**Patrones de Diseño**

Principios de diseño:

* *Tome las partes que varían y* ***encapsúlelas*** *de forma tal que luego pueda modificar o extender las partes que varían sin afectar aquellas que no lo hacen.*
* *Programe contra interfaces,* ***no*** *contra implementaciones. Programar contra interfaces quiere decir hacerlo contra un “super tipo”, es decir contra lo más abstracto que pueda.*
* *Favorezca la composición* ***sobre*** *la herencia. “Tiene un” es mejor que “es un”.*

Los patrones:

* Ayudan a decir más con menos
* Hablar a nivel de patrón permite mantenernos “en el diseño” por más tiempo
* Empodera al equipo de trabajo
* Anima a los menos experimentados a ponerse al día

Un patrón es una solución a un problema en un contexto. En detalle

**Contexto:** es la situación para la cual aplica el patrón. Debería ser frecuente

**Problema:** es el objetivo que se quiere alcanzar en el contexto, pero también se refiere a cualquier restricción que ocurra en el contexto

**Solución:** es un diseño general que cualquiera puede aplicar y resuelve el objetivo y sus restricciones.

**Text, chat or text message

Description automatically generated**

**Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application, email

Description automatically generatedCatalogo de Patrones**

**Categoría de patrones**

* **Creacionales:** implica la creación de instancias de objetos y todos proporcionan una forma de **desacoplar** a un cliente de los objetos que necesita instanciar.
* **Comportamiento:** se preocupa por cómo las clases y los objetos interactúan entre sí y distribuyen la responsabilidad.
* **Estructurales:** permiten componer clases u objetos en estructuras más grandes.

**Otras categorías:**

* **Clase:** describen cómo se definen las relaciones entre clases a través de la herencia. Las relaciones en los patrones de clase se establecen en tiempo de compilación.
* **Objeto:** describen relaciones entre objetos y se definen principalmente por composición. Las relaciones en los patrones de objetos se crean normalmente en tiempo de ejecución y son más dinámicas.

****

**Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media**

La banda de los 4 escribión “*Design Pattern – Elements of Reusable Object-Oriented Software”*

**Antipatrón Golden Hammer**

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

**Catálogo de Patrones de Diseño**

**Singleton (creacional)**

**Propósito:** garantiza que una clase solo tenga una instancia y proporciona un punto de acceso global a ella.

**Solución:** la propia clase es responsable de crear la única instancia. Permite el acceso global a dicha instancia mediante un método de clase y declara el constructor como privado para que no sea instanciable directamente.

**Consecuencias:**

* Acceso controlado
* Se logra el objetivo, pero a cambio de ensuciar la clase
* Se lo considera un patrón fácil pero intrusivo

**Factory Method (creacional)**

**Propósito:** proporciona una interfaz para crear un objeto, pero delega en sus hijas la decisión de qué objeto instanciar.

**Solución:**

* Crear una clase Factory/Creator que sea abstracta que provea una interfaz común para la creación del objeto en cuestión
* Crear clases Factories que hereden de la clase Factory abstracta y que implementen **EL** método definido creando la instancia concreta
* En la aplicación utilizar solo la clase FactoryAbstracta / Creator una vez determinada el objeto a crear.

**Consecuencias:**

* Indepndencia de las clases concretas
* Permite intercambiar el objeto creado de manera rápida y transparente

**Multiton (2..n) (creacional)**

* Idem singleton, pero con múltiples instancias

Diagram

Description automatically generated

Diagram

Description automatically generated

**Abstract Factory (creacional)**

**Propósito:** Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.

**Solución:**

* Crear una clase Factory que sea abstracta que provea una interfaz común de creación de las familias de objetos
* Crear clases Factories que hereden de la clase Factory abstracta y que implementen los métodos definidos creando las instancias concretas.

**Consecuencias:**

* Independencia de las clases concretas
* Permite intercambio de familias de objetos de manera rápida y transparente

**Diagram

Description automatically generated**

**Command (organización del trabajo)**

**Propósito:** Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de peticiones, y poder deshacer operaciones. Desacopla el código que solicita un servicio del que lo presta.

**Solución:**

* Crear una clase abstracta o una interfaz con un solo método execute()
* Cada clase descendiente implementará el método
* Para invocar el método se instanciará una de las clases y se invocará al método execute()

**Motivación:**

* Objetos como botones y menús que realizan una petición en respuesta a una entrada de usuario
* Transacciones

**Consecuencias:**

* Permite mantener referencias a métodos
* Desacopla el objeto que invoca la operación de aquel que sabe como realizarla
* Las órdenes son objetos de primera clase. Pueden ser manipulados y extendidos como cualquier otro objeto
* Se pueden ensamblar órdenes en una orden compuesta
* Es fácil añadir nuevas órdenes, ya que no hay que cambiar las clases existentes

**Diagram

Description automatically generated**

**Proxy (control de acceso)**

**Propósito:** Proporciona un representante o sustituto de otro objeto para controlar el acceso a este.

**Solución:**

* Crear una jerarquía en la que intervengan el objeto original y el objeto proxy
* En el objeto proxy habrá una referencia al objeto original
* Se redefinen todas las llamadas en el proxy incorporando código antes de derivarlas al objeto original

**Motivación:**

* Retrasar el coste de creación e inicialización hasta que realmente sea necesario

**Consecuencias:**

* Posibilidad de agregar funcionalidad de manera transparente a la aplicación
* Permite realizar optimizaciones, ocultar complejidad, establecer mecanismos de seguridad de los accesos

**Facade (control de acceso)**

**Propósito:** proporciona una interfaz unificada para un conjunto de interfaces de un subsistema y define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar

**Motivación:**

* Minimizar la comunicación y dependencias entre subsistemas

**Aplicabilidad:**

* Queramos proporcionar una interfaz simple para un subsistema omplejo
* Cuando haya muchas dependencias entre los clientes y las clases que implementan una abstracción
* Queremos dividir en capas nuestros subsistemas

**Beneficios:**

* Oculta a los clientes los componentes del subsistema, reduciendo así el número de objetos con los que tratan los clientes y haciendo que el subsistema sea más fácil de usar
* Promueve un débil acoplamiento entre el subsistema y sus clientes
* No impide que las aplicaciones usen las clases del subsistema en caso de que sea necesario. De este modo se puede elegir entre facilidad de uso y generalidad.

**Strategy (variación de servicios)**

**setStrategy viene por parámetro**

**Necesidad:** un mismo objeto debe poder tener un comportamiento que debe ser determinado en tiempo de ejecución.

**Solución:**

* Delegar el comportamiento en otro objeto
* Armar una jerarquía con los diferentes comportamientos
* **Inyectar** el comportamiento al objeto a través de un método o de su constructor

**Consecuencias**

* Se eliminan los condicionales
* Se crea una jerarquía paralela a la jerarquía base
* Los comportamientos quedan agrupados por familias
* A veces no es tan fácil aislar el comportamiento
* A veces no alcanza

**Template (variación de servicios)**

**Propósito:** define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura

**Aplicabilidad:**

* Para implementar las partes de un algoritmo que no cambian y dejar que sean las sublcases quienes implementen el comportamiento puede variar
* Cuando el comportamiento repetido de varias subclases debería factorizarse y ser localizado en una clase común para evitar código duplicado
* Para controlar las extensiones de las sublcases

**Beneficios:**

* Son una técnica fundamental de reutilización de código
* Extraen el comportamiento común de las clases de la biblioteca
* “Principio de Hollywood”, una clase padre llama a las operaciones de una subclase y no al revés
* Operaciones de enganche, propocionan el comportamiento predeterminado que puede ser modificado por las subclases

**State (variación de servicios)**

**por contexto**

**Propósito:** permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto

**Motivación:** conexión TCP, que presenta 3 estados: establecida, escuchando, cerrada

**Aplicabilidad:**

* El comportamiento de un objeto depende de su estado, y debe cambiar en tiempo de ejecución dependiendo de ese estado
* Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. Este estado se suele representar por una o más constantes enumeradas

**Beneficios:**

* Localiza el comportamiento dependiendo del estado y divide dicho comportamiento en diferentes estados
* Hace explícitas las transiciones entre estados
* Los objetos estado pueden compartirse

**Composite (Descomposición estructural)**

**Propósito:** compone objetos en estructuras árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos

**Motivación:** las aplicaciones gráficas como los editores de dibujo y los sistemas de diseño permiten agrupar componentes simples en más grandes. Manejo de excepciones

**Aplicabilidad:**

* Quiera representar jerarquías de objeto parte-todo
* Quiera que los clientes sean capaces de obviar las diferencias entre composiciones de objetos y los objetos individuales. Los clientes tratarán a todos los objetos de la estructura compuesta de manera uniforme.

**Beneficios:**

* Define jerarquías de clases formadas por objetos primitivos y compuestos
* Simplifica el cliente
* Facilita añadir nuevos tipos de componentes

**Desventajas:**

* Puede hacer que un diseño sea demasiado general

**Decorator (Extensión de Servicios)**

**Propósito:** asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad

**Motivación:** interfaces de usuarios a las que se agregan propiedades o comportamientos y cambios de piel

**Aplicabilidad:**

* Para añadir objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos
* Para responsabilidades que pueden ser retiradas
* Cuando la extensión mediante herencia no es viable

**Beneficios:**

* Más flexibilidad que la herencia estática
* Evitar clases cargadas de funciones en la parte de arriba de la jerarquía

**Desventajas:**

* Un decorador y su componente no son idénticos
* Muchos objetos pequeños