**Patrones de diseño orientado a objetos:**

\* Diseñar software orientado a objetos es **difícil**. Y diseñar software orientado a objetos **reutilizables** es todavía **más difícil**.

\* Un software capaz de evolucionar tiene que ser reutilizable (Al menos para las versiones futuras).

Hay que **preparar software** que esté diseñado para que en algún momento **se** **pueda** **cambiar**. Al momento de **codear** hay que pensar cual es la mejor solución a ese problema, para cuando que el cliente requiera un cambio, no reescribir el código ni romper lo que ya funciona.

**Para ser bueno, hay que copiar, entender, pegar, y modificar en base a nuestras necesidades.**

**¿Como se clasifican los patrones? -> Hay 3**

**Patrones arquitecturales:** Expresan cómo armar un proyecto, como estructurarlo. Expresan un paradigma fundamental para estructurar un sistema de software. Proporcionan un conjunto de subsistemas predefinidos, con reglas y guías para organizar las relaciones entre ellos.

**Patrones de diseño:** Compuestos de varias unidades arquitecturales más pequeñas. Describen el esquema básico para estructurar subsistemas y componentes.

- **Patrones de creación:** Inicialización y configuración de las clases y/o objetos.

**- Patrones de estructurales:** Trata de desacoplar varias porciones de código y dividirlo en distintas clases.

- **Patrones de comportamiento:** Permiten cambiar dinámicamente la lógica de los objetos o clases de acuerdo a un tipo de dato. Hay una lógica genérica que se adapta al contexto.

**Patrones elementales:** (COMPLETAR:

**Singleton – (Creacional):**

Es un objeto, que tiene una única instancia. Existe una única instancia. No puede haber 2 objetos de esa misma clase. Se va a ir reutilizando para el resto. Por eso nosotros (en Angular) cuando creamos un servicio, poníamos en el constructor el nombre del servicio, y accedíamos a los métodos sin hacer el new.

// Para que tenga un único ejemplar, NO hay

// que hacer un NEW de este objeto.

// --------------------- 1 ----------------------

// Hay que crear un atributo estático de servicio.

private static Servicio s;

// --------------------- 2 ----------------------

// Hay que hacer privado el constructor.

// Para que NADIE pueda crear una instancia por fuera.

private Servicio() {}

// --------------------- 3 ----------------------

// Hacemos un método que, si la instancia es nula, la crea. Y sino, la devuelve.

public static Servicio getInstance() .

{ .

. if (s == null) .

. { .

. s = new Servicio(); .

. } .

. .

. return s; .

. .

} .

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Factory:** Propósito: Crear distintos objetos de la misma naturaleza en tiempo de ejecución.

**Motivación:** Es responsable de crear el objeto que necesitamos en tiempo de ejecución. En principio el factory no conoce las clases concretas que se van a definir en una aplicación concreta. El método factory es un método abstracto que encapsula el conocimiento de qué subclase se va a crear y pone este conocimiento fuera del armazón.

Aplicación: