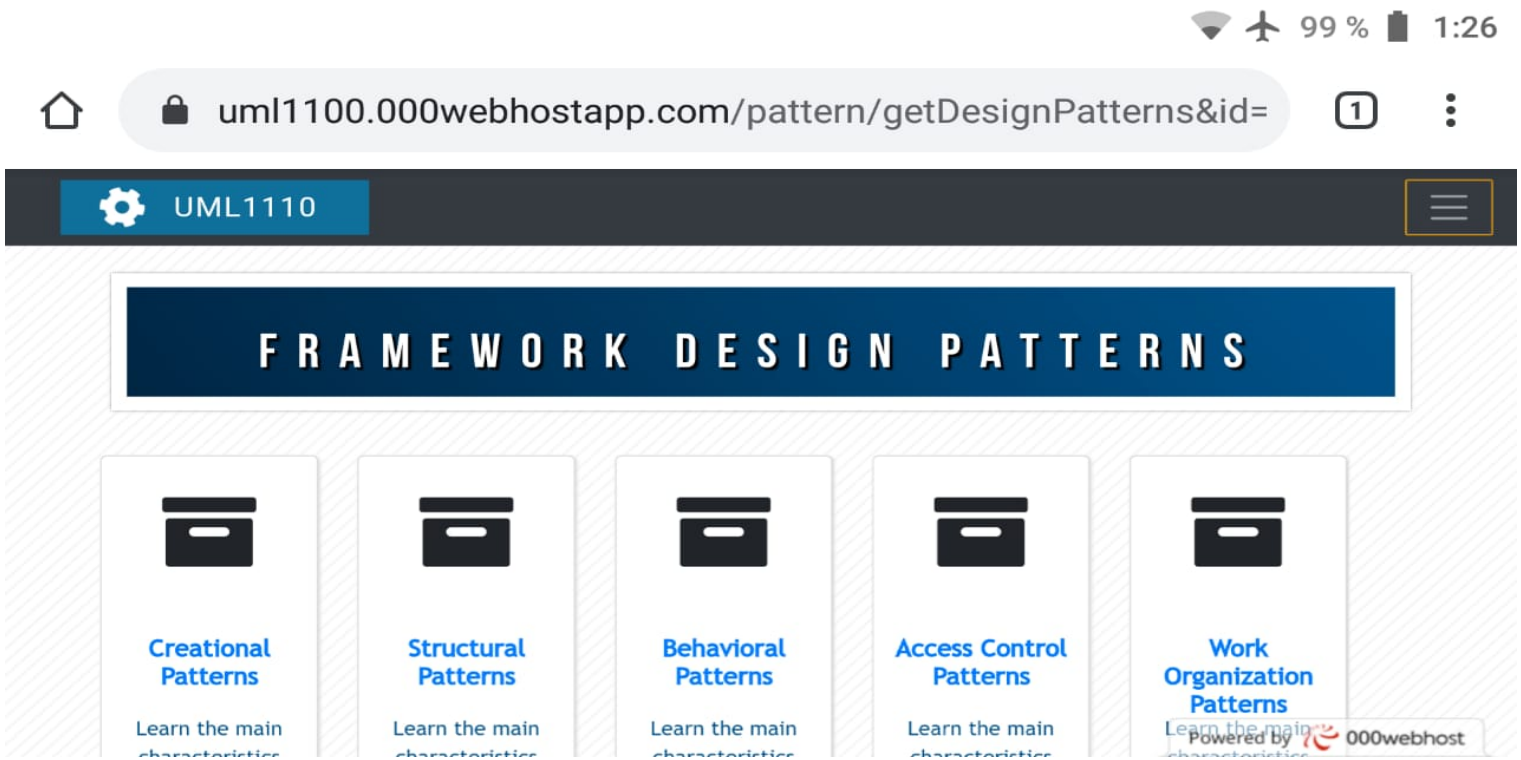

Figura 7.19: GUI Plataforma de gestión de patrones de diseño versión navegador web móvil

Un usuario puede acceder desde el navegador de su dispositivo móvil, ver figura 7.19, y el diseño de la plataforma se ajusta dependiendo la resolución del dispositivo móvil.

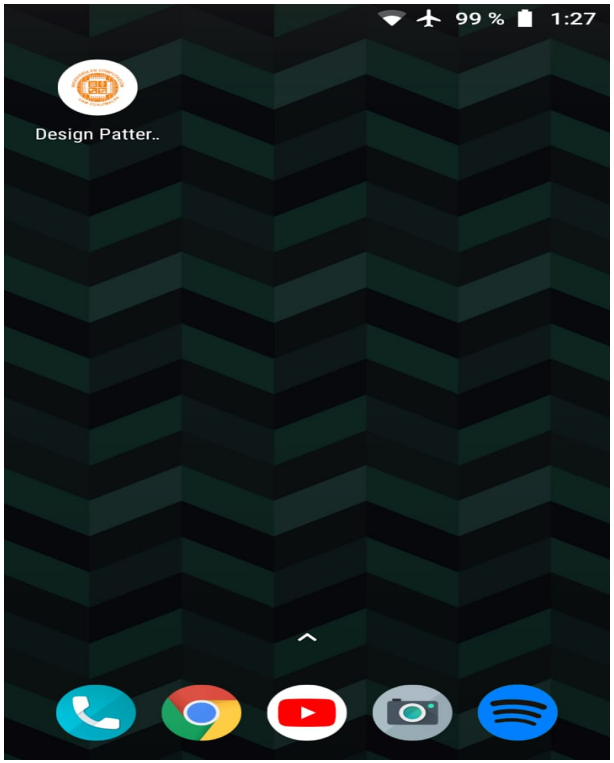

Figura 7.20: Aplicación para dispositivos moviles con sistema operativo Android

Después de que un usuario instala la versión de la plataforma para dispositivos moviles con sistema operativo Android, en su página principal se crea un icoco de acceso rápido, ver figura 7.20.

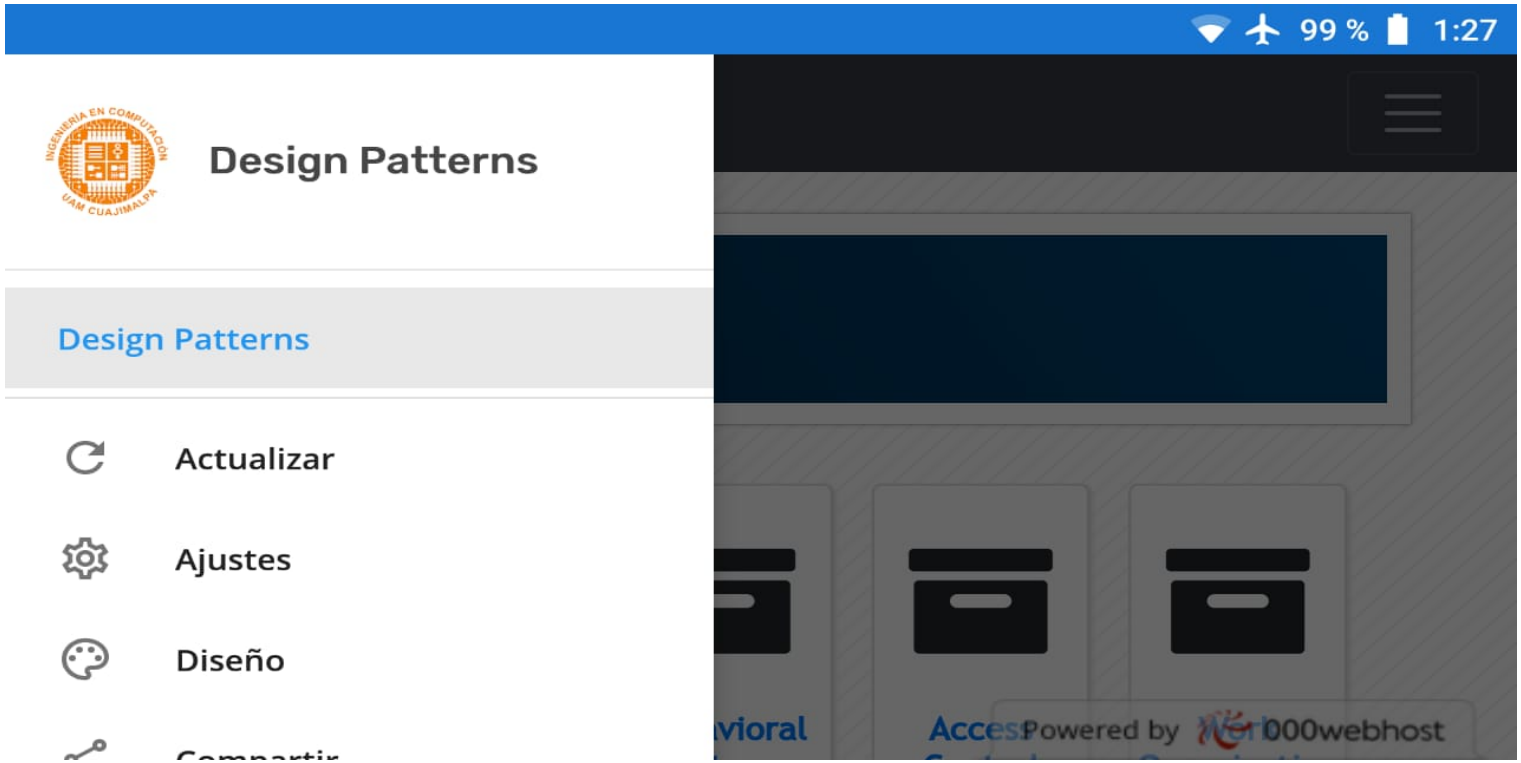

Figura 7.21: GUI Panel de configuración version móvil

En la versión móvil de la plataforma, se tiene la opción de un panel de configuración, ver figura 7.21, en dónde el usuario puede ajustar ciertas opciones de la plataforma de acuerdo a sus preferencias.

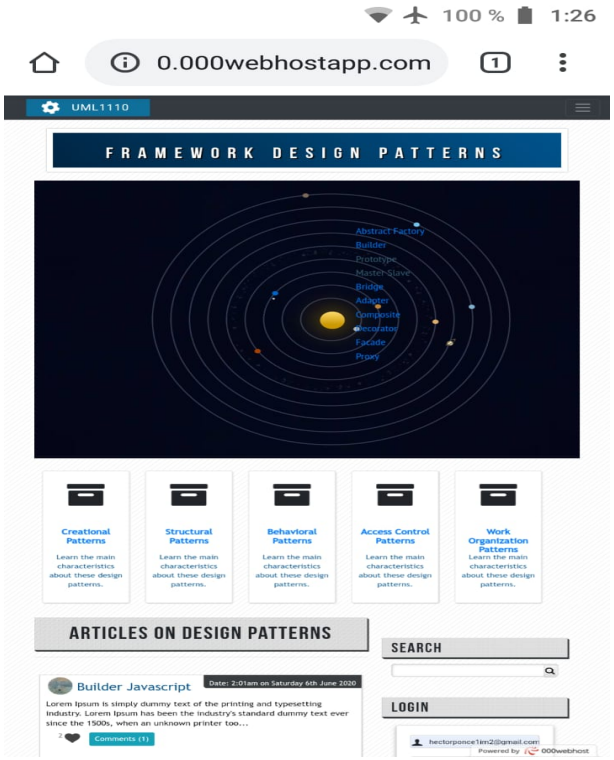

Figura 7.22: GUI Diseño responsive

En la figura 7.22, se puede apreciar que la plataforma de gestion de patrones de diseño está construida en su totalidad para cualquier dispositivo electrónico con acceso a internet, esto gracias a que su diseño es responsive y se puede ajustar a cualquier resolución de pantalla.

Capítulo 8

Conclusiones

8.0.1. Conclusiones

Los patrones de diseño constituyen una valiosa herramienta para la comprensión y entendimiento de la programación orientada a objetos, de ahí que tras veinte años de su invención, siguen siendo utilizados en el desarrollo de software y, más aún, marcando pautas para nuevas tendencias en programación.

Los patrones de diseño facilitan el mantenimiento de software y correctamente aplicados desde el diseño de un sistema, pueden mejorar altamente las prestaciones de escalabilidad, mantenimiento y seguridad del software.

Los patrones de diseño pueden optimizar los tiempos de desarrollo de una aplicación nueva, pues forman parte de librerías reutilizables.
En este trabajo el entendimiento de patrones de diseño se facilita mediante ejemplos prácticos orientados a la ingeniería de software, pues generalmente en la bibliografía se encuentran ejemplos que hacen uso de analogías, lo cual puede complicar el estudio de los patrones de diseño en la ingeniería de software.

El uso descuidado de los patrones de diseño puede desatar una serie de problemas que pueden provocar retrasos en el desarrollo de los proyectos, dificultad en la reutiliza-

ción de los módulos y problemas en la mantenibilidad; por lo cual no se debe incluir patrones de manera indiscriminada en un sistema de Software.

8.0.2. Recomendaciones

Se recomienda la inclusión de patrones de diseño en el desarrollo de software por necesidad y para solucionar problemas presentes en el sistema en desarrollo; pero no agregarlos sin mayor conocimiento de los mismos.

Es recomendable que cuando se desee incluir un patrón de diseño en un sistema, se debe, en lo posible, consultar con alguna persona con experiencia en el manejo de patrones de diseño, pues su uso apropiado demanda tener cierta experiencia al respecto.

También es recomendable que en el desarrollo de Software, se construya modelos lo más simples; además emplear los patrones más adecuados y fácilmente entendibles, aplicaciones de calidad.

Referencias

Alpízar, J. C. M., C. O. Rodriguez, and L. E. Bolaños. Patrones de diseño.

Autores, V. (2011). Reutilización de código.

Bahit, E. (2011). Poo y mvc en php. *El paradigma de la Programación*.

Campo, G. D. (2009). Patrones de diseño, refactorización y antipatrones. ventajas y desventajas de su utilización en el software orientado a objetos. *Cuadernos de Ingeniería* (4), 101–136.

Del Pilar, S. L. M. Arquitectura de software-is64-201002.

Gamma, E., R. Helm, and R. Johnson (1995). *Design Patterns*. United States of America: Addison-Wesley.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (2003). *Patrones de diseño*. Addison-Wesley.

García Peñalvo, F. J. et al. (1998). Patrones. de alexander a la tecnología de objetos.

Guerrero, C. A., J. M. Suárez, and L. E. Gutiérrez (2013). Patrones de diseño gof (the gang of four) en el contexto de procesos de desarrollo de aplicaciones orientadas a la web. *Información tecnológica*@(3), 103–114.

Leblang, D. B. and R. P. Chase Jr (1984). Computer-aided software engineering in a distributed workstation environment. *ACM Sigplan Notices*@(5), 104–112.

Peña, C. A. G. (2017). Líneas de productos software: Generando código a partir de modelos y patrones. *Scientia et technica*@(2), 175–182.

Rosas, M. D. S. (2020). Entorno de desarrollo patrones de diseño.

Tedeschi, N. (2015). ¿ qué es un patrón de diseño. *Recuperado a partir de https://msdn. microsoft. com/es-es/library/bb972240. aspx*.

Visconti, M. and H. Astudillo (2012). Fundamentos de ingeniería de software.

Yacoub, S. M. and H. H. Ammar (2004). *Pattern-oriented analysis and design: composing patterns to design software systems*. Addison-Wesley Professional.

Referencias

[1] Patrón de diseño(10 de Marzo de 2020). Recuperado de *https* : *//es.wikipedia.org/wiki/PatrondeDiseo* .