DOXA EDITION

Autor:
PATRICIO CELI V.

2023

Construyendo Software Innovador:

Implementación de patrones de diseño creacionales

Primera Edición

ITQ
WWW.ITQ.EDU.EC
INVESTIGACIÓN

ISBN: 978-9942-7156-2-3

INSTITUTO SUPERIOR TECNOLÓGICO QUITO
Excelencia en Educación Superior

CONSTRUYENDO SOFTWARE INNOVADOR
IMPLEMENTANDO PATRONES DE DISENO CREACIONALES

AUTOR:
PATRICIO CELI V.

PRIMERA EDICIÓN
2023

TRABAJO EN EDICIÓN:

DOXA EDITION

EDITOR INTERNO: DIEGO ORTEGA G.
EDITOR EXTERNO: DIEGO BASTIDAS L.

ISBN: 978-9942-7156-2-3

QUITO - ECUADOR

DEDICATORIA

A mi querido padre,

En cada página de este libro, he plasmado un pedazo de mi alma y mi mayor esfuerzo de trabajo, hoy sin duda comprendo las veces que decía porque no estas en casa o porque llegas tarde y es porque trabajaste más de 40 años por nosotros y aunque hoy en día estes un poco lejos, la enseñanza de un gran hombre de éxito la llevo conmigo.
Quiero dedicarte estas palabras como un tributo a tu amor incondicional, tu apoyo constante y tu sabiduría inagotable. Tú has sido mi inspiración y mi guía a lo largo de toda mi vida. Espero que este libro te muestre lo importante que eres en mi mundo y cuánto valoro tu influencia en mi camino.
Con cariño y admiración.

Con gratitud y respeto.
Patricio Celi Vivanco

AGRADECIMIENTO

En el viaje de la vida profesional, algunos encuentran maestros excepcionales y mentores invaluables. Hoy, quiero dedicar estas palabras a todos ustedes, que han dejado una huella imborrable en mi camino profesional en los que aparte de ser guías han sido amigos, padres y hasta jueces para nosotros, por todo eso y más gracias.

A Marthy, Madre, Amiga, Líder y sobre todo un gran ser humano, una de las personas más importantes para hoy ser lo que soy, quiero decirle gracias porque usted creyó en mí, porque usted vio oro en donde otros veían plomo, hoy quiero decirle que cada día me convierto en algo más valioso, sin duda usted es una de las personas de las que da gusto haber conocido en la vida, hoy quiero expresar mi agradecimiento por la oportunidad de aprender y crecer a su lado en el ámbito laboral. Su liderazgo, orientación y confianza en mí han sido fundamentales en ser un gran hombre y profesional con valores y fortalezas por todo muchas gracias.

Hoy, dedico mis logros y éxitos a todos ustedes, porque han sido piedras angulares en mi camino hacia el crecimiento y la realización personal. Gracias por su apoyo, sabiduría y amistad.

Con gratitud eterna.

Patricio Celi Vivanco

SOBRE EL AUTOR

Patricio Celi Vivanco destaca como profesional especializado en Análisis de Sistemas e Ingeniería en Informática, mostrando una pasión desbordante por el Desarrollo de Software. Su amor por la Tecnología y su naturaleza soñadora definen su enfoque hacia la creación y mejora continua. Sus habilidades abarcan diversos roles, siendo tanto docente como consultor, lector, tutor y capacitador especializado en temáticas relacionadas con el desarrollo de software. Como docente de Educación Superior, abraza la convicción de contribuir al inicio de aquellos que se aventuran en el mundo del Desarrollo, reconociendo que enseñar no solo beneficia a los aprendices, sino que también fortalece sus propios conocimientos.

INDICE

ÍNDICE DE FIGURAS

INSTITUTO SUPERIOR TECNOLÓGICO QUITO

Excelencia en Educación Superior

De Quito, somos EL Quito

INTRODUCCIÓN

La programación orientada a objetos (POO) es una técnica de programación que se utiliza ampliamente en la actualidad para crear software de alta calidad y fácilmente mantenible. En la POO, los objetos son los elementos fundamentales de un programa, y se utilizan para modelar el mundo real de una manera más precisa.

A medida que los programas de software se vuelven más complejos, se hace más difícil mantenerlos y extenderlos. Los patrones de diseño son soluciones recurrentes a problemas comunes en el desarrollo de software.

Los patrones de diseño son soluciones comunes a problemas recurrentes que se presentan en el desarrollo de software. Surgieron por primera vez en la década de 1990, cuando un grupo de programadores se reunió para documentar soluciones que habían encontrado a problemas específicos en sus proyectos.

En una línea de tiempo los patrones de diseño se han convertido en una herramienta clave para los programadores de todo el mundo para crear software de alta calidad y fácilmente mantenible.

Este libro se centra en una selección de los patrones de diseño más importantes y ampliamente utilizados en la programación orientada a objetos.

Los patrones de diseño que serán el caso de estudio de este trabajo de investigación son de creación o creacionales, aquí encontraremos una guía completa y práctica para los desarrolladores de software que buscan mejorar su comprensión y mantenimiento de aplicaciones; donde a través de ejemplos de código claros, concisos y explicaciones detalladas, los desarrolladores podrán aprender cómo aplicar efectivamente los patrones de diseño en su propio código.

CAPÍTULO I

PROGRAMACIÓN ORIENTADA A OBJETOS

1.1 Introducción a la programación orientada a objetos

El modelo de programación orientado a objetos denominado POO expresa un programa como un conjunto de objetos que colaboran entre sí para realizar tareas (Montero & Pérez, 2019).

Figura 1. *Ejes de la Programación orientada a Objetos*

Nota: Ilustración sobre los ejes más importantes de la programación orientada a objetos.

Los cuatro principios básicos de la programación orientada a objetos son:

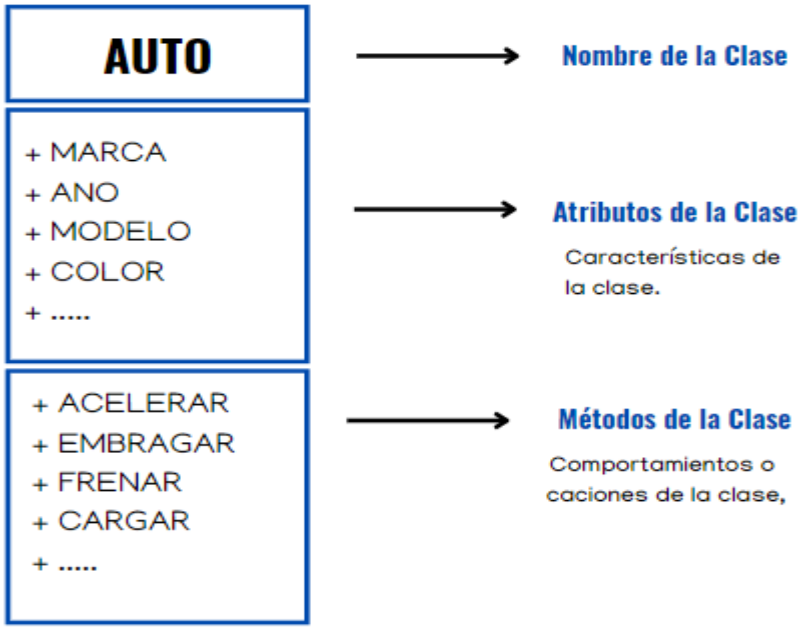
- **Abstracción:** modelar los atributos e interacciones pertinentes de las entidades como clases para definir una representación abstracta de un sistema.
- **Encapsulación:** ocultar el estado interno y la funcionalidad de un objeto y permitir solo el acceso a través de un conjunto público de funciones.
- **Herencia:** capacidad de crear nuevas abstracciones basadas en abstracciones existentes.
- **Polimorfismo:** capacidad de implementar propiedades o métodos heredados de maneras diferentes en varias abstracciones.(Barnes & Kölling, 2017)

1.2. OBJETO

Según (Montero & Pérez, 2019a) ,un objeto es una entidad, sujeto o cosa que se encuentra en situaciones o problemas de nuestro mundo real, formados por datos que representan la estructura del objeto y los métodos que implementan las operaciones que se debe realizar sobre los datos.

Por tal como podemos divisar en la siguiente figura la relación intrínseca entre el objeto y la clase.

Figura 2. *Clase y Objeto*



Nota: Descripción de una clase, atributos y métodos que contienen el desarrollo dentro de la programación

Sin embargo, podemos definir un objeto como una entidad con un estado y un comportamiento. El estado se refiere a las propiedades o atributos del objeto, mientras que el comportamiento se refiere a las acciones o métodos que el objeto puede realizar. Por ejemplo, supongamos que tienes una clase llamada "Auto". Un objeto de esta clase podría tener propiedades como "marca", "modelo", "color" y "año", que describen el estado del coche. También podría tener métodos como "acelerar", "embragar", "cargar" y "frenar", que definen el comportamiento del coche.

Cada objeto tiene una identidad única, lo que significa que es diferente a cualquier otro objeto de la misma clase. Por ejemplo, podrías tener dos objetos "Auto", uno con la marca "Toyota" y otro con la marca "Ford". Aunque ambos objetos pertenecen a la misma clase, tienen diferentes valores de propiedad y se comportan de manera diferente como se visualiza en la siguiente figura.

Figura 3. *Objetos y sus Clases*

Nota: En la figura se puede En la programación orientada a objetos, las clases son plantillas que definen atributos y comportamientos comunes para crear objetos, que son instancias específicas de esas clases. Las clases actúan como modelos para la creación de múltiples objetos con características individuales, facilitando la reutilización del código y la organización estructurada del programa.

La programación orientada a objetos se basa en la idea de que los objetos interactúan entre sí para realizar tareas más complejas. Los objetos pueden comunicarse mediante la llamada de métodos en otros objetos y la transmisión de mensajes entre ellos. Esto permite a los desarrolladores de software crear sistemas más grandes y complejos que son fáciles de entender y mantener.

1.3. HERENCIA

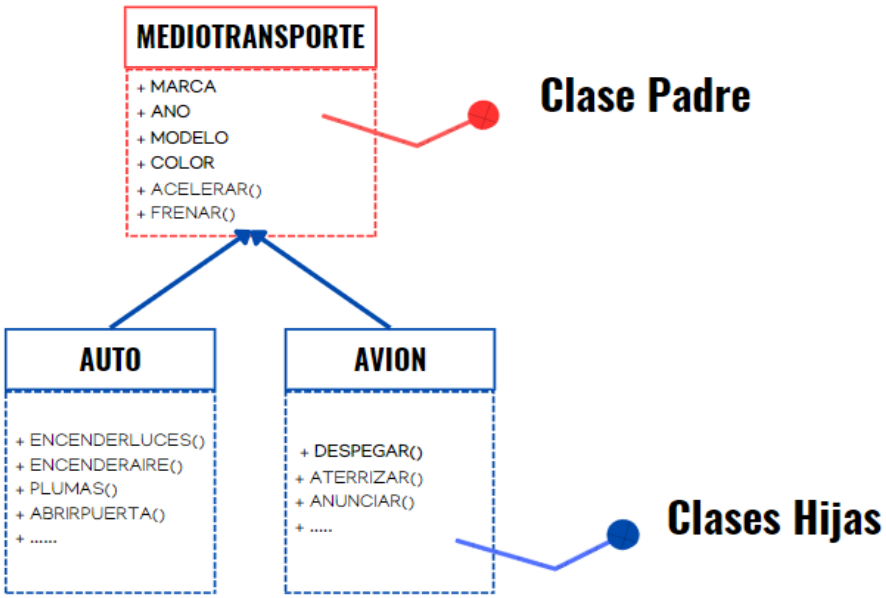
La herencia constituye uno de los procedimientos más fundamentales en el ámbito de la programación orientada a objetos. Para ilustrar esta premisa, consideremos el siguiente ejemplo: definimos con anterioridad una clase denominada "auto" a partir de la cual inferimos la existencia de múltiples instancias o mejor llamados objetos que comparten sus características esenciales. No obstante, entre los objetos hemos ejemplificado los

autos con Toyota y Ford, los cuales, si bien presentan particularidades singulares, permanecen englobados en la categoría más amplia de "auto".

Sin embargo, es posible establecer una clase principal denominada "MedioTransporte", que sirve como base para otras clases más específicas, como "Auto" y "Avión". Estas últimas representan tipos concretos de medios de transporte, cada uno con sus propias características y funcionalidades distintivas.

Al emplear una clase padre como punto de partida, se logra encapsular las propiedades y comportamientos comunes compartidos por ambos medios de transporte, como la capacidad de movimiento, el consumo de combustible y otros aspectos inherentes a los desplazamientos. Esta estructura jerárquica no solo fomenta una organización eficiente del código, sino que también refleja la relación conceptual entre los distintos medios de transporte en términos de su función fundamental, facilitando así el diseño y mantenimiento del software.

Figura 4. *Herencia*



Nota: En la figura se indica como a herencia en la programación orientada a objetos es un concepto clave donde una clase (llamada clase hija o subclase) puede heredar atributos y métodos de otra clase (llamada clase padre o superclase). Esta relación permite compartir características y comportamientos entre clases, fomentando la reutilización del código y estableciendo jerarquías que facilitan la extensión y modificación de las clases para adaptarse a necesidades específicas.

En el ejemplo ilustrado se define como superclase a la clase padre **MEDIOTRANSPORTE** y a las subclases **AUTO** y **AVION** como clases hijas. Las subclases heredan de su clase padre atributos y métodos, sin embargo, cuentan con sus propios métodos en los que se diferencian de sus otras clases hijo, por ejemplo existe un comportamiento **ABRIRPUERTA()** mismo que es un comportamiento propio de la clase **AUTO**, por su parte la clase **AVION** tiene un comportamiento como **DESPEGAR(),** esto nos indica que aunque tengan comportamientos diferentes siguen perteneciendo a la familia **MEDIOTRANSPORTE** y heredan atributos y comportamientos en común como **MARCA, MODELO** y comportamientos como **ACELERAR()** y **FRENAR ()**.

En conclusión, según (Montero & Pérez, 2019) la herencia constituye un valioso concepto en el modelamiento de la programación orientada a objetos debido a la organización de las acciones en diferentes clases, permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. El nuevo objeto puede definir nuevos atributos, nuevas operaciones, y redefinir las operaciones heredadas si se considera necesario.

CAPÍTULO II

PATRONES DE DISEÑO

2.1. ORIGEN Y DEFINICIÓN

Según (Blancarte Oscar, n.d.) Los patrones de diseño tienen su origen en la arquitectura, cuando en 1979 el Arquitecto Christopher Alexander publicó el libro Timeless Way of Building, en el cual hablaba de una serie de patrones para la construcción de edificios, comparando la arquitectura moderna con la antigua y cómo la gente había perdido la conexión con lo que se considera calidad.

Los patrones de diseño representan enfoques convencionales para abordar problemas frecuentes en el proceso de diseño de software. Son semejantes a esquemas preconcebidos que permiten ser adaptados para resolver desafíos de diseño recurrentes en tu código. No es posible seleccionar un patrón y simplemente insertarlo en el programa como si se tratara de funciones o librerías ya elaboradas.

Un patrón no constituye una sección específica de código, sino más bien un concepto general destinado a resolver una problemática particular. Es viable seguir los detalles inherentes al patrón e implementar una solución que se ajuste a las circunstancias propias de tu programa.

Por otra parte, según (Campo, 2009) Un Patrón de Diseño (design pattern) es una solución repetible a un problema recurrente en el diseño de software. Esta solución no es un diseño terminado que puede traducirse directamente a código, sino más bien una descripción sobre cómo resolver el problema, la cual puede ser utilizada en diversas situaciones.

Los patrones de diseño no son ideas complejas e incomprensibles, sino todo lo contrario. Son soluciones comunes para desafíos frecuentes en el diseño orientado a objetos. Cuando una solución se utiliza repetidamente en diversos proyectos, eventualmente alguien le asigna un nombre y la describe en profundidad, por tal este es el proceso mediante el cual se identifica un patrón y se crean nuevos a medida que avanzan las generaciones tecnológicas.

2.2 ESTRUCTURA

Un patrón de diseño típicamente sigue una estructura que consta de varios elementos clave:

- **Nombre del patrón**: Un nombre descriptivo que identifica el patrón y ayuda a comprender su propósito y función.
- **Introducción:** Una breve descripción del problema que el patrón aborda y por qué es relevante en el diseño de software.
- **Contexto:** Detalles sobre las condiciones y situaciones en las que el patrón puede ser aplicado de manera efectiva.
- **Problema:** Una explicación más detallada del problema de diseño específico que el patrón busca resolver.
- **Solución:** La descripción del enfoque o estrategia que el patrón propone para solucionar el problema. Esto puede incluir estructuras de clases, relaciones entre objetos y responsabilidades asignadas.
- **Participantes**: Las diferentes clases, objetos o componentes que participan en la implementación del patrón, junto con sus roles y responsabilidades.
- **Colaboraciones**: Cómo interactúan los diferentes participantes para llevar a cabo la solución propuesta por el patrón.
- **Consecuencias**: Los beneficios y posibles compromisos de utilizar el patrón. Esto podría incluir ventajas como flexibilidad, reusabilidad, pero también consideraciones como complejidad añadida.
- **Implementación**: Detalles técnicos sobre cómo implementar el patrón en el código, incluyendo posibles variantes o adaptaciones según el lenguaje de programación.
- **Ejemplos:** Casos de uso o ejemplos prácticos que ilustran cómo se aplica el patrón en situaciones reales.

Cada uno de estos elementos contribuye a proporcionar una comprensión completa del patrón de diseño, permitiendo que los desarrolladores lo apliquen de manera efectiva en sus proyectos de tal manera que para explicar cada uno de los patrones que serán nuestro punto de estudio se detallaran con los puntos definidos anteriormente.

2.3 CLASIFICACIÓN

Los patrones de diseño se agrupan en diversas categorías para facilitar su comprensión y aplicación en el proceso de desarrollo de software. Estas categorías permiten a los

diseñadores y desarrolladores identificar rápidamente el tipo de problema que están abordando y seleccionar el patrón más adecuado.

(Gamma et al., 2005) sostiene que el catálogo detallado en su libro consta de 23 patrones de diseño agrupados en tres grupos: Patrones Creacionales, Patrones Estructurales y Patrones de Comportamiento.

2.3.1 PATRONES CREACIONALES

Los patrones creacionales abarcan aquellos patrones de diseño que se centran en la generación y construcción de objetos. Su objetivo es dirigir el proceso de creación de objetos al implementar enfoques que evitan la creación directa.

2.3.2 PATRONES ESTRUCTURALES

Patrones Estructurales: Estos patrones abordan cómo las clases se conectan entre sí y se relacionan. Contribuyen a organizar nuestras clases de manera más sistemática, lo que a su vez fomenta la creación de componentes con mayor flexibilidad y capacidad de extensión.

2.3.3 PATRONES DE COMPORTAMIENTO

Los patrones de comportamiento están relacionados con los procedimientos ejecutados por objetos y la asignación de responsabilidades entre ellos. Incluyen también patrones que rigen la comunicación y la interacción entre objetos.

2.4 IMPORTANCIA

Es importante aprender sobre patrones de diseño porque son soluciones comprobadas por muchos desarrolladores en el mundo, existen profesionales que trabajan en el área sin conocer este tipo de herramienta, incluso trabajan utilizando algún patrón de diseño sin saberlo, por tal resulta cuan significativo al aprendizaje y aplicación de estas soluciones en proyectos de desarrollo y programación.

Uno de los pilares fundamentales del desarrollo de software se enfoca en el tiempo y costo de un proyecto, es decir si se realizar una aplicación independientemente del

lenguaje en que se realice, mientras menos tiempo y costo se enfoque en el desarrollo se podrá invertir ese dinero en otras áreas del proyecto como publicidad y marketing.

En base a esto hay que tener en cuenta un término interesante utilizado muchísimo en el área de desarrollo de proyectos de software y es el famoso reutilización de código que no es más que la utilización de un código existente para desarrollo de buenos proyectos.

Pero aquí nos cabe la siguiente pregunta: **¿Qué tan rentable es utilizar esta técnica y que tan eficiente es al abaratar los costos de un proyecto?**

Para responder esta pregunta se debe mencionar como antecedentes que la reutilización si ayuda a la reducción de costos de un proyecto sin embargo se debe considerar, que el esfuerzo de que un código existente funcione en una nueva aplicación requiere un esfuerzo extra sobre acoplamiento de clases, objetos, interfaces, métodos, funciones entre otros.

Por tal el uso patrones de diseño constituye uno de los métodos para incrementar la escalabilidad de los elementos de software y simplificar su reutilización. No obstante, en algunos casos, esto puede llevar consigo la consecuencia de añadir complejidad a dichos elementos con esto antes de realizar una práctica como la mencionada se debe considerar un análisis profundo antes de su implementación.

2.5. DISEÑO DE SOFTWARE

Cuando hablamos de diseño de software debemos centrarnos en él y sus principios por tal nos formulamos las siguientes interrogantes:

- **¿Qué es un buen diseño de software?**
- **¿Cómo medimos su calidad?**
- **¿Qué prácticas debemos llevar a cabo para lograrlo?**
- **¿Cómo podemos hacer nuestra arquitectura flexible, estable y fácil de comprender?**

Un buen diseño de software es aquel que cumple eficazmente con sus objetivos funcionales y no funcionales, al tiempo que es fácil de entender, mantener y modificar. Debe ser robusto, escalable y adaptarse a los cambios requeridos a lo largo del ciclo de

vida del software. Además, debe estar alineado con los requisitos del usuario y ser eficiente en términos de recursos.

La calidad del diseño de software se puede medir mediante varios criterios, como:

Cumplimiento de los requisitos: El diseño debe satisfacer todos los requisitos funcionales y no funcionales establecidos.

Mantenibilidad: Un diseño de calidad facilita la capacidad de mantener y modificar el software sin introducir errores o degradar su rendimiento.

Flexibilidad: Un diseño flexible permite la incorporación de cambios sin un impacto significativo en otras partes del sistema.

Eficiencia: El diseño debe utilizar los recursos de manera eficiente y evitar cuellos de botella.

Legibilidad: Un diseño legible es fácil de entender y documentar, lo que facilita la colaboración y la resolución de problemas.

Para lograr un buen diseño de software, se pueden implementar las siguientes prácticas:

- Análisis y comprensión exhaustiva de los requisitos del usuario.
- Uso de principios de diseño sólidos, como el modularidad, la cohesión y el bajo acoplamiento.
- Utilización de patrones de diseño reconocidos para abordar problemas comunes.
- Documentación adecuada del diseño, incluyendo diagramas y comentarios en el código.
- Revisión y retroalimentación por parte de otros miembros del equipo.
- Pruebas rigurosas para asegurar que el diseño funcione según lo previsto.

Los patrones de diseño son importantes, pero antes de entrar en los detalles de los patrones en sí, hablemos sobre el proceso de diseñar la arquitectura de software: aspectos a considerar y errores a evitar.

2.5.1 COSTOS Y TIEMPO

Los costos y el tiempo son dos de los factores más cruciales cuando se trata de desarrollar cualquier producto de software. La reducción del tiempo de desarrollo permite ingresar al mercado antes que la competencia, mientras que la disminución de los costos de desarrollo libera recursos adicionales para el marketing y amplía el alcance a posibles clientes.

2.5.2 REDUCCIÓN DE COSTOS

Una de las estrategias más comunes para reducir los costos de desarrollo es la reutilización de código. La lógica detrás de esta práctica es evidente: en lugar de construir algo repetidamente desde cero, ¿por qué no aprovechar el código existente en nuevos proyectos?

Sin embargo, esta idea aparentemente sencilla a menudo conlleva esfuerzos adicionales. Los acoplamientos fuertes entre componentes, las dependencias en clases concretas en lugar de interfaces y las operaciones entrelazadas en el código son obstáculos que reducen la flexibilidad del código y dificultan su reutilización en un nuevo contexto.

2.5.3 ENCAPSULACIÓN

El principio de encapsulación de lo que varía es una estrategia clave en el diseño de software que se enfoca en identificar y separar las partes de una aplicación que son propensas a cambios frecuentes. El objetivo principal de este principio es reducir al mínimo el impacto de los cambios en el sistema.

Si construimos una casa, sabes que con el tiempo habrá modificaciones en la decoración y el mobiliario interior. En lugar de construir las paredes de la casa de manera fija, puedes diseñar habitaciones separadas con paredes interiores removibles. De esta manera, cuando quieras cambiar la disposición de una habitación o su decoración, solo necesitas mover o reemplazar los elementos en esa habitación, sin afectar la estructura principal de la casa.

De manera similar, en el diseño de software, puedes identificar las partes que son propensas a cambios y aislarlas en módulos separados. Esto protege el resto del código de los efectos adversos de esos cambios. Al hacerlo, reducirás el tiempo necesario para

adaptar y probar las modificaciones, lo que te permitirá dedicar más tiempo a implementar nuevas características y mejorar la aplicación en lugar de ocuparte constantemente de ajustes imprevistos en partes del código que no estaban adecuadamente encapsuladas. En resumen, la encapsulación de lo que varía es una práctica inteligente que ayuda a mantener la estabilidad y la flexibilidad de un software a medida que evoluciona con el tiempo.

Por otra parte, la encapsulación nivel método también es un aspecto para considerar al desarrollador de diseño de software, podemos realizar una versión ejemplo de creación de una casa utilizando esta estrategia de desarrollo.

Supongamos que la creación de nuestra necesitamos que una parte de tu casa una habitación funcione como una biblioteca; dentro de esta biblioteca, tienes una colección de estantes llenos de libros.

La encapsulación a nivel del método sería como tener una biblioteca bien organizada. En lugar de tener libros esparcidos por toda la casa, los has colocado cuidadosamente en estantes y los has catalogado por género o autor. Esto significa que, si en el futuro deseas agregar, quitar o reorganizar libros, solo necesitas hacerlo dentro de la biblioteca, sin afectar la disposición de otros objetos en la casa.

De esta manera, la encapsulación a nivel del método ayuda a mantener el código organizado, facilita la identificación y modificación de partes específicas, y reduce la necesidad de realizar cambios extensos en toda la aplicación, al igual que la biblioteca bien organizada en la casa protege los libros de problemas y cambios en otras partes de la vivienda.

A continuación, tenemos el ejemplo realizado en C# para su comprensión y revisión.

```
1  using System;
2  using System.Collections.Generic;
3
4  class Casa
5  {
6      // Clase que representa la biblioteca dentro de la casa
7      public class Biblioteca
8      {
```

```
        private List<string> libros = new List<string>();

        // Método para agregar un libro a la biblioteca
        public void AgregarLibro(string libro)
        {
            libros.Add(libro);
            Console.WriteLine($"'{libro}' ha sido agregado a la biblioteca.");
        }

        // Método para listar todos los libros en la biblioteca
        public void ListarLibros()
        {
            Console.WriteLine("Libros en la biblioteca:");
            foreach (var libro in libros)
            {
                Console.WriteLine(libro);
            }
        }
    }

    public static void Main()
    {
        Casa miCasa = new Casa();
        Biblioteca miBiblioteca = new Biblioteca();

        // Agregar libros a la biblioteca
        miBiblioteca.AgregarLibro("Cien años de soledad");
        miBiblioteca.AgregarLibro("El principito");
        miBiblioteca.AgregarLibro("Don Quijote de la Mancha");

        // Listar libros en la biblioteca
        miBiblioteca.ListarLibros();
    }
}
```

Este ejemplo define una clase **Casa** que contiene una clase interna **Biblioteca**. La Biblioteca encapsula la funcionalidad relacionada con los libros, como agregar libros y listar los libros en la biblioteca. El método **AgregarLibro** permite agregar libros a la biblioteca, y el método **ListarLibros** muestra todos los libros en la biblioteca.

La encapsulación a nivel del método se logra al definir estos métodos específicos dentro de la clase Biblioteca. Esto permite que la funcionalidad relacionada con la biblioteca esté contenida en una ubicación específica y no afecte otras partes de la casa.

2.5.4 INTERFACES

La utilización de interfaces en sustitución de las implementaciones es un aspecto importante al momento de diseñar aplicaciones de cualquier índole, recordar que las aplicaciones deben tener calidad su diseño este se revela cuando puedes extenderlo sin dificultad y sin alterar el código existente.

Para comprender esto, consideremos un nuevo ejemplo relacionado con vehículos. Un vehículo que puede funcionar con cualquier tipo de combustible es más flexible que uno que solo puede usar gasolina. El primer vehículo puede adaptarse a diferentes tipos de combustibles, mientras que el segundo está limitado a uno específico.

A continuación, te dejamos un código de implementación de interfaz, para el ejemplo mencionado con anterioridad.

```
1  using System;
2
3  // Definimos una interfaz común para todos los tipos de documentos
4  public interface IDocumento
5  {
6      void Imprimir();
7  }
8
9  // Implementación de documentos específicos
10 public class DocumentoPDF : IDocumento
11 {
12     public void Imprimir()
13     {
14         Console.WriteLine("Imprimiendo documento PDF...");
15     }
16 }
17
18 public class DocumentoWord : IDocumento
19 {
```

```
20      public void Imprimir()
21      {
22          Console.WriteLine("Imprimiendo documento Word...");
23      }
24  }
25
26  public class DocumentoExcel : IDocumento
27  {
28      public void Imprimir()
29      {
30          Console.WriteLine("Imprimiendo documento Excel...");
31      }
32  }
33
34  // Clase que realiza la impresión de documentos genéricos
35  public class ImpresoraDocumentos
36  {
37      public void ImprimirDocumento(IDocumento documento)
38      {
39          documento.Imprimir();
40      }
41  }
42
43  class Program
44  {
45      static void Main()
46      {
47          ImpresoraDocumentos impresora = new ImpresoraDocumentos();
48
49          // Imprimir diferentes tipos de documentos usando la interfaz
50          IDocumento pdf = new DocumentoPDF();
51          IDocumento word = new DocumentoWord();
52          IDocumento excel = new DocumentoExcel();
53
54          impresora.ImprimirDocumento(pdf);
55          impresora.ImprimirDocumento(word);
56          impresora.ImprimirDocumento(excel);
57      }
58
```

2.5.5 LA DISYUNTIVA DE LA HERENCIA EN PROGRAMACIÓN

La herencia puede parecer la forma más directa de reutilizar código entre clases, con el tiempo puede revelar desafíos ocultos. Cuando tu programa crece y acumula una gran cantidad de clases, realizar cambios significativos se vuelve complicado.

Algunos de los problemas que surgen con la herencia incluyen la incapacidad de reducir la interfaz de la superclase, la necesidad de garantizar que las subclases no rompan el comportamiento heredado y la exposición de detalles internos de la superclase a las subclases, lo que puede generar una fuerte dependencia y acoplamiento entre ellas.

La utilización de la composición es un aspecto que nos ofrece una alternativa más flexible este significa que una clase puede contener instancias de otras clases y utilizar sus funcionalidades, sin heredar directamente de ellas.

Esta aproximación evita los problemas asociados con la herencia y fomenta una mayor flexibilidad y modularidad en el diseño del software. Además, la agregación, una variante de la composición, permite que los objetos contengan referencias a otros objetos sin gestionar su ciclo de vida, lo que ofrece una mayor libertad y versatilidad en el diseño.

Ahora tanto la utilización de herencia o no dentro de un diseño de programación es una decisión un tanto dependiente del desarrollador, del entorno y del alcance de los proyectos, por tal es imprescindible saber estos aspectos antes de comenzar un nuevo proyecto.

CAPÍTULO 3
PRINCIPIOS SOLID

3.1. Introducción a uso de Principios Solid

Si hablamos de diseño y desarrollo de aplicaciones, Principios SOLID son unas palabras que debes conocer como uno de los fundamentos de la arquitectura y desarrollo de software.(*SOLID: Los 5 Principios Que Te Ayudarán a Desarrollar Software de Calidad*, n.d.)

Uno de los aspectos principales a considerar en el diseño de aplicaciones en desarrollo de software son los principios SOLID mismos que se denominan un conjunto de cinco principios de diseño de software que se consideran fundamentales para crear sistemas de software más comprensibles, mantenibles y flexibles.

Para (Alfred & Lázaro, 2019) en su trabajo de desarrollo de contabilidad en. NetCore señala que SOLID es un acrónimo inventado por Robert C. Martin para establecer los cinco principios básicos del diseño y la programación orientada a objetos. Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento.

3.1.1 Principio de Responsabilidad Única (SRP - Single Responsibility Principle):

Este principio establece que una clase debe tener una única razón para cambiar. En otras palabras, una clase debe tener una sola responsabilidad y no debe estar sobrecargada con múltiples tareas. Cuando una clase tiene una única responsabilidad, es más fácil de entender, mantener y modificar.

Para (*View of State of the Art Research in: Clean Architecture and SOLID Principles*, n.d.) en su artículo científico señala que el e principio de responsabilidad única define que una clase debe tener una única responsabilidad hacia un actor, es decir, debe contener funciones específicas y limitadas que satisfagan la demanda generada por la funcionalidad.

Aunque (Z et al., 2021) en su artículo de revista de Utilización de Arquitecturas Limpias para Trabajo y buenas Prácticas en la Construcción de aplicaciones Java indica que SOLID funciona principalmente para combatir arquitecturas de software difíciles de mantener y mejorar el desarrollo de software, y facilitar su lectura para los usuarios de trabajo programador.

Por tal y para mayor comprensión de los principios SOLID a continuación, tenemos un ejemplo de aplicación de este ejemplo en C#, para su revisión.

```
1  class GestorDePedidos
2  {
3      public void RegistrarPedido(Pedido pedido)
4      {
5          // Código para registrar el pedido en la base de datos.
6      }
7
8      public double CalcularTotal(Pedido pedido)
9      {
10         // Código para calcular el total del pedido.
11     }
12
13     public void GenerarInformeDeVentasMensual()
14     {
15         // Código para generar el informe de ventas mensual.
16     }
17 }
```

En este caso, la clase **GestorDePedidos** tiene múltiples responsabilidades: gestionar pedidos, realizar cálculos y generar informes. Esto viola el principio SRP, ya que la clase tiene más de una razón para cambiar. Si una de estas responsabilidades cambia, podría afectar negativamente a las otras por tanto aplicando este principio el SRP dividiríamos las responsabilidades en clases separadas, cada una con una única responsabilidad.

```
1  class GestorDePedidos
2  {
3      public void RegistrarPedido(Pedido pedido)
4      {
5          // Código para registrar el pedido en la base de datos.
6      }
```

```
}

class CalculadoraDeTotal
{
    public double CalcularTotal(Pedido pedido)
    {
        // Código para calcular el total del pedido.
    }
}

class GeneradorDeInformes
{
    public void GenerarInformeDeVentasMensual()
    {
        // Código para generar el informe de ventas mensual.
    }
}
```

En este nuevo diseño, cada clase tiene una sola responsabilidad. La clase **GestorDePedidos** se encarga de registrar pedidos, la clase **CalculadoraDeTotal** calcula el total del pedido y la clase **GeneradorDeInformes** se encarga de generar informes de ventas. Esto hace que el código sea más modular y fácil de mantener, ya que cada clase tiene una única razón para cambiar y las modificaciones en una no afectan a las otras.

3.1.2 Principio de Abierto/Cerrado (OCP - Open/Closed Principle):

El principio OCP sugiere que las entidades de software, como clases y módulos, deben estar abiertas para la extensión, pero cerradas para la modificación. Esto significa que debes poder agregar nuevas funcionalidades a un sistema sin alterar el código existente. Esto se logra a menudo mediante el uso de interfaces y la herencia.

A continuación, tenemos un ejemplo de aplicación de este ejemplo en C#, para su revisión.

```
class ProcesadorDePagos
{
    public void ProcesarPagoConVisa(Pago pago)
    {
```

```
        // Código para procesar pagos con Visa.
    }

    public void ProcesarPagoConMasterCard(Pago pago)
    {
        // Código para procesar pagos con MasterCard.
    }
}
```

El problema aquí es que, si deseas agregar soporte para otro método de pago, como PayPal, necesitas modificar la clase **ProcesadorDePagos**, lo que va en contra del principio OCP.

Por tal para aplicar el principio OCP, puedes utilizar una interfaz común para los procesadores de pago y crear clases separadas para cada método de pago sin modificar la clase existente.

```
// Interfaz común para los procesadores de pago
interface IProcesadorDePago
{
    void ProcesarPago(Pago pago);
}

// Implementaciones de procesadores de pago específicos
class ProcesadorVisa : IProcesadorDePago
{
    public void ProcesarPago(Pago pago)
    {
        // Código para procesar pagos con Visa.
    }
}

class ProcesadorMasterCard : IProcesadorDePago
{
    public void ProcesarPago(Pago pago)
    {
        // Código para procesar pagos con MasterCard.
    }
}
```

```
23
24 class ProcesadorPayPal : IProcesadorDePago
25 {
26     public void ProcesarPago(Pago pago)
27     {
28         // Código para procesar pagos con PayPal.
29     }
30 }
```

En este nuevo diseño, hemos creado una interfaz **IProcesadorDePago** que define un método **ProcesarPago**. Luego, hemos implementado esta interfaz en clases separadas para cada método de pago específico.

Ahora, puedes agregar nuevos procesadores de pago (como PayPal) sin modificar la clase existente **ProcesadorDePagos**. Esto cumple con el principio OCP, ya que la clase está abierta para la extensión (agregar nuevos procesadores de pago) pero cerrada para la modificación (no necesitas cambiar la clase existente para agregar funcionalidad).

3.1.3 Principio de Sustitución de Liskov (LSP - Liskov Substitution Principle).

Este principio establece que los objetos de una subclase deben ser capaces de sustituir sin problemas a los objetos de la clase base sin afectar la corrección del programa. En otras palabras, una subclase debe ser una extensión válida de su clase base y debe comportarse de manera coherente con la clase base.

A continuación, tenemos un ejemplo de aplicación de este ejemplo en C#, para su revisión.

El ejercicio nos presenta una jerarquía de clases que representa diferentes tipos de aves; en la clase base **Ave**, definimos un método **Volar** que permite que todas las aves vuelen.

```
1  class Ave
2  {
3      public void Volar()
4      {
5          Console.WriteLine("El ave está volando.");
6      }
7  }
8
```

```
 9 class Pinguino : Ave
10 {
11     public new void Volar()
12     {
13         Console.WriteLine("Los pingüinos no pueden volar.");
14     }
15 }
```

En este ejemplo, tenemos una clase **Pinguino** que hereda de Ave. Sin embargo, la implementación del método Volar en la clase **Pinguino** es incorrecta, ya que los pingüinos no pueden volar.

Esto viola el principio de sustitución de Liskov porque no podemos sustituir un objeto de la clase base Ave por un objeto de la clase derivada **Pinguino** sin cambiar el comportamiento esperado.

Por tal para asegurarnos de cumplir con el principio LSP, debemos asegurarnos de que la subclase se comporte de manera coherente con la clase base. Podemos hacer esto refactorizando nuestro código de la siguiente manera:

```
 1 class Ave
 2 {
 3     public virtual void Volar()
 4     {
 5         Console.WriteLine("El ave está volando.");
 6     }
 7 }
 8
 9 class Pinguino : Ave
10 {
11     public override void Volar()
12     {
13         Console.WriteLine("Los pingüinos no pueden volar.");
14     }
15 }
```

Ahora en este nuevo ejemplo hemos marcado el método Volar en la clase base como virtual y hemos anulado ese método en la clase Pinguino utilizando override.

Ahora, el principio LSP se cumple porque los objetos de la clase derivada Pinguino se comportan de manera coherente con los objetos de la clase base Ave. Podemos sustituir un objeto de Ave por un objeto de Pinguino sin sorpresas inesperadas en el comportamiento.

3.1.4 Principio de Segregación de la Interfaz (ISP - Interface Segregation Principle).

Este principio establece que una interfaz no debe forzar a las clases que la implementan a depender de métodos que no utilizan. En otras palabras, las interfaces deben ser específicas y no deben incluir métodos que no sean relevantes para todas las clases que las implementan.

A continuación, tenemos un ejemplo de aplicación de este ejemplo en C#, para su revisión.

Supongamos que estamos diseñando un sistema de administración de empleados y tenemos una interfaz llamada **IEmpleado** que contiene métodos relacionados con la gestión de empleados, como **Registrar**, **Actualizar**, **Eliminar** y **CalcularSalario**.

```
1 interface IEmpleado
2 {
3     void Registrar();
4     void Actualizar();
5     void Eliminar();
6     double CalcularSalario();
7 }
```

Luego, creamos dos clases que implementan esta interfaz: **EmpleadoTiempoCompleto** y **EmpleadoTemporal**. El problema es que la clase **EmpleadoTemporal** no necesita el método **CalcularSalario**, ya que los empleados temporales no tienen un salario regular.

```
1  class EmpleadoTiempoCompleto : IEmpleado
 2 {
 3     public void Registrar() { /* Implementación */ }
 4     public void Actualizar() { /* Implementación */ }
```

```
    public void Eliminar() { /* Implementación */ }
    public double CalcularSalario() { /* Implementación */ }
}

class EmpleadoTemporal : IEmpleado
{
    public void Registrar() { /* Implementación */ }
    public void Actualizar() { /* Implementación */ }
    public void Eliminar() { /* Implementación */ }
    public double CalcularSalario() { /* Implementación, pero no es
relevante */ }
  }
```

En este ejemplo, la interfaz **IEmpleado** fuerza a la clase **EmpleadoTemporal** a implementar el método **CalcularSalario**, a pesar de que no tiene sentido para este tipo de empleado.

Por otra parte, para aplicar el ISP, dividiríamos la interfaz **IEmpleado** en interfaces más pequeñas y específicas, de modo que cada clase solo implemente las interfaces que sean relevantes para ella.

```
interface IRegistroEmpleado
{
    void Registrar();
    void Actualizar();
    void Eliminar();
}

interface ICalculoSalario
{
    double CalcularSalario();
}

class EmpleadoTiempoCompleto : IRegistroEmpleado, ICalculoSalario
{
    public void Registrar() { /* Implementación */ }
    public void Actualizar() { /* Implementación */ }
    public void Eliminar() { /* Implementación */ }
    public double CalcularSalario() { /* Implementación */ }
```

```
19 }
20
21 class EmpleadoTemporal : IRegistroEmpleado
22 {
23     public void Registrar() { /* Implementación */ }
24     public void Actualizar() { /* Implementación */ }
25     public void Eliminar() { /* Implementación */ }
26 }
```

Después de haber aplicado el principio ISP donde hemos dividido la interfaz en dos: **IRegistroEmpleado** e **ICalculoSalario**.

Cada clase implementa solo las interfaces que son relevantes para su tipo de empleado, evitando así la dependencia de métodos no utilizados. Esto cumple con el principio ISP y hace que el diseño sea más coherente y mantenible.

3.1.5 Principio de Inversión de Dependencia (DIP - Dependency Inversion Principle)

Este principio establece que las clases de alto nivel no deben depender de las clases de bajo nivel, sino de abstracciones. Además, las abstracciones no deben depender de los detalles, los detalles deben depender de las abstracciones.

A continuación, tenemos un ejemplo de aplicación de este ejemplo en C#, para su revisión.

Creamos un sistema de informes financieros que depende directamente de una clase concreta llamada **BaseDeDatosFinanciera** para obtener datos financieros.

```
1  class BaseDeDatosFinanciera
2 {
3     public string ObtenerDatosFinancieros()
4     {
5         // Código para obtener datos financieros de la base de datos.
6         return "Datos financieros obtenidos de la base de datos.";
7     }
8 }
9
10 class GeneradorDeInformes
```

```
11 {
12     private BaseDeDatosFinanciera baseDeDatos;
13 
14     public GeneradorDeInformes()
15     {
16         baseDeDatos = new BaseDeDatosFinanciera();
17     }
18 
19     public string GenerarInforme()
20     {
21         string datos = baseDeDatos.ObtenerDatosFinancieros();
22         // Código para generar un informe con los datos financieros.
23         return "Informe financiero generado: " + datos;
24     }
25 
```

En el código anterior, la clase **GeneradorDeInformes** está fuertemente relacionada a la clase concreta **BaseDeDatosFinanciera**.

Esto significa que, si deseamos cambiar la fuente de datos financiero o agregar otra fuente, debemos modificar directamente la clase **GeneradorDeInformes**, lo que va en contra del principio DIP por lo que para alinearnos al principio creamos una interfaz llamada **IFuenteDeDatos** que define un contrato (un conjunto de métodos) para obtener datos financieros.

Esta interfaz actúa como una abstracción que todas las fuentes de datos financieros deben seguir.

Implementación de **IFuenteDeDatos** en **BaseDeDatosFinanciera**: Luego, hacemos que la clase concreta **BaseDeDatosFinanciera** implemente la interfaz **IFuenteDeDatos**. Esto significa que **BaseDeDatosFinanciera** proporciona una implementación concreta de los métodos definidos en la interfaz, en este caso, el método **ObtenerDatosFinancieros**. Esta implementación representa cómo se obtienen los datos financieros de una base de datos.

Dependencia de la interfaz en **GeneradorDeInformes** en lugar de que la clase **GeneradorDeInformes** dependa directamente de la clase

BaseDeDatosFinanciera, la modificamos para que dependa de la interfaz **IFuenteDeDatos**.

Esto se logra mediante la inyección de dependencia, donde **GeneradorDeInformes** recibe una instancia de **IFuenteDeDatos** en su constructor y la ventaja de esta implementación es que **GeneradorDeInformes** ahora depende de una abstracción (**IFuenteDeDatos**) en lugar de una implementación concreta (**BaseDeDatosFinanciera**).

Esto hace que **GeneradorDeInformes** sea independiente de los detalles internos de cómo se obtienen los datos financieros. Si en el futuro deseamos cambiar la fuente de datos o agregar una nueva fuente, simplemente necesitamos proporcionar una nueva clase que implemente **IFuenteDeDatos**, y **GeneradorDeInformes** seguirá funcionando sin modificaciones.

Aplicado el ejemplo a lenguaje C# quedaría de la siguiente manera:

```
1  interface IFuenteDeDatos
2  {
3      string ObtenerDatosFinancieros();
4  }
5
6  class BaseDeDatosFinanciera : IFuenteDeDatos
7  {
8      public string ObtenerDatosFinancieros()
9      {
10         // Código para obtener datos financieros de la base de datos.
11         return "Datos financieros obtenidos de la base de datos.";
12     }
13 }
14
15 class GeneradorDeInformes
16 {
17     private IFuenteDeDatos fuenteDeDatos;
18
19     public GeneradorDeInformes(IFuenteDeDatos fuente)
20     {
21         fuenteDeDatos = fuente;
22     }
```

```
23
24      public string GenerarInforme()
25      {
26          string datos = fuenteDeDatos.ObtenerDatosFinancieros();
27          // Código para generar un informe con los datos financieros.
28          return "Informe financiero generado: " + datos;
29      }
30  }
```

Dado que hemos aplicado el principio **GeneradorDeInformes** depende de la interfaz **IFuenteDeDatos**, lo que permite que podamos cambiar la fuente de datos o agregar nuevas fuentes sin modificar **GeneradorDeInformes** esto cumple con el principio DIP, ya que las clases de alto nivel dependen de abstracciones (interfaces) en lugar de detalles de implementación concretos.

Los principios SOLID, que abarcan desde el Principio de Responsabilidad Única hasta el Principio de Inversión de Dependencia, desempeñan un papel fundamental en el diseño de software sólido y mantenible. Estos principios proporcionan directrices para estructurar el código de manera que sea fácil de entender, extender y mantener a lo largo del tiempo.

Cuando se trata de patrones de diseño, como el patrón Singleton, el patrón Factory, o el patrón Observer, estos principios actúan como cimientos sólidos sobre los cuales se construyen los patrones. Los patrones de diseño son soluciones probadas para problemas comunes en el diseño de software, y su conocimiento y aplicación se benefician enormemente de seguir los principios SOLID.

Por ejemplo, al aplicar el Principio de Responsabilidad Única, se garantiza que las clases tengan una única razón para cambiar, lo que hace que sea más fácil identificar dónde aplicar un patrón; como Singleton para controlar la creación de una única instancia de una clase.

De manera similar, el Principio de Inversión de Dependencia respalda la implementación de patrones como el patrón Factory Method, permitiendo que las clases de alto nivel dependan de abstracciones en lugar de implementaciones concretas, lo que facilita la introducción de nuevas variantes de productos en un sistema sin afectar al código existente.

A continuación, vamos a estudiar los principales patrones de diseño creacionales que utilizamos dentro de los proyectos se software, donde comprenderemos su origen, su problemática y ejemplificaremos los mismos de tal manera que el lector pueda aplicarlos a sus jornadas de trabajo.

CAPÍTULO 4

PATRONES DE DISENO CREACIONALES

4.1. CATÁLOGO DE PATRONES DE DISENO CREACIONALES

Los patrones creacionales se encargan de proporcionar varias maneras de crear objetos dentro de un proyecto de desarrollo de software, lo más importante de este tipo de patrones es que su aplicación aumenta la flexibilidad y la reutilización de código. Estos patrones abordan el proceso de instanciación y configuración de objetos, ofreciendo métodos estructurados para manejar diversas situaciones y requisitos.

Para (*Patrones Creacionales*, n.d.) el gurú de los patrones de diseño en la actualidad define a los patrones creacionales proporcionan varios mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código existente.

Por su parte (Blancarte, n.d.)en su libro Introducción a los Patrones de Diseño menciona que los patrones de diseño creacionales son patrones de diseño relacionados con la creación o construcción de objetos. Estos patrones intentan controlar la forma en que los objetos son creados implementando mecanismos que eviten la creación directa de objetos.

Sin embargo, dentro del paradigma de los patrones de diseño creacionales existen varios tipos que en este trabajo de investigación serán motivo de estudio y se detallan a continuación.

4.1.1 Factory Method

Figura 5. *Factory Method*

Nota: Ilustración creada a base de Inteligencia Artificial sobre Factory Method, elaborada por el Autor.

Introducción

El patrón de diseño Factory Method, o Método de Fábrica en español, es una técnica fundamental en la programación orientada a objetos que se utiliza para crear objetos sin especificar su clase concreta en el código cliente. En lugar de instanciar directamente una clase, el Factory Method proporciona un método que se encarga de crear y devolver objetos adecuados según ciertos parámetros o condiciones. Este enfoque promueve la flexibilidad y la extensibilidad del código, ya que permite cambiar la implementación subyacente de una clase sin afectar al código que la utiliza.

Para (Blancarte, n.d.) Factory Method es un patrón que se centra en la creación de una clase fábrica la cual tiene métodos que nos permitan crear objetos de un subtipo determinado.

Factory Method consta de dos componentes principales: la clase abstracta o interfaz de la fábrica y las clases concretas de fábrica que implementan esta interfaz. El cliente interactúa con la interfaz de la fábrica para crear objetos, sin necesidad de conocer las implementaciones concretas de dicha fábrica.

Este patrón se usa ampliamente en la creación de objetos en bibliotecas y frameworks, permitiendo que los desarrolladores adapten el comportamiento de las clases sin modificar el código existente.

Problema

Imagina que estás desarrollando un sistema para una pizzería y necesitas crear objetos de diferentes tipos de pizzas (por ejemplo, pizza de pepperoni, pizza vegetariana, pizza hawaiana, etc.). Sin embargo, no sabes de antemano qué tipo de pizza se pedirá en cada momento. Necesitas una forma de crear objetos de pizza de manera flexible y sin acoplar tu código a clases concretas de pizza.

Solución

El patrón Factory Method actúa como una solución elegante y flexible para el problema de la creación de objetos, permitiendo que el código del cliente no esté acoplado a las clases concretas de los objetos que crea, a continuación, lo explicamos de manera más específica.

Desacopla la creación de objetos de su uso:

En una aplicación típica, la creación de objetos generalmente se realiza dentro de una clase que los usa. Esto acopla el código de usuario a las clases concretas de objetos que se crean, lo que hace que el código sea menos flexible y más difícil de mantener.

El Factory Method desacopla esta creación, moviéndola a una clase separada (el Creador), lo que permite que el código de usuario se comunique con el Creador a través de una interfaz en lugar de una clase concreta. Esto hace que el código sea más modular y fácil de extender.

Define una interfaz común para la creación de objetos.

El Factory Method define una interfaz o método abstracto (generalmente llamado "Crear" o "FactoryMethod") que todas las subclases de Creador deben implementar. Esta interfaz proporciona una forma consistente de crear objetos.
Cada subclase de Creador puede decidir qué tipo de objeto concreto crear y cómo inicializarlo, pero el código de cliente interactúa con estas subclases a través de la interfaz común, sin preocuparse por los detalles de la implementación concreta.

Permite la flexibilidad en la elección de implementaciones.

Debido a que cada subclase de Creador puede proporcionar su propia implementación de FactoryMethod, el código del cliente puede elegir qué tipo de objeto crear en tiempo de ejecución. Esto permite adaptar dinámicamente la creación de objetos según las necesidades del cliente.

Por ejemplo, en el contexto de una pizzería, si un cliente solicita una "**PizzaPepperoni**", el FactoryMethod en la subclase **PizzeriaPepperoni** creará una instancia de **PizzaPepperoni**. Si otro cliente solicita una "**PizzaVegetariana**", el FactoryMethod en la subclase **PizzeriaVegetariana** creará una instancia de **PizzaVegetariana**. El código del cliente no necesita conocer los detalles de implementación.

Facilita la expansión y mantenimiento del código:

A medida que se agregan nuevos tipos de productos (nuevos sabores de pizza en nuestro ejemplo), simplemente se crea una nueva subclase de Creador y se implementa su FactoryMethod. Esto no afecta al código existente del cliente, que sigue utilizando la misma interfaz común.

Esto hace que la base de código sea más fácil de mantener y extender a medida que evolucionan los requisitos.

Estructura

El patrón Factory Method consta de las siguientes partes clave:

- **Product (Producto):** Define la interfaz de los objetos que el Factory Method crea.
- **ConcreteProduct (Producto Concreto):** Son las implementaciones concretas de la interfaz del producto.
- **Creator (Creador):** Declara el método de fábrica abstracto que las subclases deben implementar. Puede contener código que utilice los objetos de producto, independientemente de sus clases concretas.
- **ConcreteCreator (Creador Concreto):** Implementa el método de fábrica abstracto para crear objetos concretos. Puede devolver diferentes tipos de objetos de producto.

Pseudocódigo

Para ejemplificar el patrón de diseño Factory Method, se ha generado un seudocódigo con la resolución del problema, te lo muestro a continuación.

```
// Paso 1: Define la interfaz del producto (Pizza)
Interfaz IPizza
    Método Preparar()
    Método Hornear()
    Método Cortar()
    Método Empacar()
Fin Interfaz

// Paso 2: Implementa las clases concretas de productos (por ejemplo,
PizzaPepperoni, PizzaVegetariana, etc.)
Clase PizzaPepperoni implementa IPizza
    Método Preparar()
        Escribir "Preparando pizza de pepperoni"
    Fin Método

    Método Hornear()
        Escribir "Horneando pizza de pepperoni"
    Fin Método

    Método Cortar()
        Escribir "Cortando pizza de pepperoni"
    Fin Método

    Método Empacar()
        Escribir "Empacando pizza de pepperoni"
    Fin Método
Fin Clase

Clase PizzaVegetariana implementa IPizza
    Método Preparar()
        Escribir "Preparando pizza vegetariana"
    Fin Método

    Método Hornear()
        Escribir "Horneando pizza vegetariana"
```

```
36      Fin Método
37
38      Método Cortar()
39          Escribir "Cortando pizza vegetariana"
40      Fin Método
41
42      Método Empacar()
43          Escribir "Empacando pizza vegetariana"
44      Fin Método
45  Fin Clase
46
47  // Paso 3: Define la interfaz del creador (Pizzeria)
48  Interfaz IPizzeria
49      Método CrearPizza() retorna IPizza
50  Fin Interfaz
51
52  // Paso 4: Implementa las clases concretas de creadores (por ejemplo,
53  PizzeriaPepperoni, PizzeriaVegetariana, etc.)
54  Clase PizzeriaPepperoni implementa IPizzeria
55      Método CrearPizza() retorna IPizza
56          Retorna nueva PizzaPepperoni()
57      Fin Método
58  Fin Clase
59
60  Clase PizzeriaVegetariana implementa IPizzeria
61      Método CrearPizza() retorna IPizza
62          Retorna nueva PizzaVegetariana()
63      Fin Método
64  Fin Clase
65
66  // Cliente
67  Función Main()
68      IPizzeria pizzeria = nueva PizzeriaPepperoni()
69      IPizza pizza = pizzeria.CrearPizza()
70
71      pizza.Preparar()
72      pizza.Hornear()
73      pizza.Cortar()
74      pizza.Empacar()
75  Fin Función
```

Implementación

Para realizar la implementación del patrón de Diseño Factory Method basándonos en el modelo de pseudocodigo se puede realizar en cualquier lenguaje de programación, en este caso lo implementamos en C# detallados a continuación.

```
1  using System;
2
3  // Paso 1: Define la interfaz del producto (Pizza)
4  public interface IPizza
5  {
6      void Preparar();
7      void Hornear();
8      void Cortar();
9      void Empacar();
10 }
11
12 // Paso 2: Implementa las clases concretas de productos (por ejemplo,
13 PizzaPepperoni, PizzaVegetariana, etc.)
14 public class PizzaPepperoni : IPizza
15 {
16     public void Preparar()
17     {
18         Console.WriteLine("Preparando pizza de pepperoni");
19     }
20
21     public void Hornear()
22     {
23         Console.WriteLine("Horneando pizza de pepperoni");
24     }
25
26     public void Cortar()
27     {
28         Console.WriteLine("Cortando pizza de pepperoni");
29     }
30
31     public void Empacar()
32     {
33         Console.WriteLine("Empacando pizza de pepperoni");
34     }
```

```
35 }
36
37 public class PizzaVegetariana : IPizza
38 {
39     public void Preparar()
40     {
41         Console.WriteLine("Preparando pizza vegetariana");
42     }
43
44     public void Hornear()
45     {
46         Console.WriteLine("Horneando pizza vegetariana");
47     }
48
49     public void Cortar()
50     {
51         Console.WriteLine("Cortando pizza vegetariana");
52     }
53
54     public void Empacar()
55     {
56         Console.WriteLine("Empacando pizza vegetariana");
57     }
58 }
59
60 // Paso 3: Define la interfaz del creador (Pizzeria)
61 public interface IPizzeria
62 {
63     IPizza CrearPizza();
64 }
65
66 // Paso 4: Implementa las clases concretas de creadores (por ejemplo,
67 PizzeriaPepperoni, PizzeriaVegetariana, etc.)
68 public class PizzeriaPepperoni : IPizzeria
69 {
70     public IPizza CrearPizza()
71     {
72         return new PizzaPepperoni();
73     }
74 }
```

```
75
76 public class PizzeriaVegetariana : IPizzeria
77 {
78     public IPizza CrearPizza()
79     {
80         return new PizzaVegetariana();
81     }
82 }
83
84 class Program
85 {
86     static void Main(string[] args)
87     {
88         // Cliente
89         IPizzeria pizzeria = new PizzeriaPepperoni();
90         IPizza pizza = pizzeria.CrearPizza();
91
92         pizza.Preparar();
93         pizza.Hornear();
94         pizza.Cortar();
95         pizza.Empacar();
96
97         Console.ReadLine();
       }
```

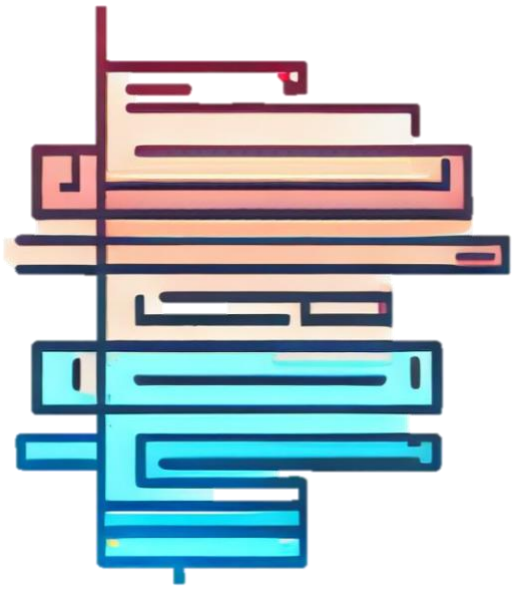
Aplicabilidad

El patrón Factory Method es adecuado cuando:

- No sabes de antemano qué clases concretas de objetos necesitas crear.
- Quieres desacoplar la creación de objetos de su uso, lo que facilita la extensibilidad y el mantenimiento del código.
- Deseas permitir que las subclases elijan la clase concreta de objeto a crear.

4.1.2 SINGLETON

Figura 6. *Singleton*



Nota: Ilustración creada a base de Inteligencia Artificial sobre Singleton, elaborada por el Autor.

Introducción

En su Revista de Ciencias de la Universidad Pablo de Olavide (Ionut Cosmin, 2016) menciona que el patrón de diseño de software de creación Singleton busca restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención es garantizar que una clase solo sea instanciada una vez y, además, proporcionar un único punto de acceso global a la misma. Esto lo consigue gracias a que es la propia clase la responsable de crear esa única instancia y a que se permite el acceso global a dicha instancia mediante un método de clase.

Aunque el patrón Singleton es una solución elegante y ampliamente adoptada para asegurar que una clase tenga una única instancia y proporcionar un punto de acceso global a esa instancia, exploraremos en detalle este patrón, analizando sus principios fundamentales, ejemplos, su estructura básica y su aplicabilidad en una variedad de contextos de desarrollo de software.

Entender el patrón Singleton es esencial para cualquier desarrollador que busque escribir código limpio, eficiente y robusto por tanto este patrón no solo garantiza la

diversidad de una instancia, sino que también mejora la organización del código y facilita la gestión de recursos compartidos, lo que resulta en un software más coherente y eficaz.

En este apartado profundizaremos en cómo implementar y aprovechar el patrón Singleton de manera efectiva en tus proyectos de desarrollo de software.

Problema

Se necesita desarrollar una aplicación de comercio electrónico en la que múltiples usuarios pueden agregar productos a su carrito de compras de manera concurrente.
Cada usuario tiene su propio carrito de compras que debe mantenerse en un estado consistente en todo momento. Si no se gestiona adecuadamente la concurrencia, podría ocurrir que dos usuarios agreguen un producto al carrito al mismo tiempo, lo que podría resultar en problemas como la duplicación de productos en el carrito o la pérdida de actualizaciones.

El patrón Singleton se puede aplicar en este escenario para garantizar que solo exista una instancia compartida de un carrito de compras en toda la aplicación. De esta manera, todos los usuarios comparten la misma instancia del carrito de compras, lo que evita problemas de concurrencia como la duplicación de productos.

Solución

El patrón Singleton se usa para garantizar que en toda la aplicación exista una única instancia compartida del carrito de compras. El patrón Singleton se implementa creando una clase **ShoppingCart** con un constructor privado y una propiedad estática llamada **Instance** que proporciona acceso a la instancia única.

Esto asegura que, sin importar cuántos usuarios intenten agregar productos simultáneamente, todos comparten la misma instancia del carrito. Esto evita problemas de concurrencia, como la duplicación de productos en el carrito, al tiempo que permite un acceso seguro y controlado al carrito de compras compartido en toda la aplicación.

Estructura

La estructura de la solución que utiliza el patrón Singleton para gestionar un carrito de compras compartido en una aplicación de comercio electrónico se compone de las siguientes partes:

Clase ShoppingCart (Carrito de Compras):

- Esta es la clase que representa el carrito de compras y aplica el patrón Singleton.
- Contiene una lista de productos para mantener los elementos del carrito.
- Tiene un constructor privado para evitar la creación directa de instancias desde fuera de la clase.
- Define una propiedad estática llamada **Instance**, que proporciona acceso a la única instancia del carrito de compras. Si la instancia aún no existe, se crea; de lo contrario, se devuelve la instancia existente.
- Proporciona métodos como **AddProduct** para agregar productos al carrito y **GetProducts** para obtener el contenido del carrito.

- **Clase Product (Producto):**
- Representa un producto que puede agregarse al carrito de compras. Esta clase puede tener propiedades como Name y Price para describir el producto.

La estructura general de la solución se enfoca en la clase ShoppingCart y cómo se gestiona su instancia única a través del patrón Singleton. La aplicación puede utilizar esta única instancia compartida para que múltiples usuarios interactúen con el carrito de compras de manera segura y evitando problemas de concurrencia.

Pseudocódigo

A continuación, se detalla la solución con el patrón de diseño Singleton al problema propuesto sobre un carrito de compras en pseudocódigo.

```
1  /// Paso 1: Crear la clase ShoppingCart que implementa el Singleton
2
3 Clase ShoppingCart
4     Instancia Privada Estática instance /
5     // Paso 2: Crear una instancia estática privada
6     Lista products // Datos del carrito de compras
7
8     // Paso 3: El constructor es privado para
9           //evitar instancias directas
10    Función Privada ShoppingCart()
11        products = Nueva Lista de Productos
```

```
12      Fin Función
13
14      // Paso 4: Propiedad estática para
15      //acceder a la instancia Singleton
16      Función Estática Obtener Instancia()
17          Si instance es Nulo Entonces
18          // Si la instancia no existe, crearla
19              instance = Nueva Instancia de ShoppingCart
20          Fin Si
21          Devolver instance
22          // Devolver la instancia existente o recién creada
23      Fin Función
24
25           // Métodos para agregar productos y obtener
26          //el contenido del carrito de compras
27      Función AgregarProducto(Producto product)
28          products.Agregar(product)
29      Fin Función
30
31      Función ObtenerProductos()
32          // Devolver una copia de la lista de productos
33          Devolver Nueva Lista de Productos(products)
34      Fin Función
35
36  Fin Clase
37
38  // Clase para representar un producto
39  Clase Producto
40      Propiedad Nombre
41      Propiedad Precio
42  Fin Clase
```

Implementación

Para realizar la implementación del patrón de Singleton nos basamos en el modelo de pseudocódigo realizado con anterioridad, aunque se puede realizar en cualquier lenguaje de programación, en este caso lo implementamos en C#.

```
1  // Paso 1: Crear la clase ShoppingCart que implementa el Singleton
```

```
2
3 public class ShoppingCart
4 {
5     private static ShoppingCart instance; // Paso 2: Crear una
6 instancia estática privada
7     private List<Product> products; // Datos del carrito de compras
8
9     // Paso 3: El constructor es privado para evitar instancias
10 directas
11     private ShoppingCart()
12     {
13         products = new List<Product>();
14     }
15
16     // Paso 4: Propiedad estática para acceder a la instancia
17 Singleton
18     public static ShoppingCart Instance
19     {
20         get
21         {
22             if (instance == null) // Si la instancia no existe,
23 crearla
24             {
25                 instance = new ShoppingCart();
26             }
27             return instance; // Devolver la instancia existente o
28 recién creada
29         }
30     }
31
32     // Métodos para agregar productos y obtener el contenido del
33 carrito de compras
34     public void AddProduct(Product product)
35     {
36         products.Add(product);
37     }
38
39     public List<Product> GetProducts()
40     {
41
```

```
42          return new List<Product>(products); // Devolver una copia de
43 la lista de productos
44      }
45 }
46
47 // Clase para representar un producto
48 public class Product
49 {
50      public string Name { get; set; }
51      public decimal Price { get; set; }
    }
```

Para entender la implementación de este código te indico cuales son los pasos para seguir para su creación.

- Creamos la clase **ShoppingCart** que implementa el Singleton.
- En el paso 2, creamos una instancia privada y estática de ShoppingCart para almacenar la única instancia Singleton.
- El constructor de **ShoppingCart** es privado (paso 3) para evitar que se creen instancias directamente desde fuera de la clase. Esto garantiza que la única instancia se controle internamente.
- En el paso 4, proporcionamos una propiedad estática llamada Instance que permite a otros componentes de la aplicación acceder a la instancia Singleton de ShoppingCart.
- Si la instancia aún no existe, se crea al llamar a Instance, de lo contrario, se devuelve la instancia existente.
- Se proporcionan métodos como **AddProduct** y **GetProducts** para agregar productos al carrito y obtener el contenido del carrito de compras.

La aplicación del patrón Singleton garantiza que todos los usuarios compartan la misma instancia del carrito de compras, lo que evita problemas de concurrencia al agregar productos simultáneamente. El acceso a la instancia Singleton se controla de manera segura y eficiente, lo que garantiza un comportamiento consistente del carrito de compras en toda la aplicación.

Aplicabilidad

El patrón Singleton es relevante en situaciones en las que es esencial asegurar que una clase tenga una única instancia en toda la aplicación y proporcionar un punto de acceso global a esa instancia. Se aplica especialmente cuando se desea centralizar el control sobre ciertos recursos compartidos o configuraciones en la aplicación.

Su utilidad radica en garantizar que múltiples partes del programa interactúen con una única instancia compartida, evitando así problemas de concurrencia y permitiendo un control coherente y eficiente de esos recursos o configuraciones a lo largo del ciclo de vida de la aplicación. Esto mejora la mantenibilidad y la gestión de componentes críticos, como registros, conexiones de base de datos o configuraciones de aplicación, simplificando el código y previniendo la creación de instancias múltiples no deseadas.

4.1.3 PROTOTYPE

Figura 7. *Prototype*

Nota: Ilustración creada a base de Inteligencia Artificial sobre Prototype, elaborada por el Autor.

Introducción

El patrón de diseño Prototype, es un patrón de diseño muy importante en la programación orientada a objetos que se utiliza para crear nuevos objetos duplicando o clonando instancias existentes, conocidas como prototipos. En lugar de crear objetos desde cero, el patrón Prototype ofrece una manera eficiente de generar copias idénticas de objetos existentes, lo que resulta particularmente útil cuando la creación de un objeto es costosa en términos de recursos o tiempo. Este enfoque promueve la flexibilidad y la reutilización del código, permitiendo la creación de nuevos objetos a partir de un prototipo predefinido sin conocer los detalles de su implementación interna.

Para (Ionut Cosmin, 2016) el patrón de diseño Prototype consiste en crear un duplicado de un objeto, clonando, para ello, una instancia de ese objeto que ya haya sido creado. Para ello, el patrón tiene que especificar el tipo de objeto que quiere clonar, creando así un 'prototipo' de esta instancia. Este tipo o clase de objetos deberá contener en su interfaz el procedimiento que permite solicitar esa copia, siendo desarrollado luego por las clases concretas del patrón que desean crear ese clon.

El patrón Prototype consta de dos componentes clave: el prototipo, que es la clase base de la que se crean las copias, y el gestor de prototipos, que administra y almacena las instancias prototipo. Esta técnica se utiliza ampliamente en el desarrollo de software cuando se requiere la creación eficiente de objetos con características similares, como en la clonación de objetos, la creación de instancias de objetos complejos o la gestión de estados.

El patrón Prototype se destaca por su capacidad para reducir la duplicación de código y recursos, permitiendo la creación de objetos personalizados de manera rápida y eficiente. Ahora presentamos un problema que nos ayudará a comprender como el patrón de diseño Prototype funciona dentro de la programación.

Problema

Con un grupo de amigos estamos desarrollando un sistema de diseño de personajes para un videojuego. Los personajes en el juego pueden tener una amplia variedad de atributos, como apariencia, habilidades, armadura y armas.

El desafío es que los diseñadores de personajes necesitan crear y ajustar continuamente nuevas variaciones de personajes con diferentes combinaciones de atributos. Sin embargo, crear cada variación desde cero es una tarea laboriosa y propensa a errores. Además, si deseas realizar cambios en un personaje existente, tendrías que modificar múltiples instancias, lo que podría llevar a inconsistencias en el diseño.

Necesitas una manera eficiente de crear y ajustar personajes, manteniendo la coherencia en el diseño y permitiendo la reutilización de atributos entre diferentes personajes.

Solución

En este escenario, el patrón de diseño Prototype se presenta como la solución más adecuada para abordar los desafíos de la creación y personalización eficiente de personajes en el videojuego.
El patrón Prototype se destaca por su capacidad para crear copias idénticas de objetos existentes, lo que encaja perfectamente con la necesidad de generar múltiples variaciones de personajes a partir de prototipos.

En lugar de construir cada personaje desde cero, podemos diseñar un conjunto de prototipos de personajes que representan combinaciones de atributos predefinidos, como apariencia, habilidades, armadura y armas.

Cuando necesitemos crear un nuevo personaje o una variante, simplemente clonamos el prototipo correspondiente y luego ajustamos o personalizamos los atributos específicos según sea necesario. Esto no solo ahorra tiempo y esfuerzo, sino que también garantiza la coherencia en el diseño, ya que todos los personajes creados a partir del mismo prototipo tendrán las mismas características base.

El patrón Prototype también permite una gestión eficiente de los cambios en el diseño de personajes. Si necesitamos realizar modificaciones en un prototipo (por ejemplo, ajustar la apariencia de un personaje guerrero), estos cambios se reflejarán automáticamente en todas las instancias clonadas del prototipo, evitando inconsistencias en el diseño.

Estructura

La estructura de la solución en programación, utilizando el patrón de diseño Prototype para el problema de diseño de personajes en un videojuego, incluiría los siguientes elementos:

Clase CharacterPrototype (Prototipo de Personaje):

- Esta es la clase base que representa un prototipo de personaje.
- Contiene atributos y métodos que son comunes a todos los personajes, como apariencia, habilidades iniciales, etc.
- Define un método Clone abstracto que las subclases deben implementar para crear copias del prototipo.

Subclases de CharacterPrototype (Ejemplos de Personajes):

- Cada subclase representa un tipo específico de personaje (por ejemplo, guerrero, mago, arquero, etc.).
- Implementa el método Clone para crear una copia del personaje con atributos específicos.

Gestor de Prototipos (Prototypes Manager):

- Esta clase almacena y administra los prototipos de personajes.
- Tiene un diccionario o una estructura de datos para mapear nombres de personajes a sus respectivos prototipos.
- Proporciona métodos para registrar prototipos y para clonar prototipos en función de un nombre dado.

Para entender de manera efectiva como el patrón de diseño Prototype trabaja con este problema a continuación te mostramos el pseudocódigo de la solución.

```
1  Clase CharacterPrototype
 2     Atributos:
 3         - apariencia
 4         - habilidades
 5         - armadura
 6         - armas
```

```
    Método Clone() -> CharacterPrototype
        // Este método debe ser implementado en las subclases
Fin Clase

Subclase Guerrero : CharacterPrototype
    Implementa Clone() -> CharacterPrototype
        // Crea una nueva instancia de Guerrero y copia los atributos

Subclase Mago : CharacterPrototype
    Implementa Clone() -> CharacterPrototype
        // Crea una nueva instancia de Mago y copia los atributos

Subclase Arquero : CharacterPrototype
    Implementa Clone() -> CharacterPrototype
        // Crea una nueva instancia de Arquero y copia los atributos

Clase PrototypesManager
    Atributos:
        - Diccionario de prototipos (nombre -> CharacterPrototype)
    Método RegistrarPrototipo(nombre, prototipo)
        // Registra un prototipo con un nombre dado en el diccionario
    Método ClonarPersonaje(nombre) -> CharacterPrototype
        // Clona un personaje basado en el nombre del prototipo registrado
        Si nombre existe en el diccionario de prototipos
            Prototipo = Obtener el prototipo asociado al nombre
            Devolver Prototipo.Clone()
        Sino
            Devolver nulo o manejar el error de prototipo no encontrado
Fin Clase


```

El ejemplo planteado muestra cómo se pueden definir las clases, subclases y el gestor de prototipos. Cada subclase de **CharacterPrototype** implementa su propio método **Clone** para crear copias específicas del personaje.

El **PrototypesManager** se encarga de registrar prototipos y clonar personajes según el nombre del prototipo registrado. Esto permite una creación eficiente y personalización de personajes en el videojuego, manteniendo la coherencia en el diseño.

Implementación

Para realizar la implementación del ejemplo del patrón de diseño Prototype en relación con el problema propuesto se ha realizado un código en C# para su revisión y aplicación.

```
1 using System;
2 using System.Collections.Generic;
3
4 // Paso 1: Definir la clase base CharacterPrototype
5 public abstract class CharacterPrototype
6 {
7     public string Name { get; set; }
8     public string Appearance { get; set; }
9     public List<string> Abilities { get; set; }
10    public string Armor { get; set; }
11    public string Weapon { get; set; }
12
13    // Método abstracto para clonar personajes
14    public abstract CharacterPrototype Clone();
15 }
16
17 // Paso 2: Implementar subclases
18 //de CharacterPrototype (por ejemplo, Guerrero, Mago, Arquero)
19 public class Guerrero : CharacterPrototype
20 {
21     public Guerrero()
22     {
23         Name = "Guerrero Genérico";
24         Appearance = "Apariencia de Guerrero";
25         Abilities = new List<string> { "Habilidad de Espada",
26 "Habilidad de Escudo" };
27         Armor = "Armadura de Guerrero";
28         Weapon = "Espada";
29     }
30
31     // Implementar el método Clone para clonar Guerrero
```

```
32      public override CharacterPrototype Clone()
33      {
34          return MemberwiseClone() as CharacterPrototype;
35      }
36  }
37
38  public class Mago : CharacterPrototype
39  {
40      public Mago()
41      {
42          Name = "Mago Genérico";
43          Appearance = "Apariencia de Mago";
44          Abilities = new List<string> { "Habilidad de Hechicería",
45              "Habilidad de Telequinesis" };
46          Armor = "Vestimenta de Mago";
47          Weapon = "Varita Mágica";
48      }
49
50      // Implementar el método Clone para clonar Mago
51      public override CharacterPrototype Clone()
52      {
53          return MemberwiseClone() as CharacterPrototype;
54      }
55  }
56
57  // Paso 3: Implementar el gestor de prototipos (PrototypesManager)
58  public class PrototypesManager
59  {
60      private Dictionary<string, CharacterPrototype> prototypes = new
61  Dictionary<string, CharacterPrototype>();
62
63      // Registrar prototipos en el gestor
64      public void RegisterPrototype(string name, CharacterPrototype
65  prototype)
66      {
67          prototypes[name] = prototype;
68      }
69
70      // Clonar un personaje basado en el nombre del prototipo
71      public CharacterPrototype CloneCharacter(string name)
```

```
72      {
73          if (prototypes.ContainsKey(name))
74          {
75              return prototypes[name].Clone();
76          }
77          else
78          {
79              Console.WriteLine("Prototipo no encontrado.");
80              return null;
81          }
82      }
83  }
84
85  class Program
86  {
87      static void Main(string[] args)
88      {
89          // Paso 4: Uso del patrón Prototype
90
91          // Crear un gestor de prototipos
92          var manager = new PrototypesManager();
93
94          // Registrar prototipos de personajes
95          manager.RegisterPrototype("Guerrero", new Guerrero());
96          manager.RegisterPrototype("Mago", new Mago());
97
98          // Clonar personajes
99          var guerrero1 = manager.CloneCharacter("Guerrero");
100         var mago1 = manager.CloneCharacter("Mago");
101
102         // Personalizar personajes clonados
103         guerrero1.Name = "Sir Lancelot";
104         mago1.Name = "Merlín";
105
106         // Mostrar información de los personajes clonados
107         Console.WriteLine("Personaje 1: " + guerrero1.Name);
108         Console.WriteLine("Apariencia: " + guerrero1.Appearance);
109         Console.WriteLine("Habilidades:  "  +  string.Join(",  ",
110 guerrero1.Abilities));
111         Console.WriteLine("Armadura: " + guerrero1.Armor);
```

```
112         Console.WriteLine("Arma: " + guerrero1.Weapon);
113
114         Console.WriteLine();
115
116         Console.WriteLine("Personaje 2: " + mago1.Name);
117         Console.WriteLine("Apariencia: " + mago1.Appearance);
118         Console.WriteLine("Habilidades:  "  +  string.Join(",  ",
119 mago1.Abilities));
120         Console.WriteLine("Armadura: " + mago1.Armor);
121         Console.WriteLine("Arma: " + mago1.Weapon);
122     }
123 }
```

En el ejemplo detallamos una breve explicación del código generado, a continuación.

- Se define la clase base **CharacterPrototype** con atributos comunes y un método abstracto Clone para clonar personajes.
- Se implementan subclases como Guerrero y Mago, que heredan de **CharacterPrototype** y proporcionan valores predeterminados para los atributos en sus constructores. También implementan el método Clone utilizando **MemberwiseClone** para crear copias superficiales.
- Se crea la clase **PrototypesManager**, que almacena prototipos y permite clonar personajes basados en el nombre del prototipo.
- En el método Main, se registran los prototipos, se clonan personajes y se personalizan. Finalmente, se muestra la información de los personajes clonados.

Aplicabilidad

El patrón de diseño Prototype es aplicable en una variedad de situaciones en las que se necesita crear objetos basados en prototipos o modelos existentes para indicar cuales son los aspectos más importantes en los cuales este patrón de diseño actúa los hemos detallado a continuación.

- **Creación de Objetos Complejos:** Cuando la creación de objetos es compleja y costosa en términos de recursos (como tiempo, CPU o memoria), el patrón Prototype permite crear copias de objetos ya existentes, evitando el proceso completo de inicialización.

- **Personalización de Objetos:** Si necesitas crear objetos similares con pequeñas variaciones, el patrón Prototype facilita la creación de copias y la personalización de atributos específicos según las necesidades, lo que evita la repetición de código y mejora la eficiencia.
- **Mantenimiento de Coherencia:** En aplicaciones donde es esencial mantener la coherencia entre objetos relacionados (por ejemplo, personajes en un juego), el patrón Prototype garantiza que los objetos compartan un estado inicial común y que los cambios en un prototipo no afecten a otros objetos basados en él.
- **Reducción de Duplicación de Código:** Ayuda a reducir la duplicación de código al permitir que objetos se creen a partir de prototipos en lugar de escribir lógica de creación redundante.
- **Manejo de Estados:** Útil en situaciones donde los objetos pueden estar en varios estados y necesitas crear copias de objetos en diferentes estados sin afectar a otros objetos.
- **Aplicaciones de Diseño Gráfico:** En herramientas de diseño gráfico y editores de imágenes, el patrón Prototype se utiliza para clonar objetos gráficos (como formas, iconos o elementos de diseño) y permitir su personalización y manipulación sin afectar a objetos originales.
- **Gestión de Configuraciones:** En aplicaciones que requieren configuraciones o perfiles específicos, el patrón Prototype puede utilizarse para clonar configuraciones base y ajustarlas según las necesidades del usuario sin afectar a otras configuraciones.
- **Copias de Seguridad:** En sistemas de copias de seguridad, el patrón Prototype se puede emplear para crear copias de seguridad de datos complejos o configuraciones de sistema.

4.1.4 BUILDER

Figura 8. *Builder*

Nota: Ilustración creada a base de Inteligencia Artificial sobre Builder, elaborada por el Autor.

El patrón de diseño Builder, es un patrón de diseño utilizado en la programación orientada a objetos que se utiliza para construir objetos complejos paso a paso. A menudo, en la creación de objetos complejos, es necesario configurar numerosos atributos y opciones, lo que puede resultar en constructores con múltiples parámetros o configuraciones complejas.

El patrón Builder resuelve este problema al separar el proceso de construcción de un objeto de su representación, permitiendo la creación de objetos compuestos de manera flexible y clara.

Builder consta de dos componentes clave: el director y el Constructor, el primero se encarga de dirigir el proceso de construcción, mientras que el Constructor define cómo se crea el objeto y proporciona métodos para configurar sus atributos paso a paso. Esto permite construir objetos con diferentes configuraciones sin la necesidad de una gran cantidad de constructores sobrecargados.

Builder es especialmente útil cuando se trabaja con objetos complejos, como documentos XML, consultas de bases de datos, elementos gráficos o configuraciones avanzadas.

Proporciona un enfoque estructurado y claro para la construcción de objetos, mejorando la legibilidad del código y la flexibilidad en la creación de diferentes variantes del mismo objeto.

Seguido a esto vamos a exponer un problema en el que se deba explicar cuál es el accionar del patrón de diseño builder en la programación de soluciones.

Problema

Se requiere trabajar en un desarrollo de un sistema de construcción de casas personalizadas. Los clientes pueden elegir entre una variedad de opciones, como el número de habitaciones, el estilo arquitectónico, los materiales de construcción y las características adicionales (como piscina o garaje).

La creación de una casa requiere una configuración compleja de estas opciones, y los clientes pueden tener preferencias específicas. Sin embargo, el proceso de construcción debe ser claro y flexible, permitiendo a los clientes personalizar sus casas de manera eficiente y sin errores. Además, es necesario generar documentación detallada para la construcción de cada casa, incluyendo planos, lista de materiales y costos estimados.

En este punto la aplicación del patrón Builder se vuelve crucial para abordar estos desafíos y garantizar que las casas se construyan de manera precisa según las especificaciones de los clientes.

Solución

En atención al problema planteado para este caso el patrón de diseño Builder emerge como la solución más adecuada; en el que se el desarrollo debe garantizar la construcción de una casa que involucra numerosas opciones y configuraciones que varían de un cliente a otro, lo que podría llevar a un constructor de casas con una cantidad abrumadora de parámetros en su constructor, he aquí donde el patrón Builder demuestra su utilidad, prolijidad y aplicabilidad para el tema propuesto.

El patrón Builder permite desacoplar el proceso de construcción de una casa de su representación final; en lugar de utilizar un constructor con una lista interminable de parámetros o métodos, el patrón Builder introduce un Director que guía el proceso de

construcción, mientras que el Constructor, a través de interfaces claras y métodos específicos, se encarga de crear objetos de casa de manera paso a paso, estos desvelan las ventajas cruciales del patrón en respuesta a este tipo de problemas de trabajo.

En primer lugar, permite a los clientes personalizar sus casas de manera eficiente, seleccionando y configurando opciones específicas sin preocuparse por las partes internas del proceso de construcción.

En segundo lugar, el patrón Builder asegura la coherencia y la precisión, ya que el Constructor se encarga de configurar cada componente de la casa de manera adecuada, garantizando que no se omita ninguna opción.

Para finalizar con la separación entre el director y el Constructor facilita la generación de documentación detallada, ya que se pueden registrar las decisiones tomadas durante el proceso de construcción.

Estructura

La estructura de la solución en programación utilizando el patrón de diseño Builder para el problema de construcción de casas personalizadas constaría de los siguientes componentes:

Clase House (Casa):

- Representa el objeto final que se construirá.
- Contiene atributos que definen la casa, como el número de habitaciones, el estilo arquitectónico, los materiales de construcción y las características adicionales.
- Interfaz **IHouseBuilder** (Constructor de Casa):
- Define métodos abstractos para configurar cada aspecto de la casa.

Clase ConcreteHouseBuilder (Constructor de Casa Concreto):

- Implementa la interfaz IHouseBuilder.
- Proporciona métodos para configurar cada aspecto de la casa de manera específica, como **BuildRooms**, **SetArchitecturalStyle**, **ChooseMaterials**, **AddFeatures**, etc.
- Tiene un método para obtener la casa construida, como **GetHouse**.

Clase HouseDirector (Director de Construcción):

- Contiene una instancia de IHouseBuilder.
- Proporciona un método, por ejemplo, **ConstructHouse**, que utiliza el **IHouseBuilder** para construir una casa siguiendo un proceso paso a paso.

Cliente (Main o parte del sistema):

- Utiliza el **HouseDirector** para construir una casa personalizada.
- Puede configurar y personalizar la casa según las preferencias del cliente.

A continuación, vamos a detallar la solución del problema planteado en pseudocódigo para su mayor comprensión.

```
// Paso 1: Definir la Clase Casa (House)
Clase Casa
    Atributos:
        - NumeroHabitaciones
        - EstiloArquitectonico
        - MaterialesConstruccion
        - CaracteristicasAdicionales
    // Constructor para inicializar la Casa con valores predeterminados o
nulos
    Constructor Casa()

// Paso 2: Definir la Interfaz IConstructorCasa (IHouseBuilder)
Interfaz IConstructorCasa
    Método ConstruirHabitaciones(numero: entero)
    Método DefinirEstiloArquitectonico(estilo: cadena)
    Método ElegirMateriales(materiales: cadena)
    Método AgregarCaracteristicas(caracteristicas: cadena)
    Método ObtenerCasa() -> Casa

// Paso 3: Implementar el Constructor de Casa Concreto (ConcreteHouseBuilder)
Clase ConstructorCasa : IConstructorCasa
    Atributos:
        - Casa casaActual
    Constructor ConstructorCasa()
        // Inicializar la Casa actual
        casaActual = nueva Casa()
    Método ConstruirHabitaciones(numero: entero)
        // Implementar la lógica para construir habitaciones
    Método DefinirEstiloArquitectonico(estilo: cadena)
        // Implementar la lógica para definir el estilo arquitectónico
```

```
    Método ElegirMateriales(materiales: cadena)
        // Implementar la lógica para elegir los materiales de construcción
    Método AgregarCaracteristicas(caracteristicas: cadena)
        // Implementar la lógica para agregar características adicionales
    Método ObtenerCasa() -> Casa
        // Devolver la Casa actual construida

// Paso 4: Crear el Director de Construcción (HouseDirector)
Clase DirectorConstruccion
    Atributo constructorCasa: IConstructorCasa
    // Constructor que recibe un IConstructorCasa
    Constructor DirectorConstruccion(constructor: IConstructorCasa)
        constructorCasa = constructor
    Método ConstruirCasa()
        // Lógica para dirigir el proceso de construcción paso a paso
utilizando constructorCasa

// Paso 5: Uso del Patrón Builder
// Crear un Constructor de Casa Concreto
ConstructorCasa constructor = nueva ConstructorCasa()

// Crear un Director de Construcción y asignar el Constructor de Casa
DirectorConstruccion director = nueva DirectorConstruccion(constructor)

// Construir una Casa personalizada
director.ConstruirCasa()

// Obtener la Casa construida
Casa casaPersonalizada = constructor.ObtenerCasa()
```

Este pseudocódigo detallado representa la estructura de la solución usando el patrón Builder para la construcción de casas personalizadas del problema de ejemplo planteado. Ahora una de las cosas más importantes para reconocer el funcionamiento de los patrones de diseño es su aplicación en un lenguaje de programación si bien es cierto lo podemos realizar en cualquier lenguaje, hoy lo hemos realizado en C#, para su aplicación y comprensión.

```
// Paso 1: Definir la Clase Casa (House)
public class House
{
    public int NumberOfRooms { get; set; }
    public string ArchitecturalStyle { get; set; }
```

```
6     public string ConstructionMaterials { get; set; }
7     public string AdditionalFeatures { get; set; }
8
9     // Constructor para inicializar la Casa con valores
10 predeterminados o nulos
11    public House()
12    {
13        // Puedes establecer valores predeterminados aquí si es
14 necesario
15    }
16 }
17
18 // Paso 2: Definir la Interfaz IConstructorCasa (IHouseBuilder)
19 public interface IHouseBuilder
20 {
21    void BuildRooms(int numberOfRooms);
22    void DefineArchitecturalStyle(string style);
23    void ChooseMaterials(string materials);
24    void AddFeatures(string features);
25    House GetHouse();
26 }
27
28 // Paso 3: Implementar el Constructor de Casa Concreto
29 (ConcreteHouseBuilder)
30 public class ConcreteHouseBuilder : IHouseBuilder
31 {
32    private House currentHouse = new House();
33
34    public void BuildRooms(int numberOfRooms)
35    {
36        currentHouse.NumberOfRooms = numberOfRooms;
37    }
38
39    public void DefineArchitecturalStyle(string style)
40    {
41        currentHouse.ArchitecturalStyle = style;
42    }
43
44    public void ChooseMaterials(string materials)
45    {
```

```
46          currentHouse.ConstructionMaterials = materials;
47      }
48
49      public void AddFeatures(string features)
50      {
51          currentHouse.AdditionalFeatures = features;
52      }
53
54      public House GetHouse()
55      {
56          return currentHouse;
57      }
58  }
59
60  // Paso 4: Crear el Director de Construcción (HouseDirector)
61  public class HouseConstructionDirector
62  {
63      private IHouseBuilder houseBuilder;
64
65      public HouseConstructionDirector(IHouseBuilder builder)
66      {
67          houseBuilder = builder;
68      }
69
70      public void ConstructHouse()
71      {
72          // Lógica para dirigir el proceso de construcción paso a paso
73  utilizando houseBuilder
74          houseBuilder.BuildRooms(4); // Ejemplo: 4 habitaciones
75          houseBuilder.DefineArchitecturalStyle("Colonial");        //
76  Ejemplo: Estilo colonial
77          houseBuilder.ChooseMaterials("Brick"); // Ejemplo: Ladrillo
78          houseBuilder.AddFeatures("Swimming   Pool");   //   Ejemplo:
79  Piscina
80      }
81  }
82
83  // Paso 5: Uso del Patrón Builder
84  // Crear un Constructor de Casa Concreto
85  IHouseBuilder builder = new ConcreteHouseBuilder();
```

```
86
87 // Crear un Director de Construcción y asignar el Constructor de Casa
88 HouseConstructionDirector director = new
89 HouseConstructionDirector(builder);
90
91 // Construir una Casa personalizada
92 director.ConstructHouse();
93
94 // Obtener la Casa construida
95 House customHouse = builder.GetHouse();
96
```

Este código en C# ilustra cómo se implementa el patrón Builder para construir casas personalizadas de manera eficiente y flexible, permitiendo a los clientes configurar los detalles de la casa sin preocuparse por la lógica interna de construcción.
Para comprender el código proporcionado te dejamos una explicación detallada de la solución al problema con su respectiva explicación.

- Se define la clase **House** para representar una casa personalizada con atributos como número de habitaciones, estilo arquitectónico, materiales de construcción y características adicionales.
- Se define la interfaz **IHouseBuilder** que contiene métodos para configurar los aspectos de la casa, como el número de habitaciones, el estilo arquitectónico, los materiales de construcción y las características adicionales.
- Se implementa la clase **ConcreteHouseBuilder**, que implementa la interfaz **IHouseBuilder** y se encarga de construir una casa concreta siguiendo los pasos definidos.
- Se crea la clase **HouseConstructionDirector**, que supervisa el proceso de construcción de la casa, utiliza un objeto IHouseBuilder para configurar los detalles y asegura que la construcción siga un flujo específico.
- En el paso 5, se utiliza el patrón Builder. Se crea un constructor concreto y se asigna al director de construcción. Luego, se invoca el método **ConstructHouse** del director para guiar el proceso de construcción. Finalmente, se obtiene la casa personalizada construida.

Aplicabilidad

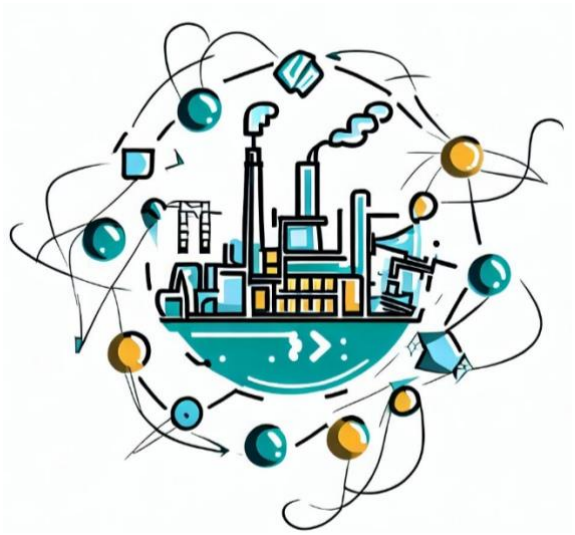
El patrón Builder es uno de los más importantes dentro de la programación de aplicaciones, para ello hemos detallado su accionar dentro de algunos de los aspectos de su campo de acción en la programación los cuales te detallamos a continuación.

- **Creación de Objetos Complejos:** Cuando necesitas crear objetos que tienen una configuración compleja con múltiples partes y opciones, el patrón Builder permite construir estos objetos paso a paso sin sobrecargar los constructores con numerosos parámetros.
- **Personalización de Objetos:** Cuando los objetos deben ser altamente personalizables y permitir que los clientes configuren diferentes aspectos de un objeto según sus preferencias sin tener que proporcionar un gran número de constructores con parámetros.

- **Múltiples Representaciones:** Cuando deseas crear diferentes representaciones de un objeto, como formatos de archivo diferentes o configuraciones variadas, pero deseas mantener un proceso de construcción coherente.
- **Construcción de Productos Completos:** En casos donde la construcción de un objeto implica varios pasos complejos y se necesita un control detallado sobre el proceso de construcción. Esto es común en la creación de documentos complejos, informes, objetos gráficos y configuraciones avanzadas.
- **Separación de Responsabilidades:** Cuando quieres separar la lógica de construcción de un objeto de su estructura, lo que facilita la reutilización del código y la mantenibilidad al permitir cambios en la estructura sin afectar el proceso de construcción.
- **Generación de Documentación o Informes:** En situaciones en las que es necesario generar documentación detallada o informes que requieren recopilar información de múltiples fuentes y configurarla en un formato específico.
- **Evitar Constructores Sobrecargados:** Cuando deseas evitar la creación de constructores con una gran cantidad de parámetros, lo que puede ser propenso a errores y difícil de mantener.

4.1.5 ABSTRACT FACTORY

Figura 9. *Abstract Factory*



Nota: Ilustración creada a base de Inteligencia Artificial sobre Abstract Factory, elaborada por el Autor.

El patrón de diseño Abstract Factory es un patrón de diseño fundamental en la programación orientada a objetos que se utiliza para crear familias de objetos relacionados sin especificar sus clases concretas en el código cliente.
A diferencia de la Fábrica Simple, que crea objetos individuales, la Fábrica Abstracta se centra en la creación de familias completas de objetos que deben trabajar juntos de manera coherente.

Para (Maciej Serda et al., 2009) en su libro Patrones de diseño, refactorización y anti-patrones define al Abstract Factory como un patrón de diseño que provee una interfaz para crear familias de objetos producto relacionados o que dependen entre sí, sin especificar sus clases concretas.

Abstract Factory consta de dos componentes principales: la Fábrica Abstracta y las Fábricas Concretas. La Fábrica Abstracta define una interfaz para crear objetos relacionados, pero no implementa la creación real de objetos. Las Fábricas Concretas, por otro lado, implementan esa interfaz para crear objetos específicos.
Este patrón promueve la flexibilidad y la extensibilidad del código, ya que permite cambiar la implementación subyacente de una familia de objetos sin afectar al código que los utiliza. Es especialmente útil cuando necesitas garantizar que los objetos creados sean compatibles entre sí y cuando debes adaptar la creación de objetos a diferentes contextos o plataformas.

Para mayor comprensión de lo que es Abstract Factory formulamos un problema en cual implementamos este patrón de diseño lo analizaremos y pondremos en práctica su accionar en la programación de soluciones.

Problema

Una escuela del milenio necesita desarrollar una aplicación de dibujo en la que los usuarios pueden crear formas geométricas, como círculos y cuadrados, y aplicarles diferentes estilos de llenado y bordes, como colores y patrones. Además, se necesita que tu aplicación sea capaz de funcionar tanto en un entorno de escritorio como en un entorno web, y necesitas que la creación de formas y la aplicación de estilos sean coherentes en ambas plataformas.

El desafío es cómo diseñar un sistema que permita la creación de formas y la aplicación de estilos de manera flexible y consistente en diferentes plataformas. El patrón Abstract Factory se convierte en una solución eficaz y más importante para este problema, ya que permite crear familias de objetos relacionados, como formas y estilos, adaptados a cada plataforma específica y garantiza su coherencia en la aplicación.

Solución

Para dar solución a este requerimiento nos centramos en la implementación del patrón Abstract Factory para abordar el desafío de crear formas geométricas y aplicar estilos de manera uniforme en entornos de escritorio y web.

Mediante la definición de interfaces abstractas para formas, fábricas abstractas para formas y estilos de llenado y bordes, hemos logrado una estructura flexible que permite la creación de familias coherentes de objetos relacionados.

Las fábricas concretas para cada plataforma (entorno de escritorio y web) se encargan de la creación de objetos y la aplicación de estilos específicos, garantizando la adaptabilidad y la consistencia en la interfaz de usuario. Al seleccionar la fábrica concreta según la plataforma en tiempo de ejecución, hemos logrado una solución escalable y mantenible que satisface las necesidades de diferentes contextos de aplicación sin comprometer la coherencia del diseño.

Estructura

La estructura de la solución es relativamente importante para comprender como se implementará esto en un caso de solución real, aquí la estructura de nuestro problema planteado.

- **Interfaz Abstracta para Formas (IAbstractShape):** Define métodos para operaciones relacionadas con formas geométricas, como el método "Draw" para dibujar una forma. Las clases concretas que implementan esta interfaz representarán formas específicas, como círculos y cuadrados.
- **Interfaz Abstracta para Fábricas de Formas (IAbstractShapeFactory):** Define métodos para crear diferentes tipos de formas geométricas (por ejemplo, círculos y cuadrados) y para aplicar estilos a estas formas, como llenado y bordes. Las clases concretas de estas fábricas implementarán estos métodos según la plataforma.
- **Interfaz Abstracta para Estilos de Llenado (IAbstractFillStyle):** Define métodos relacionados con el llenado de formas, como el método "Fill" para aplicar un estilo de llenado específico a una forma.
- **Interfaz Abstracta para Estilos de Borde (IAbstractBorderStyle):** Define métodos relacionados con los bordes de las formas, como el método "ApplyBorder" para aplicar un estilo de borde a una forma.
- **Fábricas Concretas (DesktopShapeFactory y WebShapeFactory):** Implementan la interfaz IAbstractShapeFactory para crear objetos de formas concretas y aplicar estilos específicos para la plataforma en la que se ejecuta la aplicación. Cada fábrica concreta se adapta a su entorno respectivo.
- **Aplicación Principal:** aquí se selecciona la fábrica concreta adecuada (DesktopShapeFactory o WebShapeFactory) según la plataforma en la que se esté ejecutando la aplicación. Luego, se utilizan los métodos de la fábrica para crear formas y aplicar estilos de manera coherente, adema esta estructura garantiza la coherencia en la creación de formas geométricas y la aplicación de estilos en diferentes plataformas, permitiendo una adaptabilidad eficiente a entornos de escritorio y web sin afectar la lógica principal de la aplicación.

Podemos comprender de mejor manera esta solución su le damos un vistazo al pseudocodigo, mismo que te lo dejo a continuación para tu revisión y aplicación.

Pseudocodigo

```
// Paso 1: Definir una Interfaz Abstracta para Formas
Interfaz IAbstractShape
    Método Draw()

// Paso 2: Definir una Interfaz Abstracta para Fábricas de Formas
Interfaz IAbstractShapeFactory
    Método CreateCircle() -> IAbstractShape
    Método CreateSquare() -> IAbstractShape
    Método ApplyFillStyle(IAbstractFillStyle fillStyle)
    Método ApplyBorderStyle(IAbstractBorderStyle borderStyle)

// Paso 3: Definir una Interfaz Abstracta para Estilos de Llenado
Interfaz IAbstractFillStyle
    Método Fill()

// Paso 4: Definir una Interfaz Abstracta para Estilos de Borde
Interfaz IAbstractBorderStyle
    Método ApplyBorder()

// Paso 5: Implementar Fábricas Concretas para cada Plataforma
Clase DesktopShapeFactory Implementa IAbstractShapeFactory
    // Implementar métodos para crear formas y aplicar estilos en el
entorno de escritorio

Clase WebShapeFactory Implementa IAbstractShapeFactory
    // Implementar métodos para crear formas y aplicar estilos en el
entorno web

// Paso 6: En la aplicación principal, seleccionar la Fábrica Concreta
según la Plataforma
IAbstractShapeFactory factory

Si plataforma == Plataforma.Desktop
    factory = Nueva Instancia de DesktopShapeFactory()
Si plataforma == Plataforma.Web
    factory = Nueva Instancia de WebShapeFactory()

// Paso 7: Utilizar la Fábrica Concreta para crear Formas y Aplicar
Estilos
IAbstractShape circle = factory.CreateCircle()
```

```
IAbstractShape square = factory.CreateSquare()

factory.ApplyFillStyle(fillStyle)
factory.ApplyBorderStyle(borderStyle)

circle.Draw()
square.Draw()
```

A continuación, proporcionamos la aplicación de este pseudocodigo en el lenguaje C# para que lo puedan implementar en su entorno y comprender el funcionamiento de Abstract Factory en una solución de desarrollo, en este caso de creación de figuras geométricas.

Implementación

```
// Paso 1: Definir una Interfaz Abstracta para Formas
public interface IAbstractShape
{
    void Draw();
}

// Paso 2: Definir una Interfaz Abstracta para Fábricas de Formas
public interface IAbstractShapeFactory
{
    IAbstractShape CreateCircle();
    IAbstractShape CreateSquare();
    void ApplyFillStyle(IAbstractFillStyle fillStyle);
    void ApplyBorderStyle(IAbstractBorderStyle borderStyle);
}

// Paso 3: Definir una Interfaz Abstracta para Estilos de Llenado
public interface IAbstractFillStyle
{
    void Fill();
}

// Paso 4: Definir una Interfaz Abstracta para Estilos de Borde
public interface IAbstractBorderStyle
```

```
24 {
25     void ApplyBorder();
26 }
27
28 // Paso 5: Implementar Fábricas Concretas para cada Plataforma
29 public class DesktopShapeFactory : IAbstractShapeFactory
30 {
31     // Implementar métodos para crear formas y aplicar estilos en
32 el entorno de escritorio
33     public IAbstractShape CreateCircle()
34     {
35         return new DesktopCircle();
36     }
37
38     public IAbstractShape CreateSquare()
39     {
40         return new DesktopSquare();
41     }
42
43     public void ApplyFillStyle(IAbstractFillStyle fillStyle)
44     {
45         // Aplicar estilo de llenado en el entorno de escritorio
46     }
47
48     public void ApplyBorderStyle(IAbstractBorderStyle borderStyle)
49     {
50         // Aplicar estilo de borde en el entorno de escritorio
51     }
52 }
53
54 public class WebShapeFactory : IAbstractShapeFactory
55 {
56     // Implementar métodos para crear formas y aplicar estilos en
57 el entorno web
58     public IAbstractShape CreateCircle()
59     {
60         return new WebCircle();
61     }
62
63     public IAbstractShape CreateSquare()
```

```
64      {
65          return new WebSquare();
66      }
67
68      public void ApplyFillStyle(IAbstractFillStyle fillStyle)
69      {
70          // Aplicar estilo de llenado en el entorno web
71      }
72
73      public void ApplyBorderStyle(IAbstractBorderStyle borderStyle)
74      {
75          // Aplicar estilo de borde en el entorno web
76      }
77  }
78
79  // Paso 6: En la aplicación principal, seleccionar la Fábrica
80  Concreta según la Plataforma
81  public class MainApp
82  {
83      static void Main(string[] args)
84      {
85          IAbstractShapeFactory factory;
86
87          if (plataforma == Plataforma.Desktop)
88          {
89              factory = new DesktopShapeFactory();
90          }
91          else if (plataforma == Plataforma.Web)
92          {
93              factory = new WebShapeFactory();
94          }
95
96          // Paso 7: Utilizar la Fábrica Concreta para crear Formas y
97  Aplicar Estilos
98          IAbstractShape circle = factory.CreateCircle();
99          IAbstractShape square = factory.CreateSquare();
100
101         factory.ApplyFillStyle(fillStyle);
102         factory.ApplyBorderStyle(borderStyle);
103
```

```
104         circle.Draw();
105         square.Draw();
106     }
107 }
```

El patrón de diseño Abstract Factory se puede aplicar a cualquier lenguaje de programación, aunque con la variación de la sintaxis de acuerdo con el que el programador decida utilizar, a continuación, tenemos una breve explicación del código generado en C#.

- **Interfaz IAbstractShape:** Esta interfaz define un método Draw() que representa la operación común para dibujar una forma geométrica.
- **Interfaz IAbstractShapeFactory:** Esta interfaz abstracta declara métodos para crear diferentes tipos de formas geométricas (círculos y cuadrados) y para aplicar estilos de llenado y bordes a estas formas. Establece una interfaz común para las fábricas de formas concretas.
- **Interfaz IAbstractFillStyle e IAbstractBorderStyle:** Estas interfaces abstractas definen métodos relacionados con el llenado y los bordes de las formas, respectivamente. Permiten la implementación de estilos de llenado y bordes específicos para cada plataforma.
- **DesktopShapeFactory y WebShapeFactory:** Son clases concretas que implementan la interfaz IAbstractShapeFactory. Cada una de estas fábricas concretas se encarga de crear objetos de formas concretas (por ejemplo, círculos y cuadrados) y aplicar estilos específicos para la plataforma en la que se ejecuta la aplicación (entorno de escritorio y web, respectivamente).
- **Aplicación Principal:** se selecciona la fábrica concreta adecuada (ya sea DesktopShapeFactory o WebShapeFactory) según la plataforma en la que se ejecuta la aplicación.

En este capítulo, hemos explorado los fundamentos de los patrones de diseño creacionales, un conjunto esencial de herramientas en el mundo de la programación orientada a objetos. Estos patrones se centran en la creación de objetos, abordando cuestiones como la encapsulación de la creación, la flexibilidad en la instancia de objetos y la gestión de la complejidad en el proceso de creación.

Comenzamos con el patrón Factory Method, que proporciona una interfaz para crear objetos, pero permite a las subclases alterar el tipo de objetos que se crearán. Esto promueve la flexibilidad y extensibilidad al permitir que las clases hijas determinen qué objetos específicos deben crear.

También exploramos el patrón Abstract Factory, que va un paso más allá al proporcionar una familia de objetos relacionados, lo que permite crear múltiples objetos que funcionan juntos de manera coherente. Este patrón es útil cuando necesitamos crear sistemas con componentes interdependientes.

Por otra parte, Singleton, aborda el problema de asegurar que solo exista una instancia de una clase en toda la aplicación. Esto es esencial cuando necesitamos compartir recursos o datos críticos entre diferentes partes de un programa.

Además, discutimos cómo los patrones creacionales pueden ayudarnos a lograr varios objetivos de diseño, como la encapsulación de la complejidad, la promoción de la reutilización del código y la garantía de la coherencia en la creación de objetos. También enfatizamos la importancia de elegir el patrón adecuado para el problema específico que estamos tratando de resolver.

En resumen, los patrones de diseño creacionales son herramientas valiosas para crear objetos de manera flexible, eficiente y controlada. Al comprender y aplicar estos patrones, los programadores pueden mejorar la estructura y el diseño de sus aplicaciones, lo que resulta en sistemas más mantenibles, extensibles y adaptables. En los capítulos siguientes, exploraremos patrones de diseño adicionales que abordan otros aspectos cruciales del desarrollo de software.

CAPITULO 5

EJERCICIOS PROPUESTOS

5.1 Ejercicios Factory Method

Creación de vehículos: Supongamos que estás desarrollando un programa de simulación de una fábrica de automóviles. Crea una jerarquía de clases que incluya un "Factory Method" para crear diferentes tipos de vehículos, como coches, camiones y motocicletas. Cada tipo de vehículo debe tener sus propias subclases con características específicas. Utiliza el patrón Factory Method para instanciar los diferentes vehículos en tu programa.

Creación de personajes de un videojuego: Imagina que estás desarrollando un videojuego de rol en el que los jugadores pueden elegir entre diferentes clases de personajes, como guerreros, magos y arqueros. Crea una jerarquía de clases que incluya un "Factory Method" para crear instancias de cada tipo de personaje. Cada clase de personaje debe tener sus propias habilidades y atributos únicos. Utiliza el patrón Factory Method para crear personajes en el juego.

Creación de documentos en un editor de texto: Diseña un editor de texto simple que pueda crear diferentes tipos de documentos, como documentos de texto, hojas de cálculo y presentaciones. Crea una jerarquía de clases que incluya un "Factory Method" para crear instancias de estos documentos. Cada tipo de documento debe tener su propia lógica de edición y formato. Utiliza el patrón Factory Method para crear documentos en el editor.

5.2. Ejercicios Singleton

Registro de eventos: Imagina que estás desarrollando un sistema de registro de eventos para una aplicación. Quieres asegurarte de que solo haya una instancia del registro de eventos en todo el programa para evitar registros duplicados. Implementa el patrón Singleton para la clase de registro de eventos de manera que solo haya una instancia de la clase en toda la aplicación.

Configuración global: En una aplicación, es común tener una clase que maneje la configuración global, como preferencias de usuario o configuraciones de la aplicación. Utiliza el patrón Singleton para crear una clase de configuración que almacene y proporcione acceso a estas configuraciones globales. Esto garantizará que todas las partes de la aplicación compartan la misma configuración.

Manejo de conexiones a una base de datos: En una aplicación que se conecta a una base de datos, es importante garantizar que solo haya una instancia activa de la conexión en todo momento para evitar problemas de concurrencia y recursos. Implementa el patrón Singleton para administrar la conexión a la base de datos de manera que solo haya una instancia de la conexión en toda la aplicación.

5.3. Ejercicios Prototype

Editor de Mapas Interactivo: Supongamos que estás construyendo un editor de mapas para un juego donde se necesita implementar el patrón Prototype para permitir a los usuarios clonar y modificar regiones del mapa. Cada región podría ser un objeto que almacena información sobre terreno, elementos y enemigos. Los usuarios pueden clonar regiones existentes, ajustarlas y colocarlas en diferentes partes del mapa sin tener que crearlas desde cero.

Sistema de Menús Personalizados: Diseña un sistema de menús para una aplicación de restaurante. Utiliza el patrón Prototype para crear elementos de menú, como entradas, platos principales y postres. Los usuarios pueden personalizar su propio menú clonando elementos de menú existentes y ajustando los ingredientes o las descripciones según sus preferencias.

Gestión de Widgets de Interfaz de Usuario: En una aplicación de interfaz de usuario compleja, puedes aplicar el patrón Prototype para administrar widgets de interfaz de usuario, como botones, campos de entrada y paneles. Crea una clase base Widget con métodos de clonación. Los diseñadores de interfaz de usuario pueden clonar widgets existentes y personalizar su apariencia y comportamiento según las necesidades de la aplicación.

5.4. Ejercicios Builder

Construcción de Automóviles Personalizados: Imagina que estás desarrollando un sistema para configurar automóviles personalizados. Implementa el patrón Builder para crear un conjunto de clases que permitan a los usuarios seleccionar opciones de motor, color de carrocería, tapicería, características adicionales, etc. Los usuarios pueden utilizar el Builder para configurar su automóvil ideal paso a paso y, al final, obtener un objeto automóvil completamente personalizado.

Creación de Menús en un Restaurante: En un sistema de pedidos en un restaurante, usa el patrón Builder para crear menús personalizados. Crea una clase MenuBuilder que permita a los chefs y gerentes del restaurante construir menús al seleccionar platos, bebidas y postres disponibles. Esto facilitaría la creación y gestión de menús especiales para eventos o promociones.

Construcción de Informes Personalizados: En una aplicación empresarial, implementa el patrón Builder para generar informes personalizados. Crea una clase ReportBuilder que permita a los usuarios seleccionar qué datos desean incluir en el informe, cómo se deben formatear y cómo se debe presentar. Los usuarios pueden utilizar el Builder para configurar informes específicos para sus necesidades sin tener que preocuparse por la complejidad de la generación de informes.

5.5. Ejercicios Con Abstract Factory

Creación de Aplicaciones Móviles para Plataformas Múltiples: Imagina que estás desarrollando una suite de aplicaciones móviles que deben estar disponibles tanto en Android como en iOS. Utiliza el patrón Abstract Factory para crear familias de objetos relacionados, como botones, cuadros de diálogo y barras de navegación, específicos de cada plataforma. Luego, implementa fábricas abstractas separadas para Android y iOS que produzcan estos elementos de interfaz de usuario correspondientes. Esto garantizará una apariencia y funcionalidad coherentes en ambas plataformas.

Producción de Productos Electrónicos: Supongamos que estás diseñando un sistema de producción para una empresa de electrónicos que fabrica dispositivos como teléfonos inteligentes y tabletas. Utiliza el patrón Abstract Factory para crear familias de productos electrónicos, como componentes de hardware y software, para diferentes modelos de dispositivos. Crea una fábrica abstracta para cada tipo de dispositivo (por ejemplo, teléfonos y tabletas) que se encargue de crear productos coherentes y compatibles dentro de cada familia.

Creación de Juegos de Mesa: En un proyecto de desarrollo de juegos de mesa, puedes aplicar el patrón Abstract Factory para crear juegos de diferentes temáticas y estilos. Crea una fábrica abstracta llamada FactoryJuego que defina métodos para crear componentes comunes de juegos de mesa, como tableros, piezas y reglas. Luego, implementa fábricas concretas, como FactoryAjedrez y FactoryMonopoly, que produzcan juegos específicos con componentes y reglas únicas para cada juego.

REFERENCIAS

Alfred, J., & Lázaro, C. (2019). *Desarrollo del módulo de contabilidad en lenguaje .net core cumpliendo con los principios de desarrollo solid y de arquitectura limpia usando los framework Entity, Code first, Devexpress y Bootstrap 4 en un repositorio SQL server con control de versiones git bajo el marco de trabajo scrum en microshif.* http://repositoriodspace.unipamplona.edu.co/jspui/handle/20.500.12744/5636

Barnes, D. J., & Kölling, M. (2017). *Programación orientada a objetos con Java Una introducción práctica usando BlueJ*. PEARSON. www.FreeLibros.me

Blancarte, O. J. (n.d.). *Introducción a los Patrones de Diseño*.

Blancarte Oscar. (n.d.). *Introducción a los Patrones de Diseño*.

Campo, G. D. (2009). *Patrones de Diseño, Refactorización y Antipatrones. Ventajas y Desventajas de su Utilización en el Software Orientado a Objetos* . *4*.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2005). *Patrones de Diseño* (Primera Edición).

Ionut Cosmin, P. (2016). Patrones de Diseño. *MoleQla: Revista de Ciencias de La Universidad Pablo de Olavide, ISSN-e 2173-0903, Nº. 23, 2016*, *23*, 36. https://dialnet.unirioja.es/servlet/articulo?codigo=5680557&info=resumen&idioma=SPA

Maciej Serda, Becker, F. G., Cleary, M., Team, R. M., Holtermann, H., The, D., Agenda, N., Science, P., Sk, S. K., Hinnebusch, R., Hinnebusch A, R., Rabinovich, I., Olmert, Y., Uld, D. Q. G. L. Q., Ri, W. K. H. U., Lq, V., Frxqwu, W. K. H., Zklfk, E., Edvhg, L. V, … فاطمی, ح. (2009). Patrones de diseño, refactorización y antipatrones. *Uniwersytet Śląski*, *7*(1), 343–354. https://doi.org/10.2/JQUERY.MIN.JS

Montero, E. L. P., & Pérez, F. de M. H. (2019a). La programación orientada a objetos facilidad para crear. *I+ T+ C- Research, Technology and Science*, *1*(13), 96–100. https://revistas.unicomfacauca.edu.co/ojs/index.php/itc/article/view/241

Montero, E. L. P., & Pérez, F. de M. H. (2019b). La programación orientada a objetos facilidad para crear. *I+ T+ C- Research, Technology and Science*, *1*(13), 96–100. https://revistas.unicomfacauca.edu.co/ojs/index.php/itc/article/view/241

Patrones creacionales. (n.d.). Retrieved September 27, 2023, from https://refactoring.guru/es/design-patterns/creational-patterns

SOLID: los 5 principios que te ayudarán a desarrollar software de calidad. (n.d.). Retrieved September 27, 2023, from https://profile.es/blog/principios-solid-desarrollo-software-calidad/

View of State of the art research in: Clean Architecture and SOLID principles. (n.d.). Retrieved September 27, 2023, from https://rsdjournal.org/index.php/rsd/article/view/37198/31702

Z, J. C. M., Henao, C., Henao, F., & Zapata, E. (2021). Utilización de Arquitecturas Limpias para Trabajo con Buenas Prácticas en la Construcción de Aplicaciones Java. *Revista Innovación Digital y Desarrollo Sostenible - IDS*, *1*(2), 133–140. https://doi.org/10.47185/27113760.v1n2.37