**Pattern: BRIDGE**

**Información general del proyecto, deben incluir la URL para consultar el proyecto:**

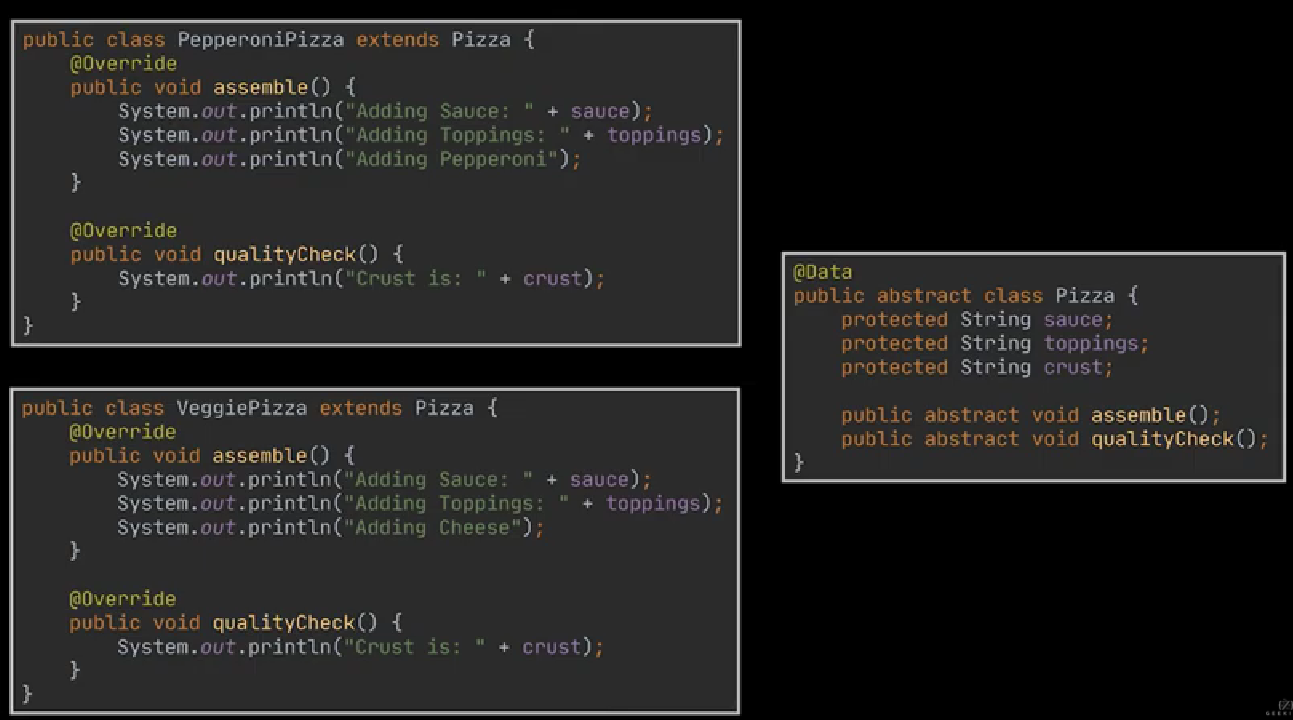
* En este caso, este proyecto es una implementación de un restaurante de pizzas, que busca poder vender dos tipos de pizza de pepperoni y vegetariana. Pero el negocio fue tan bueno, que empezaron a vender a americanos e italianos; por lo que tuvieron que modificar su software para que se pudieran generar pedidos de estas dos pizzas al modo italiano y al modo americano.

<https://www.youtube.com/watch?v=88kAIisOiYs&ab_channel=Geekific>

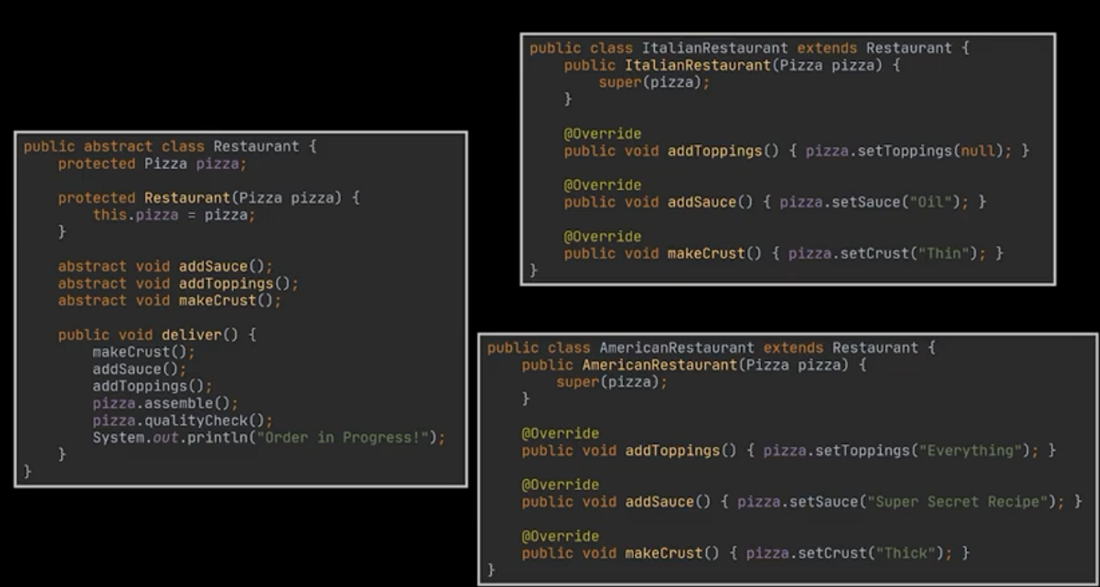
Link: <https://github.com/geekific-official/geekific-youtube/tree/main/pattern-structural-bridge>

**Información y estructura del fragmento del proyecto donde aparece el patrón. No se limiten únicamente a los elementos que hacen parte del patrón: para que tenga sentido su uso, probablemente van a tener que incluir elementos cercanos que sirvan para contextualizarlo.**

En la imagen de abajo, podemos ver la implementación de Pizza y sus implementaciones concretas (PeperoniPizza y VeggiePizza). Aquí se puede observar como las implementaciones concretar tienen comportamientos específicos y hacen que el código se comporta de maneras diferentes acorde a lo que el cliente pida.



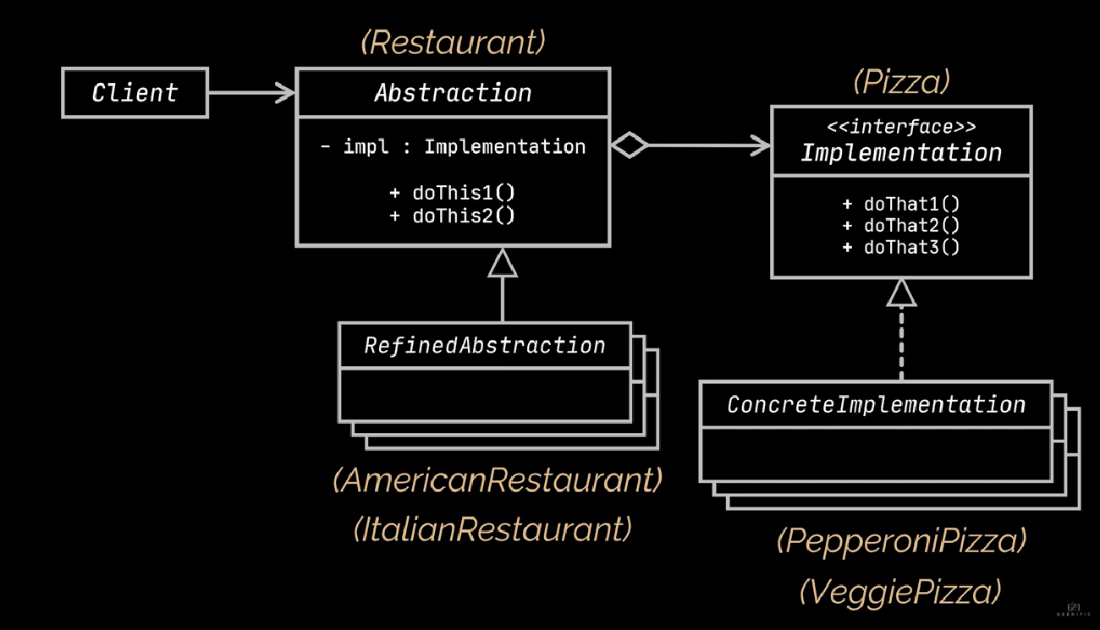
En esta imagen podemos ver, la abstracción de Restaurante y sus Abstracciones refinadas (ItalianRestaurant/AmericanRestaurant). Aquí podemos observar, como las abstracciones refinadas ayudan a definir los atributos de la forma deseada y especificada.



Finalmente, podemos observar como en la clase Principal, se facilita mucho más poder crear objetos con las dos abstracciones deseadas sin generar mayor conflicto a la hora de instanciarlas.



Aquí podemos observar el UML de “análisis” el cual no tiene los métodos. Debajo de la imagen se encuentra la explicación de cada uno de los participantes del patrón bridge orientado específicamente a las clases del proyecto.



Para este proyecto se puede decir que el “puente” o nuestro patrón se encuentra entre la clase Restaurante y la clase Pizza; ya que entre ellas dos se definen las dos abstracciones necesarias; las cuales debían independizarse para poder tener un mayor orden y eficiencia en el código.

**Abstracción: Restaurante**

La función principal de restaurante es controlar la petición del cliente y adaptarla a la abstracción refinada para luego así, pasar el mensaje a la implementación.

-Métodos: addSauce(), addToppings(), MakeCrust() y deliver()

**Abstracción refinada: American/Italian Restaurant**

Estas clases tiene el propósito de añadir funcionalidad o detalles a la Abstracción; con el fin de darle un “molde” más específico.

-Métodos: addSauce(), addToppings(), MakeCrust()

**Implementación: Pizza**

En este caso, la función de la clase pizza es abstraer la información adicional requerida por el cliente para la preparación de la pizza.

-Métodos: Assemble() y QualityCheck()

**Implementación concreta: Pepperoni/Veggie Pizza**

Estas clases lo que hacen es definir los atributos específicos para la pizza.

-Métodos: Assemble() y QualityCheck()

**Información general sobre el patrón: qué patrón es y para qué se usa usualmente:**

**Tipo:** Estructural

#### Intención: Separar una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas: abstracción (interfaz) e implementación. Para que se puedan desarrollar independientemente la una de la otra.

#### Aplicabilidad:

**Utiliza el patrón Bridge cuando:**

* Quieras dividir y organizar una clase monolítica que tenga muchas variantes de una sola funcionalidad (por ejemplo, si la clase puede trabajar con diversos servidores de bases de datos). El patrón Bridge te permite dividir la clase monolítica en varias jerarquías de clase. Después, puedes cambiar las clases de cada jerarquía independientemente de las clases de las otras. Esta solución simplifica el mantenimiento del código y minimiza el riesgo de descomponer el código existente.
* Utiliza el patrón cuando necesites extender una clase en varias dimensiones independientes.El patrón Bridge sugiere que extraigas una jerarquía de clase separada para cada una de las dimensiones. La clase original delega el trabajo relacionado a los objetos pertenecientes a dichas jerarquías, en lugar de hacerlo todo por su cuenta.

* Utiliza el patrón Bridge cuando necesites poder cambiar implementaciones durante el tiempo de ejecución. Aunque es opcional, el patrón Bridge te permite sustituir el objeto de implementación dentro de la abstracción. Es tan sencillo como asignar un nuevo valor a un campo.

#### Estructura:

Diagrama

Descripción generada automáticamente

#### Participantes

**Abstraction (Window)**

– Define la interfaz.

– Contiene una referencia a un objeto de tipo “Implementor”

**RefinedAbstraction (IconWindow)**

– Extiende la interfaz definida por Abstractionce

**Implementor (WindowImp)**

– Define la interfaz para las clases de implementacion. Esta interfaz no tiene que se igual a la interfaz de la abstraccion. Es decir usualmente la interfaz de Implementor provee las operaciones base y Abtraccion basandose en estas define unas operaciones de mayor nivel.

**ConcreteImplementor (XWindowImp, PMWindowImp)**

–Implementa la interfaz de Implementor y define su implementacion concreta.

**Collaborations**

- La abstraccion manda los pedidos del cliente a su objeto Implmentor.

**Información del patrón aplicado al proyecto: explicar cómo se está utilizando el patrón dentro del proyecto**

El proyecto es una implementación de un restaurante de pizzas, que busca poder vender dos tipos de pizza de pepperoni y vegetariana. Dado el negocio fue tan bueno, que empezaron a vender a americanos e italianos; por lo que tuvieron que modificar su software para que se pudieran generar pedidos de estas dos pizzas al modo italiano y al modo americano. El problema surge cuando tendrán que crear nuevas clases para cada combinación de pizzas y que, de este, es decir están intentando extender la clase de pizza en dos dimensiones independientes (restaurante y modo), modo si se siguen expandiendo las clases crecerán exponencialmente. Por ende, lo que se hace es separar el restaurante, y que esto tenga una conexión (bridge) a pizza, de este modo cada restaurante puede hacer ambas pizzas sin necesidad de una subclase de sabor por cada restaurante.

**¿Por qué tiene sentido haber utilizado el patrón en ese punto del proyecto? ¿Qué ventajas tiene?**

En este caso, el probema principal que tuvieron es que se vio la necesidad de implementar de mejor forma el software, de tal forma, que se pudiera añadir distintas formas de preparacion de una mismo sabor (ej. peperoni) para distintas nacionalidades. El problema más especificamente radico en que se intento extender la clase pizza a dos dimensiones independientes (por tipo de pizza y la forma de preparacion). En este caso, este patron fue muy util porque cambio la herencia en composicion. Esto significa que se extrae una de las dimensiones en una clase separada, así las asi las clases originales referencian un objeto del nuevo tipo. Esto hace que una de las clases funcione como un puente que facilita el flujo de información.

En general se logró:

* Independizar el tipo de preparación de la pizza dependiendo del país.
* El código cliente funciona con abstracciones de alto nivel. No está expuesto a los detalles de la plataforma.
* *Principio de abierto/cerrado*. Puedes introducir nuevas abstracciones e implementaciones independientes entre sí.
* *Principio de responsabilidad única*. Puedes centrarte en la lógica de alto nivel en la abstracción y en detalles de la plataforma en la implementación.

**¿Qué desventajas tiene haber utilizado el patrón en ese punto del proyecto?**

* Puede ser que el código se llegue a complique si aplicas el patrón a una clase muy cohesionada.
* Incrementa la complejidad del código
* También se puede llegar a aumentar mucho la cantidad de clases o variaciones si se agregan restaurantes y sabores por igual

**¿De qué otras formas se le ocurre que se podrían haber solucionado, en este caso particular, los problemas que resuelve el patrón?**

El proyecto es una implementación de un restaurante de pizzas, que busca poder vender dos tipos de pizza de pepperoni y vegetariana. Dado el negocio fue tan bueno, que empezaron a vender a americanos e italianos; por lo que tuvieron que modificar su software para que se pudieran generar pedidos de estas dos pizzas al modo italiano y al modo americano. El problema surge cuando tendrán que crear nuevas clases para cada combinación de pizzas y que, de este, es decir están intentando extender la clase de pizza en dos dimensiones independientes (restaurante y modo), modo si se siguen expandiendo las clases crecerán exponencialmente. Por ende, lo que se hace es separar el restaurante, y que esto tenga una conexión (bridge) a pizza, de este modo cada restaurante puede hacer ambas pizzas sin necesidad de una subclase de sabor por cada restaurante.

Este mismo problema se podria solucionar ppor ejemplo usando otros patrones, especificamente Strategy o Adapter.

Strategy:

* El patrón Strategy te permite alterar indirectamente el comportamiento del objeto, en este caso la pizza durante el tiempo de ejecución asociándolo con distintos subobjetos como puede ser el restaurante que pueden realizar subtareas específicas como cambiar la forma de preparacion o ingredientes.

- Utiliza el patrón cuando tu clase tenga un enorme operador condicional como la forma de preparar según restaurante, que cambie entre distintas variantes del mismo algoritmo. Entonces, Strategy te permite suprimir dicho condicional extrayendo todos los algoritmos para ponerlos en clases separadas, las cuales implementan la misma interfaz. El objeto original pizza delega la ejecución a uno de esos objetos (tipo restuarante), en lugar de implementar todas las variantes del algoritmo.

State:

Utiliza el patrón cuando tengas una clase contaminada con enormes condicionales como preparación o sabor que alteran el modo en que se comporta la clase de acuerdo con los valores actuales de los campos de la clase. Entonces State te permite extraer ramas de esos condicionales a métodos de las clases estado correspondientes (sabor y forma). Al hacerlo, también puedes limpiar campos temporales y métodos de ayuda implicados en código específico del estado de fuera de tu clase principal.