**SERVICIO NACIONAL DE APRENDIZAJE – SENA**

Logotipo

Descripción generada automáticamente

**CENTRO DE COMERCIO REGIONAL ANTIOQUIA**

**TECNOLOGÍA EN ANÁLISIS Y DESARROLLO DE SOFTWARE - 2675805**

**Evidencia de desempeño: GA4-220501095-AA2-EV05 - Desarrollar la arquitectura de software de acuerdo al patrón de diseño seleccionado**

**DANIEL FELIPE ARIAS CORREDOR**

**2023**

Introducción

El presente documento tiene como propósito presentar el resultado de la investigación acerca de los patrones de diseño en el desarrollo de software. Se describirá brevemente que es un patrón, se presentarán sus tres clasificaciones y se darán ejemplos de cada uno.

Patrones de diseño de software

Los patrones de diseño de software son estándares comúnmente utilizados para la creación y estructuración del código de una aplicación. Cada patrón es como un plano o modelo que se puede adecuar a las necesidades del proyecto, de acuerdo con la funcionalidad que puedan aportar, permitiendo un código más ordenado o ahorro de tiempo y recursos. Los patrones se dividen en tres grupos fundamentales: comportamentales, estructurales y creacionales. A continuación, se presentarán algunos ejemplos de cada uno de estos tipos de patrones.

Patrones Creacionales

Los patrones creacionales proporcionan mecanismos o estándares de creación de objetos, lo que facilitaría la reutilización de código.

* Factory Method

Es un patrón de diseño enfocado en la creación de objetos nuevos únicamente desde un método denominado *fábrica.* Especialmente útil cuando no se conocen de antemano todos los tipos de objetos que usará el código, permitiendo crear nuevos objetos sin tener que modificar todo el código. Un ejemplo se puede apreciar en la figura 1 donde se invoca al método fábrica ‘createTransport()’ para crear un nuevo tipo de transporte a partir de una superclase.

Diagrama

Descripción generada automáticamente

Figura : Método fábrica para crear el objeto Truck() y Ship(). Recuperado de https://refactoring.guru/es/design-patterns/factory-method

* Builder

Ese patrón surge de una problemática que surge al construir subclases cada vez más especializadas a partir de una superclase: la superclase irá acumulando todos los parámetros posibles, los cuales no siempre serán usados, por lo que un llamado a la clase o creación de objeto será engorroso por todos los parámetros que espera.

La solución que ofrece el patrón builder es la de extraer las propiedades de los objetos de la superclase y crear objetos independientes para cada una de las propiedades. La figura 2 muestra un ejemplo, donde las propiedades la clase ‘casa’ (puertas, ventanas, paredes) han sido asignadas a constructores para su determinación, con esto cuando se quieran establecer el número de paredes o ventanas para una subclase basta con llamar únicamente a los constructores necesarios.

Diagrama

Descripción generada automáticamente

Figura . Patrón builder con los constructores para las propiedades de la clase casa. Recuperado de: https://refactoring.guru/es/design-patterns/builder

* Prototype

Permite copiar o clonar objetos mediante una interfaz (para un prototipo) que contiene un método *clonar,* llevando todos los valores del objeto original al nuevo, copiando incluso atributos privados. Un *prototipo* es entonces un objeto que soporta la clonación. Permite ahorrarse el proceso de creación de un objeto desde cero.

* Singleton

Patrón en el que se crea una clase *Singleton* la cual tiene una única instancia u objeto, al cual se puede acceder desde cualquier parte del programa, pero sin ser sobreescrito. Cuando se llama a esta clase, ella devuelve al objeto (almacenado en caché) o lo crea por primera vez y no permite crear más. Resulta de especial utilidad al garantizar la seguridad de datos almacenados en base, por ejemplo, permite el acceso global a un único objeto de la base de datos, pero sin permitir la creación de mas objetos o la sobreescritura de sus variables globales.

Diagrama

Descripción generada automáticamente

Figura . Representación de una clase Singleton. Recuperado de : https://refactoring.guru/es/design-patterns/singleton

Patrones Estructurales

Son un estándar a la hora de ensamblar objetos y clases en estructuras más grandes.

* Bridge

Patrón que permite dividir a una clase en dos jerarquías o tipos de atributos (estrechamente relacionadas) para que puedan desarrollarse una independiente a la otra. Permite la herencia de atributos de dimensiones independientes extrayendo estas dimensiones restantes en otra jerarquía, en crear subclases para cada una de las combinaciones posibles de las dimensiones.

Un ejemplo de este patrón se puede apreciar en la figura 4, donde para una superclase figuras existen 2 dimensiones: forma y color. En lugar de crear subclases para todas las combinaciones de atributos (ej. ‘CirculoRojo()’) se divide la dimensión de color como una jerarquía independiente, la cual tendrá su herencia particular, por lo que añadir un color nuevo no requerirá modificar la estructura jerárquica de la forma. Este patrón permite implementar la abstracción de la Programación Orientada a Objetos.

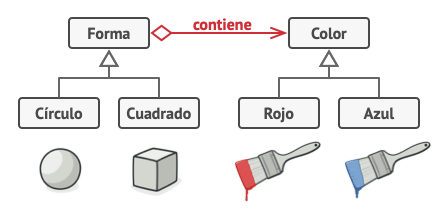


Figura . Ejemplo de patrón bridge, donde se ha establecido un puente entre la dimensión forma y la dimensión color. Recuperado de: https://refactoring.guru/es/design-patterns/bridge

* Decorator

Este patrón permite colocar objetos dentro de otros llamados wrapper (o envoltorios) para así agregarles las funcionalidades de estos. De utilidad cuando no se puedan extender los comportamientos de una clase mediante la herencia. Es usado en arquitecturas por capas al crear un decorador para cada capa.

* Proxy

Permite proporcionar un sustituto o proxy para otro objeto, controlando el acceso al objeto original. Resulta útil cuando el objeto original es pesado y solo se desea acceder a alguna propiedad específica del objeto, por lo que el proxy se encarga de desarrollar este trabajo y emular un comportamiento del objeto original. Un ejemplo se da en la figura 5 donde ‘CreditCard’ es un proxy de ‘Cash’, realizando una función que puede hacer el dinero, pero de manera menos laboriosa.

Diagrama

Descripción generada automáticamente

Figura 5. CreditCard como un proxy de Cash. Recuperado de: https://refactoring.guru/es/design-patterns/proxy

* Adapter

Patrón que incluye uno objeto que adapta la interfaz de un objeto de tal manera que otro pueda comprenderla o trabajar con ella. Un adaptador es útil cuando se quiera hacer uso de una clase o módulo cuya interfaz no sea compatible con el resto del código. Un ejemplo es la figura 6, donde se aprecia un adaptador de información en XML a notación JSON.

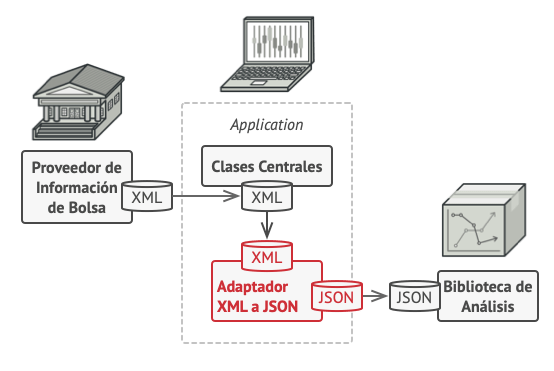


Figura 6. Ejemplo de patrón adapter: adaptador de XML a JSON. Recuperado de: https://refactoring.guru/es/design-patterns/adapter

* Flyweight

Patrón estructural que permite el ahorro de memoria RAM al permitr que la información común a un grupo de objetos (extrínseca) sea guardada en un contenedor enlazado a los objetos, mientras que estos seguirán almacenando cada uno su información única (intrínseca), pasando a llamarse objetos *flyweight o* peso mosca. La figura 7 presenta una clase ‘MovingParticle’ con información extrínseca (común a los objetos) mientras que los objetos *flyweight* de la clase *Particle* mantienen su información distintiva.

Diagrama

Descripción generada automáticamente

Figura 7. Ejemplo de patron flyweight. Recuperado de: https://refactoring.guru/es/design-patterns/flyweight

Patrones de Comportamiento

Se enfocan en la asignación de responsabilidades a distintos componentes garantizando una comunicación efectiva.

* Chain of Responsibility

Patrón que permite el trámite de solicitudes a lo largo de una cadena de manejadores, donde cada manejador tiene una tarea y al recibir una solicitud decide si encargarse de realizarla o pasarla al siguiente manejador. Cada comportamiento deseado del proceso pasa a ser un manejador, el cual va enlazado con el siguiente, por lo que una solicitud viaja por todos los manejadores de la cadena hasta ser resuelta.

Un ejemplo de este patrón puede ser un sistema de autenticación de ingreso de usuario a un sistema, donde existirá una cadena para realizar la comprobación, con un manejador para la verificación de email correctamente escrito, otro para email en base de datos, otro para contraseña y así sucesivamente.

* Template Method

Define la estructura de un algoritmo como superclase, pero permite que las subclases sobrescriban los pasos del algoritmo, pero sin cambiar su estructura. Su aplicación consiste en convertir los pasos del algoritmo en métodos y colocar las llamadas a estos dentro de un método *plantilla*. Para cada subclase se implementan los métodos necesarios y se sobrescriben. Un ejemplo se muestra en la figura 8, donde existe una subclase para extraer datos de archivos y se crean subclases para el tipo de archivo específico, donde cada uno tiene su particularidad para su lectura, por lo que los pasos se sobrescriben.

Diagrama

Descripción generada automáticamente

Figura 8. Subclases para minería de datos. Recuperado de: https://refactoring.guru/es/design-patterns/template-method

* Command

Patrón que convierte cada solicitud enviada al sistema en un objeto, el cual contiene toda la información de la solicitud. Permite almacenar y poner en orden de espera las solicitudes pendientes, así como evitar cuellos de botella cuando una operación no se puede realizar. Favorece la separación de responsabilidades. Un ejemplo del mundo real es cuando en un restaurante los meseros anotan las órdenes en papeles y las envían al cocinero, el cocinero tiene la libertad de priorizar órdenes o realizar dos a la vez, ya que tiene la información completa.

* Mediator

Patrón que restringe las múltiples comunicaciones directas entre objetos al implementar un mediador que redirecciona las llamadas. Evita la comunicación caótica y reduce el acoplamiento entre varios componentes de un programa. La figura 9 presenta a un mediador denominado *Dialog* mediante el cual los demás componentes se pueden comunicar indirectamente.

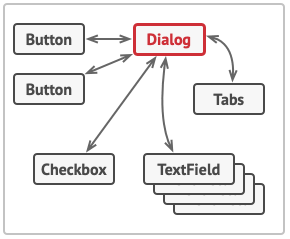


Figura 9. Ejemplo de patrón mediator. Recuperado de https://refactoring.guru/es/design-patterns/mediator

* State

Patrón que permite a los a objetos alterar su comportamiento cuando su estado interno ha cambiado. Su implementación implica crear clases para todos los estados posibles de un objeto, las cuales contienen los comportamientos específicos. Para la transición de uno a otro estado se sustituye un objeto por otro de otra clase. Un ejemplo del mundo real es el comportamiento de un teléfono móvil dependiendo de su estado: cuando está bloqueado no permite llamar a otro número, cuando está desbloqueado permite multitud de comportamientos y cuando se está cargando muestra la pantalla de carga por defecto.

Referencias

Patrones de diseño. *Refactoring.guru*. Recuperado de: *https://refactoring.guru/es/design-patterns*