POLITÉCNICA

"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

Patrones de Diseño

2. Definición

Luis Fernández Muñoz

https://www.linkedin.com/in/luisfernandezmunyoz

setillofm@gmail.com

INDICE

1. Definición

"Cada patron describe un problema el cual ocurre una y otra vez en nuestro entorno y describe el núcleo de una solución para el problema, de tal forma que se puede usar esta solución un millón de veces sin hacerlo de la misma forma dos veces" [Alexander]

- Nuestras soluciones se expresan en términos de objetos e interfaces en vez de paredes y puertas pero el núcleo de ambas clases de patrones es **una solución a un problema en un contexto**
- Son la descripción de la comunicación de objetos y clases que son particularizados para resolver un problema de diseño general en un contexto particular

1. Elementos esenciales

- El **nombre** del patrón es una herramienta que podemos utilizar para describir un problema de diseño, sus soluciones y las consecuencias con una o dos palabras.
  - El nombre del patrón aumenta inmediatamente nuestro vocabulario de diseño.
  - Nos permite diseñar a un nivel más alto de abstracción.
  - Tener un vocabulario para los patrones nos permite hablar de ellos con nuestros colegas, en nuestra documentación, e incluso a nosotros mismos.
  - Esto hace que sea más fácil pensar en diseños y comunicarlos con sus compromisos a los demás.

1. Elementos esenciales

- El **problema** se describe cuándo aplicar el patrón.
  - Explica el problema y su contexto.
  - Podría describir los problemas específicos de diseño tales como la forma de representar algoritmos como objetos.
  - Podría describir las estructuras de clases u objetos que son sintomáticos de un diseño inflexible.
  - A veces el problema incluirá una lista de condiciones que se deben cumplir antes de que tanga sentido aplicar el patrón

1. Elementos esenciales

- La **solución** describe los elementos que llevan a cabo el diseño, sus relaciones, responsabilidades y colaboraciones.
  - La solución no describe un diseño o implementación concreta particular porque un patrón es como una plantilla que puede ser aplicada en muchas situaciones diferentes
  - En lugar de ello, el patrón proporciona una descripción abstracta de un problema de diseño y cómo la estructura general de elementos resuelve (clases y objetos en nuestro caso).

1. Elementos esenciales

- Las **consecuencias** son los resultados, las ventajas y desventajas de aplicar el patrón.
  - Aunque las consecuencias son a menudo silenciosas cuando describimos las decisiones de diseño, son fundamentales para la evaluación de alternativas de diseño y para la comprensión de los costes y beneficios de aplicar el patrón.
  - Las consecuencias para el software a menudo se refieren a compromisos de espacio/tiempo. Pueden referirse a cuestiones del lenguaje y de ejecución.
  - La reutilización es a menudo un factor en el diseño orientado a objetos, las consecuencias de un patrón incluyen su impacto en la flexibilidad, extensibilidad o portabilidad de un sistema.
  - El listado explícito de estas consecuencias ayuda a comprender y evaluarlos.

2. Elementos

- **Nombre:**
  - **Nombre**. El nombre del patrón transmite la esencia del patrón de manera sucinta. Un buen nombre es vital, porque se convertirá en parte de su vocabulario de diseño.
  - **Sinónimos**. Otros nombres conocidos para el patrón, si los hubiere.

2. Elementos

- **Problema:**
  - **Intención**. Una breve declaración que responde a las siguientes preguntas: ¿Qué hace el patrón de diseño? ¿Cuál es su razón de ser e intención? ¿Qué problema de diseño en particular o problema aborda?
  - **Motivación**. Un escenario que ilustra el problema de diseño y cómo las estructuras de clases y objetos en el patrón resuelven el problema. El escenario ayuda a entender la descripción más abstracta del patrón.
  - **Aplicabilidad**. ¿Cuáles son las situaciones en las que el patrón de diseño se puede aplicar? ¿Cuáles son ejemplos de diseños pobres al que el patrón puede hacer frente? ¿Cómo se puede reconocer estas situaciones?

2. Elementos

- **Solución:**
  - **Estructura**. Una representación gráfica de las clases en el patrón utilizando una notación basada en [... UML]
  - **Participantes**. Las clases y/u objetos que participan en el patrón de diseño y sus responsabilidades.
  - **Colaboraciones**. Cómo los participantes colaboran para llevar a cabo sus responsabilidades.
  - **Implementación**. ¿Qué trampas, pistas o técnicas se deben tener en cuenta al aplicar el patrón? ¿Hay problemas específicos del lenguaje?

2. Elementos

- **Solución:**
  - **Código de ejemplo**. Fragmentos de código que ilustran cómo se puede implementar el patrón en C ++ y Smalltalk.
  - **Usos conocidos**. Ejemplos del patrón encontrado en los sistemas reales. Se incluye al menos dos ejemplos de diferentes dominios.
  - **Patrones relacionados**. ¿Qué patrones de diseño están estrechamente relacionados con éste? ¿Cuáles son las diferencias importantes? ¿Con que otros patrones se debe utilizar éste?

2. Elementos

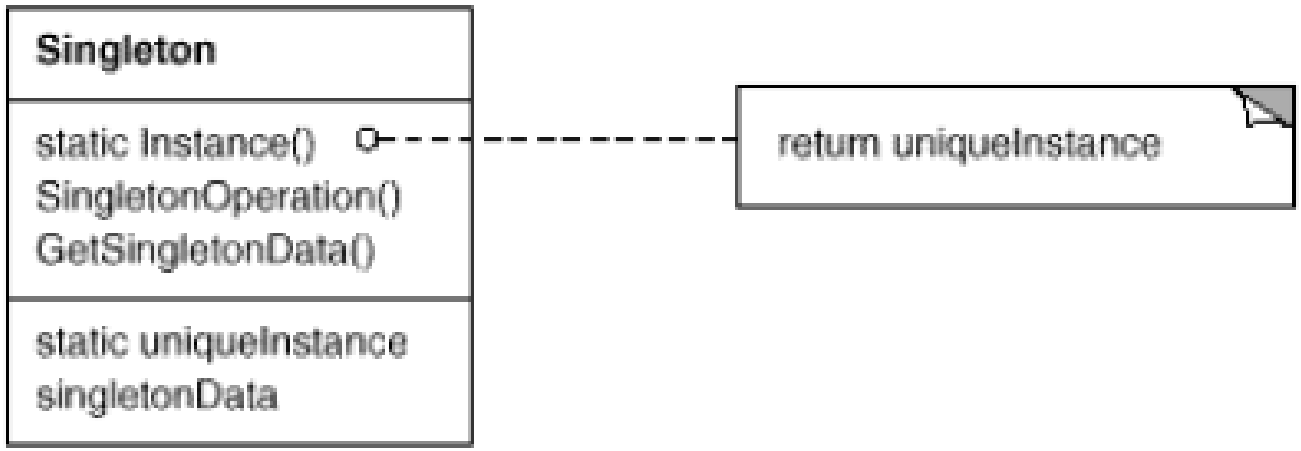
- **Consecuencias:**
  - **Consecuencias**. ¿Cómo apoya el patrón a sus objetivos? ¿Cuáles son los compromisos y resultados de la utilización del patrón? ¿Qué aspectos de la estructura del sistema permite variar de forma independiente?

3. Ejemplo: *Singleton* (*Único*)

- *Nombre*. ***Singleton***
- *Sinónimos*.
- *Intención*. Asegurar que una clase tenga una sola instancia y proveer un punto de acceso global para ésta.
- *Motivación*. Para algunas clases es importante tener exactamente una instancia. Aunque haya muchas impresoras en un sistema, debería haber una única cola de impresión. Debería haber un solo sistema de ficheros y un solo gestor de ventanas. Un filtro digital tendría un único convertidor analógico/digital. Un sistema de cuentas bancarias será dedicado para servir a una compañía

3. Ejemplo: *Singleton* (*Único*)

- *Aplicabilidad*.
  - Debe ser exactamente una única instancia de la clase y debe ser accesible a los clientes desde un punto de acceso bien conocido.
  - Cuando la única instancia debería ser extensible por sub-clasificación y los clientes deberían ser capaces de usar la instancia extendida sin modificar su código
- *Estructura*.



3. Ejemplo: *Singleton* (*Único*)

- *Participantes*.
  - *Singleton* define una operación *Instance* que permite a los clientes acceder a su única instancia. *Instance* es una operación de clase (o sea, un método de clase en Smalltalk y un función miembro estática en C++)
  - *Singleton* puede ser responsable de crear su propia instancia única
- *Colaboraciones*. Los clientes acceden a la instancia única a través de la operación *Instance* de *Singleton*
- *Consecuencias*.
  - *Acceso controlado a una única instancia.* Como la clase encapsula su única instancia, puede tener un control estricto sobre cómo y cuándo los clientes acceden a ésta.

3. Ejemplo: *Singleton* (*Único*)

- *Consecuencias*.
  - *Reducir el espacio de nombres.* Es una mejora sobre las variables globales ya que se evita contaminar el espacio de nombres con las variables globales que almacenen las instancias únicas.
  - *Permite el refinamiento de operaciones y su representación.* Se pueden tener subclases y es fácil configurar una aplicación con una instancia de esta clase extendida que se necesite en tiempo de ejecución.
  - *Permite un número variable de instancias.* Facilita cambiar de opinión y permitir más de una instancia de la clase. Además, se puede utilizar el mismo enfoque para controlar el número de instancias que utiliza la aplicación.
  - *Más flexible que operaciones de clase (static).* Esta técnica hace que sea difícil cambiar un diseño para permitir más de una instancia de una clase. Y las subclases no pueden redefinirlas polimórficamente

3. Ejemplo: *Singleton* (*Único*)

- *Implementación*. **Asegurar una instancia única.** No es suficiente para definir el *Singleton* un objeto global o estático y luego confiar en la inicialización automática.
  - No podemos garantizar que sólo una instancia de un objeto estático será declarado
  - Puede que no tengamos suficiente información para crear una instancia del *Singleton* en el momento de la inicialización estática. Un *Singleton* podría requerir valores que se calculan más adelante en la ejecución del programa.
  - C ++ no define el orden en el que los constructores de objetos globales se denominan en todas las unidades de traducción. Esto significa que no pueden existir dependencias entre *Singleton;* si alguno las hay, entonces los errores son inevitables.
  - Obliga a todos los *Singleton* que se creen tanto si se utilizan o no.

3. Ejemplo: *Singleton* (*Único*)

- *Implementación*. **Sub-clasificando la clase *Singleton*.**
  - La técnica más simple es determinar qué *Singleton* se requiere usar en la operación *Instance* del *Singleton*. Un ejemplo para implementar esta técnica sería con variables de entorno.
  - Otra forma de elegir la subclase del *Singleton* es poner la implementación de *Instance* fuera de la clase padre (p.e. Factoría) y ponerla en la subclase. Esto permite al programador de C++ decidir la clase de *Singleton* en tiempo de compilación (p.e. enlazando a un fichero objeto que contenga la implementación) pero se mantiene oculto desde los clientes del *Singleton*.
  - El enfoque de enlace fija la elección de la clase de *Singleton* en tiempo de compilación, lo cual hace difícil la elección en tiempo de ejecución. Usando sentencias condicionales para determinar la subclase es más flexible, pero con fuertes conexiones al conjunto de posibles clases *Singleton*.

3. Ejemplo: *Singleton* (*Único*)

- *Implementación*. **Subclasificando la clase *Singleton*.**
  - Un enfoque más flexible usa un registro de *Singletons*, donde registrar sus instancias por nombre en un registro bien conocido. El registro aplica nombres a *Singletons*. Cuando *Instance* necesita un *Singleton*, consulta al registro solicitando el *Singleton* por nombre. El registro busca el correspondiente *Singleton* (si existe) y lo devuelve. Este enfoque libera a *Instance* de conocer todas las posibles clases u objetos de *Singleton*. Todo esto requiere un interfaz común para todos las clases *Singleton* que incluye operaciones para el registro. Para mantener el registro sencillo, tendremos que almacenar una lista de objetos de *NameSingletonPair*. Se asume que una variable de entorno especifica el nombre del *Singleton* deseado. ¿Dónde se registran las clases *Singleton*? Una posibilidad es en su constructor y definiendo instancias estáticas.

3. Ejemplo: *Singleton* (*Único*)

- *Código ejemplo*.

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // existing interface goes here
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};
```

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

- *Usos conocidos*. *ChangeSet* int *Smalltalk-80*, *Session* and *WidgetKit* en *InterViews*.
- *Patrones relacionados*. *Abstract Factory*, *Builder*, y *Prototype*.