



Patrones de diseño

Resolución de problemas

- Nos enfrentamos a un problema.
- Analizamos el problema.
- Buscamos una solución.
- Si la solución cumple con los
- requerimientos, ¿es una buena solución?

Resolución de problemas

- Nos enfrentamos a un problema.
- Analizamos el problema.
- Buscamos una solución.
- Si la solución cumple con los
- requerimientos, ¿es una buena solución?





Diseños Erróneos



Los cuatro jinetes de la programación

Rigidez

¿Alguna vez ha tenido que hacer un “cambio” simple en el código que después de alguna investigación pareció ser una enorme cadena de otros cambios? Podrías decir "eso es fácil, solo me tomará varias horas hacerlo". ¡Pero después de un poco de trabajo te das cuenta de que no te llevó horas sino días! ¡Nos gustaría cortar esas cadenas de cambio lo más rápido posible! Este problema puede ser especialmente visible cuando la aplicación no está dividida en capas o niveles distintos y cada módulo habla con todos los demás.

Fragilidad



Después de hacer un cambio simple (incluso si le tomó varios días en lugar de horas), pueden suceder otras cosas inesperadas. Ese pequeño fragmento de código modificado podría influir en otra parte de la aplicación y es posible que obtenga errores muy extraños que parecen no estar relacionados con el problema original. El código es tan "frágil" que se deshace después de una pequeña modificación.

Inmovilidad

Deberíamos escribir nuestro código con el fin de usarlo en algún otro software. Nuestro código debe ser "móvil" y utilizable por otros.

Desafortunadamente, suele ser solo una teoría y en realidad las cosas pueden ir en sentido contrario. Esa gran clase que acaba de escribir es tan difícil de usar y de mover, que es más fácil volver a escribirla desde cero. Cierta código, por supuesto, no se puede mover de una aplicación a otra; por ejemplo, la lógica empresarial para un problema en particular: en la mayoría de los casos es único. Pero seguro, tiene algunas utilidades, algún código general, código que maneja las reglas comerciales básicas, etc. que pueden y deben usarse en su proyecto futuro. Si tiene "miedo" de usar código de sus proyectos anteriores ... eso no es bueno.

Viscosidad

La viscosidad se presenta en dos formas: viscosidad del software y viscosidad del entorno.

Cuando se enfrentan a un cambio, los desarrolladores suelen encontrar más de una forma de realizar ese cambio. Algunas de las formas conservan el diseño; otros no lo hacen (es decir, que son hacks). Cuando los métodos de preservación de diseño son más difíciles de emplear que los hacks, la viscosidad del diseño es alta. Es fácil hacer lo incorrecto, pero difícil hacer lo correcto. Queremos diseñar nuestro software de manera que los cambios que preservan el diseño sean fáciles de realizar.

Viscosidad

La viscosidad del entorno se produce cuando el entorno de desarrollo es lento e ineficiente. Por ejemplo, si los tiempos de compilación son muy largos, los desarrolladores se verán tentados a realizar cambios que no obliguen a grandes recompilaciones, aunque esos cambios no conserven el diseño. Si el sistema de control del código fuente requiere horas para registrar solo unos pocos archivos, entonces los desarrolladores se verán tentados a realizar cambios que requieran la menor cantidad posible de registros, independientemente de si se conserva el diseño.

En ambos casos, un proyecto viscoso es un proyecto en el que el diseño del software es difícil de preservar. Queremos crear sistemas y entornos de proyectos que faciliten la conservación del diseño.

¿Qué podemos hacer para no caer en estos diseños
erróneos?

Ó ¿que existe para ayudarnos?

Patrones de diseño

- “Son descripciones de clases y objetos relacionados que están adaptados para resolver un problema de diseño general en un contexto determinado”.

Erich Gamma, Richard Helm, John Vlissides y Ralph Johnson

Patrones de diseño

- “*Design Patterns: Elements of Reusable Object Oriented Software*” de 1994
- Catalogan y describen patrones existentes dentro de la comunidad del software
- Catalogan 23 patrones
- Destacan estrategias y aproximaciones basadas en el diseño de patrones

Clasificación de patrones (GoF)

- **Patrones de creación:** Tratan la inicialización y configuración de clases y objetos.
- **Patrones estructurales:** Tratan de desacoplar interfaz e implementación de clases y objetos.
- **Patrones de comportamiento:** Tratan las interacciones dinámicas entre clases y objetos

Patrones de creación

- Abstraen el proceso de instanciación haciendo al sistema independiente de como los objetos son creados, compuestos o representados.
- Alternativas de diseño por herencia.
- Encapsulan el mecanismo de creación.
- Independencia de los tipos de objetos “producto” que manejamos

Patrones de creación

- **The Factory Method** retorna una de las posibles subclases de una clase abstracta dependiendo de los datos que se le provee.
- **The Abstract Factory Method** retorna una de las varias familias de objetos.
- **The Builder Pattern** separa la construcción de un objeto complejo de su representación. Varias representaciones se pueden crear dependiendo de las necesidades del programa.
- **The Prototype Pattern** inicializa e instancia una clase y luego copia o clona si necesita otras instancias, mas que crear nuevas instancias.
- **The Singleton Pattern** es una clase de la cual no puede existir mas de una instancia.

Patrones estructurales

- Determinan como combinar objetos y clases para definir estructuras complejas.
 - Comunicar dos clases incompatibles,
 - Añadir funcionalidad a objetos

Patrones estructurales

- **Adapter:** cambia la interfaz de una clase a la de otra.
- **Bridge:** permite mantener constante la interfaz que se presenta al cliente, cambiando la clase que se usa
- **Composite:** una colección de objetos
- **Decorator:** una clase que envuelve a una clase dándole nuevas capacidades.
- **Facade:** reúne una jerarquía compleja de objetos y provee una clase nueva permitiendo acceder a cualquiera de las clases de la jerarquía.
- **Flyweight:** permite limitar la proliferación de pequeñas clases similares.

Patrones de comportamiento

- Se ocupan de los algoritmos y reparto de responsabilidades.
- Describen los patrones de comunicación entre objetos y clases.
- Podremos definir abstracciones de algoritmos (Template Method)
- Cooperaciones entre objetos para realizar tareas complejas, reduciendo las dependencias entre objetos (Iterator, Observer, etc.)
- Asociar comportamiento a objetos e invocar su ejecución (Command, Strategy, etc.)

Patrones de comportamiento

- **Cadena de responsabilidad:** permite que un conjunto de clases intenten manejar un requerimiento.
- **Interpreter:** define una gramática de un lenguaje y usa esa gramática para interpretar sentencias del lenguaje.
- **Iterator:** permite recorrer una estructura de datos sin conocer detalles de cómo están implementados los datos
- **Observer:** algunos objetos reflejan un cambio a raíz del cambio de otro, por lo tanto se le debe comunicar el cambio de este último.
- **Strategy:** cantidad de algoritmos relacionados encerrados en un contexto a través del cual se selecciona uno de los algoritmos.

¿Son todos los patrones existentes?

Otros tipos de patrones

- Patrones de programación concurrente
- Patrones de interfaz gráfica
- Patrones de organización de código
- Patrones de optimización de código
- Patrones de robustez de código
- Patrones guiados por prueba

¿Por qué usar Patrones de diseño?

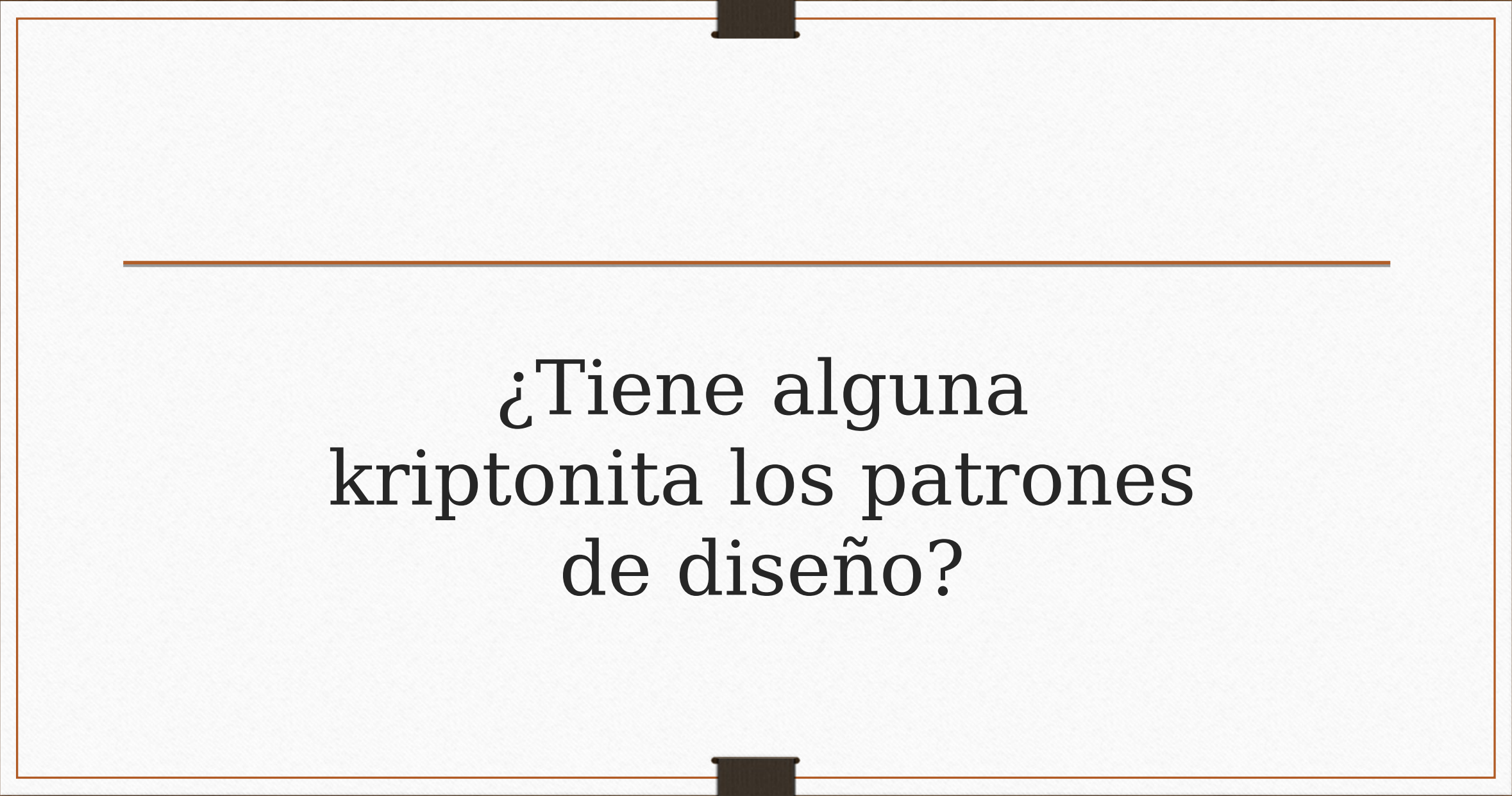
- Se definen con un alto nivel de abstracción.
- Son independientes de los lenguajes de programación y de los detalles de implementación.
- Proporciona un esquema para refinar los subsistemas o componentes de un sistema de software, o las relaciones entre ellos.

Beneficios de los patrones

- Los patrones favorecen la reutilización de diseños y arquitecturas a gran escala.
- Capturan el conocimiento de los expertos y lo hacen accesible a toda la comunidad software.
- Establecer terminología común. Los nombres de los patrones forman parte del vocabulario técnico del ingeniero software.

Y más importante

Usando patrones de diseño logramos que el software tenga una buena estructura que sea **flexible, mantenible y reutilizable**



¿Tiene alguna
kryptonita los patrones
de diseño?

Problema de los patrones

- La integración de los patrones en el proceso de desarrollo se hace todavía de forma manual.
- El número de patrones identificados es cada vez más grande. Las clasificaciones actuales no siempre sirven de guía para decidir cual usar.
- El número de combinaciones patrones estilos y atributos que se dan en la práctica son incontables.
- Los patrones se validan por la experiencia y el debate, no mediante la aplicación de técnicas formales

¿Que es lo único constante en el proceso de desarrollo de software?

Cambios en el proceso de desarrollo

Los cambios en un diseño de software, normalmente si no fueron cambios previstos en el diseño original, degradan el mismo. Incluyen dependencias.

- Generalmente lo hacen personas que no estaban relacionados con la filosofía de diseño original.

¿Cómo usar un patrón de diseño?

1. Tener una visión general de los patrones.
2. Volver y estudiar la estructura del patrón a implementar.
3. Ver un ejemplo concreto codificado del patrón.
4. Elegir nombres significativos en el contexto del patrón.
5. Implementar el patrón.

-
- “Una cosa que los diseñadores expertos no hacen es resolver cada problema desde el principio [...]. Cuando encuentras una buena solución la usas una y otra vez. Esta experiencia es lo que los hace expertos”

- Gamma, 1995



Patrones de diseño
