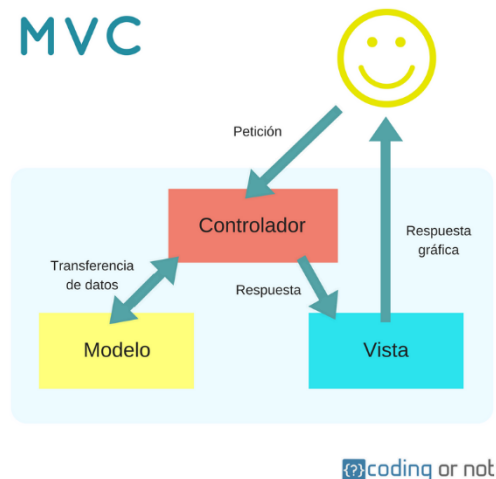


PATRONES CREACIONALES

Patrones de diseño de software que solucionan problemas de creación de instancias.

- Builder (Constructor virtual). **Abstrae** el **proceso de creación** de un objeto complejo, centralizando dicho proceso en un **único punto**.
- Abstract **Factory** (Factoría abstracta). Permite trabajar con objetos de distintas familias de manera que **las familias no se mezclan entre si y haciendo transparente el tipo de familia concreta que se está usando**. [Ejemplo de SuperMario Maker (O **todo es de Nintendo DS o Todo es de Gameboy pero no se puede entremezclar los ítems**) Protegen de utilizar ciertos objetos que no deberían utilizarse juntos. **No se debe mostrar el tipo de clase que estás usando**. [Abstrae distintas familias [fábricas]]
- **Factory Method** (Método de fabricación). **Centraliza en una clase constructora la creación de objetos de un subtipo determinado**, ocultando al usuario la casuística. (Utilizando polimorfismo [Ejemplo **Enemigos de un Juego de la Implementación Enemigos**] **BettaTech**) [Única fábrica que construye objetos] [Unas fábricas que construye un método]
- Prototype (**Prototipo**). Crea **nuevos objetos clonándolos de una instancia ya existente**. [Definición -> **Primer ejemplar que se fabrica** de una figura, un invento u otra cosa, y que **sirve de modelo** para **fabricar otras iguales**]
- Singleton (Instancia única). **Garantiza la existencia** de una **única instancia** para una clase y la creación de un **mecanismo de acceso global** a dicha **instancia**.
- MVC. (Modelo vista controlador).
 - Este patrón plantea la separación del problema en tres capas: la capa **model**, que **representa la realidad**; la capa **controller**, que **conoce los métodos y atributos del modelo, recibe y realiza** lo que el **usuario quiere hacer**; y la capa **vista**, que muestra un **aspecto del modelo** y es utilizada por la capa anterior para interactuar con el usuario.



PATRONES ESTRUCTURALES

Patrones de diseño que solucionan problemas de **composición** (**agregación**) de **clases** y **objetos**.

- Adapter o Wrapper. **Adapta** una **interfaz** para que **pueda ser utilizada** por **una clase** que de otra forma no podría utilizarla. [Ejemplo Distancia Pies – Metros (Clase Adapte de intermediaria)]

- Proxy (**Apoderado**). Proporciona un intermediario de un objeto para **controlar** su **acceso**. Clase Intermediaria que **controla** (filtra), (no adapta). [Ejemplo correo electrónico a una sola versión] [Apodera de la clase y la controla por eso utiliza filtros]
- Facade (Fachada). Provee a una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema. [Ofrece una facha que simplifica el conjunto de interfaces que hay por detrás (no resuelve nada, pero lo hace todo más simple)]
- Bridge. **Desacopla** una abstracción de su implementación. [Divide las implementaciones en elementos individuales y **establece un puente entre las dos**, de forma que puedas utilizar 1 implementación, la otra o todas]. [Ejemplo de celdas con información e imágenes que nos obligaba a tener estos dos componentes]
- **CompOsite**. Permite tratar objetos compuestos como si de uno simple se tratase. [Array de ObjetOs] [Nave y flota de naves, **permite manipular** “una flota de naves” como si de una “nave” se tratase. Crea una especie de arreglo con funciones iguales a los elementos individuales. Permite hacer colecciones de colecciones, es decir “flotas de flotas de naves”]
- Decorator. **Añade funcionalidad a una clase dinámicamente** (en tiempo de ejecución) [Parecido a composite, pero no quiere agrupar opciones sino **seleccionarlas**] [Evita crear clases para cada funcionalidad, en lugar de eso, crea una clase con las adiciones]. [Ejemplo de armas de juegos, tipo de mira, textura... (**Dinámicamente**)]
- Flyweight (**Peso ligero**). Reduce la **redundancia** cuando gran cantidad de objetos poseen idéntica información.
- Module. **Agrupar** varios **elementos** relacionados, como clases, singletons y métodos, utilizados globalmente, **en una entidad única**. [Los agrega juntos a todos, en un gran module]

PATRONES DE COMPORTAMIENTO

Se define como patrones de diseño de software que ofrecen **posibilidades respecto** a la **interacción** y **responsabilidades entre clases** y objetos, así como los algoritmos que encapsulan.

- **Chain of Responsibility** (Cadena de responsabilidad). Permite establecer la línea que deben llegar los mensajes para que los objetos realicen la tarea indicada.
- Command (orden). **Encapsula una operación en un objeto**, permitiendo ejecutar dicha operación sin necesidad de conocer el objeto de la misma. [Refresco en un vaso opaco, estas bebiendo, pero no sabes a ciencia cierta el que] [Envoltorio]
- **Interpreter**. Dado un lenguaje, define una gramática para dicho lenguaje, así como las herramientas necesarias para interpretarlo.
- **Iterator**. Permite realizar recorridos sobre objetos compuestos independientemente de la implementación de estos. [Recorre objetos independientemente de lo que contenga]
- **Mememento** (Recuerdo). **Permite volar a estados anteriores del sistema**.
- **State** (Estado). Permite que un objeto modifique su comportamiento cada vez que **cambie** su estado interno. [Evento de modificación] [Ejemplo boss subiendo de nivel, cambiando su forma, sus ataques...]
- **Strategy** (Estrategia) Permite disponer de varios métodos para resolver un problema y elegir cual utilizar en tiempo de ejecución. [Varias estrategias en tiempo de ejecución]
- **Template Method** (**Método plantilla**) Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos, esto **permite** que las

subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura. [Plantilla de Word, aprovecho la estructura para redefinir una mejor]

- **Visitor.** Define nuevas operaciones sobre una jerarquía sin modificar las clases sobre las que operan. [El visitante define, pero no modifica la clase] [Turista, añades cosas “como mucho”, pero no tocas (modificas)] [Similar a un decorador]
- **Observer.** Define una dependencia de uno o varios entre objetos de manera que cuando uno cambia su estado, todos los objetos dependientes son notificados y actualizados automáticamente. [**Observador de clases dependientes**]
- **Mediator.** Permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, **forzándolas a colaborar a través de un objeto mediator**.