Trabajo Práctico Nro 1.4 (01/Abril/2020)

Patrón de Diseño: Bridge

Nombre y Apellido : Aranda, Renzo Gabriel

DNI: 40.178.023

Objetivo:

Bridge es un patrón de diseño estructural que le permite dividir una clase grande o un conjunto de clases estrechamente relacionadas en dos jerarquías separadas: abstracción e implementación, que pueden desarrollarse independientemente una de la otra.

Problema:

Supongamos que tiene una clase de forma geométrica con un par de subclases: círculo y cuadrado. Desea ampliar esta jerarquía de clases para incorporar colores, por lo que planea crear subclases de formas rojas y azules. Sin embargo, como ya tiene dos subclases, deberá crear cuatro combinaciones de clases. (círculo azul, círculo rojo, cuadrado azul, cuadrado rojo)

Agregar nuevos tipos de formas y colores a la jerarquía lo hará crecer exponencialmente. Por ejemplo, para agregar una forma de triángulo necesitaría introducir dos subclases, una para cada color. Y después de eso, agregar un nuevo color requeriría crear tres subclases, una para cada tipo de forma. Cuanto más avanzamos, peor se vuelve.

Este problema se produce porque estamos tratando de extender las clases de formas en dos dimensiones independientes: por forma y por color. Ese es un problema muy común con la herencia de clases.

Solución:

El patrón Bridge intenta resolver este problema cambiando de herencia a la composición del objeto. Lo que esto significa es que extrae una de las dimensiones en una jerarquía de clases separada, de modo que las clases originales hagan referencia a un objeto de la nueva jerarquía, en lugar de tener todos sus estados y comportamientos dentro de una clase.

(Puede evitar la explosión de una jerarquía de clases transformándola en varias jerarquías relacionadas.)

Siguiendo este enfoque, podemos extraer el código relacionado con el color en su propia clase con dos subclases: Rojo y Azul. La clase Figura/Forma luego obtiene un campo de referencia que apunta a uno de los objetos de color. Ahora la forma puede delegar cualquier trabajo relacionado con el color al objeto de color vinculado. Esa referencia actuará como un puente entre las clases de Figura/Forma y Color. De ahora en adelante, agregar nuevos colores no requerirá cambiar la jerarquía de formas, y viceversa.

Implementación:

- ➔ Use el patrón Bridge cuando desee dividir y organizar una clase monolítica que tenga varias variantes de alguna funcionalidad (por ejemplo, si la clase puede trabajar con varios servidores de bases de datos).
  - ◆ Cuanto más grande se vuelve una clase, más difícil es descubrir cómo funciona y más tiempo se necesita para hacer un cambio. Los cambios realizados en una de las variaciones de funcionalidad pueden requerir realizar cambios en toda la clase, lo que a menudo resulta en cometer errores o no abordar algunos efectos secundarios críticos.
  - ◆ El patrón Bridge le permite dividir la clase monolítica en varias jerarquías de clases. Después de esto, puede cambiar las clases en cada jerarquía independientemente de las clases en las demás. Este enfoque simplifica el mantenimiento del código y minimiza el riesgo de romper el código existente.

➔ Use el patrón cuando necesite extender una clase en varias dimensiones ortogonales (independientes).
- ◆ El patrón Bridge sugiere que extraiga una jerarquía de clases separada para cada una de las dimensiones. La clase original delega el trabajo relacionado a los objetos que pertenecen a esas jerarquías en lugar de hacer todo por sí mismo.

➔ Use el Bridge si necesita poder cambiar las implementaciones en tiempo de ejecución.
- ◆ Aunque es opcional, el patrón Bridge le permite reemplazar el objeto de implementación dentro de la abstracción. Es tan fácil como asignar un nuevo valor a un campo.
- ◆ Por cierto, este último elemento es la razón principal por la que tanta gente confunde el Bridge con el patrón de Strategy. Recuerda que un patrón es más que una determinada forma de estructurar tus clases. También puede comunicar intención y un problema que se está abordando.

Cómo implementar:

1. Identifica las dimensiones ortogonales en tus clases. Estos conceptos independientes pueden ser: abstracción/plataforma, dominio/infraestructura, front-end/back-end o interfaz/implementación.

2. Vea qué operaciones necesita el cliente y defínalos en base a la clase de abstracción.

3. Determine las operaciones disponibles en todas las plataformas. Declare las que necesite la abstracción en la interfaz de implementación general.

4. Para todas las plataformas en su dominio, cree clases de implementación concretas, pero asegúrese de que todas sigan la interfaz de implementación.

5. Dentro de la clase de abstracción, agregue un campo de referencia para el tipo de implementación. La abstracción delega la mayor parte del trabajo al objeto de implementación al que se hace referencia en ese campo.

6. Si tiene varias variantes de lógica de alto nivel, cree abstracciones refinadas para cada variante extendiendo la clase de abstracción base.

7. El código del cliente debe pasar un objeto de implementación al constructor de la abstracción para asociar uno con el otro. Después de eso, el cliente puede olvidarse de la implementación y trabajar solo con el objeto de abstracción.

Ventajas y desventajas:

✔Puede crear clases y aplicaciones independientes de la plataforma.

✔El código del cliente funciona con abstracciones de alto nivel. No está expuesto a los detalles de la plataforma.

✔*Principio abierto / cerrado*. Puede introducir nuevas abstracciones e implementaciones independientemente una de la otra.

✔*Principio de responsabilidad única.* Puede centrarse en la lógica de alto nivel en la abstracción y en los detalles de la plataforma en la implementación.

❌Puede hacer que el código sea más complicado aplicando el patrón a una clase altamente cohesionada.

Relaciones con otros patrones

Bridge generalmente se diseña por adelantado, lo que le permite desarrollar partes de una aplicación independientemente una de la otra. Por otro lado, Adapter se usa comúnmente con una aplicación existente para hacer que algunas clases incompatibles funcionen bien juntas.

Bridge, State, Strategy (y hasta cierto punto Adapter) tienen estructuras muy similares. De hecho, todos estos patrones se basan en la composición, que delega el trabajo a otros objetos. Sin embargo, todos resuelven diferentes problemas. Un patrón no es solo una receta para estructurar su código de una manera específica. También puede comunicar a otros desarrolladores el problema que resuelve el patrón.

Puedes usar Abstract Factory junto con Bridge. Este emparejamiento es útil cuando algunas abstracciones definidas por Bridge solo pueden funcionar con implementaciones específicas. En este caso, Abstract Factory puede encapsular estas relaciones y ocultar la complejidad del código del cliente.

Puede combinar Builder con Bridge: la clase de director desempeña el papel de la abstracción, mientras que los diferentes constructores actúan como implementaciones.