**Taller 5**

**Proyecto:** <https://github.com/BrijeshSaxena/design-pattern-bridge>

**Información general:**

El proyecto simula la manufactura de una bicicleta, un carro y un bus. Cada vehículo llama a determinados talleres según le corresponda y se imprime por pantalla la acción que se está ejecutando en el taller y cuánto tiempo tomó completarla.

El proyecto cuenta con; una clase Main para manufacturar los vehículos; 2 clases abstractas que son Vehicle y WorkShop; 3 clases que extienden de Vehicle y son Bike, Bus y Car; y 5 clases que extienden de WorkShop y son AssembleWorkShop, PaintWorkShop, ProduceWorkShop, RepairWorkShop y TestWorkShop.

A screenshot of a computer

Description automatically generated with medium confidence

La clase Main ejecuta el programa y crea primero una bicicleta de tipo Vehicle (y tipo Bike), le añade los talleres Produce, Assemble y Test, y por último manufactura la bicicleta.

Luego hace lo mismo con un carro y un bus. Para cada vehículo llama talleres diferentes.

A picture containing text, screenshot, menu

Description automatically generated

La clase Vehicle construye los objetos de tipo Vehicle, asignándoles un atributo de tipo List que contiene la lista de talleres por los que debe pasar el vehículo. El método joinWorkShop se encarga de añadir un taller a la lista. Adicionalmente, tiene los métodos abstractos manufacture y minWorkTime para que sean implementados particularmente por cada vehículo.

A screen shot of a computer program

Description automatically generated with low confidence

Para cada uno de los vehículos en particular se implementan los 2 métodos abstractos que heredan desde Vehicle. El método minWorkTime retorna una constante entera que es mayor cuanto más grande/complejo es el vehículo. El método manufacture se encarga de imprimir por consola el vehículo que se está manufacturando, luego recorre cada uno de los talleres en su lista de talleres y hace sus respectivos trabajos, y por último indica que el proceso terminó.

A picture containing text, screenshot, software, multimedia software

Description automatically generatedA picture containing text, screenshot, font

Description automatically generatedA picture containing text, screenshot, software

Description automatically generated

La clase WorkShop construye los objetos de tipo WorkShop, Tiene el método abstracto work para ser implementado particularmente por cada taller.

A screen shot of a computer code

Description automatically generated with low confidence

En cada uno de los talleres particulares se implementa el método abstracto que heredó desde WorkShop. El método work se encarga de imprimir por consola el trabajo que se está llevando a cabo, luego calcula el tiempo que tomará el trabajo multiplicando una constante por el minWorkTime del vehículo en cuestión, espera en tiempo real el tiempo calculado, y por último indica que el proceso terminó e imprime cuánto tiempo tomó.

A picture containing text, screenshot

Description automatically generated A picture containing text, screenshot

Description automatically generated A picture containing text, screenshot

Description automatically generated A screen shot of a computer program

Description automatically generated with low confidence A picture containing text, screenshot

Description automatically generated

En resumen, la clase main instancia vehículos y cada vehículo tiene un atributo workshops que contiene la lista de talleres que debe usar en su manufactura. La clase WorkShop y la clase Vehicle son abstracciones y cada clase (Bike, Car, Bus, Assembly, Paint, Produce, Repair y Test) es una implementación particular de su abstracción y por lo tanto retorna cosas diferentes en sus métodos heredados. El UML que representa el programa es el siguiente: A picture containing diagram, text, line, parallel

Description automatically generated

Como se puede ver, las abstracciones se relacionan entre sí con un atributo y adicionalmente cada implementación de vehículo utiliza las implementaciones de los talleres mediante su abstracción.

**Patrón de diseño**

Este proyecto utiliza un patrón Bridge. El patrón Bridge consiste en reducir el acoplamiento entre las abstracciones y las implementaciones, de forma que cada una pueda crecer de manera independiente. En este patrón las abstracciones se hacen mediante interfaces o clases abstractas que contienen referencias hacia los implementadores. El código encargado del cliente solo interactúa con las abstracciones sin preocuparse por las implementaciones, y asímismo, cambios en las implementaciones no afectan el código del cliente ya que no interactúan directamente entre sí. Es un diseño enfocado a la flexibilidad y extensibilidad del código.

**Bridge en el proyecto**

En este proyecto en particular se puede ver claramente la aplicación del diseño Bridge. El código del cliente (Main) únicamente puede interactuar con la abstracción Vehicle, y a su vez, la abstracción Vehicle cuenta con 3 implementaciones concretas que utilizan la abstracción Workshop y sus distintas implementaciones concretas según corresponda. En este caso, la clase Main solo instancia los vehículos y hace llamados a las funciones que hacen parte de los vehículos, sin embargo, no le conciernen los detalles de las implementaciones concretas. Es decir, la clase Main no hace acciones diferentes en caso de que sea una Bike, un Car o un Bus, sino que solo trabaja con objetos de tipo Vehicle sin importarle cómo funciona internamente cada vehículo y mucho menos lo que hace cada taller. En este proyecto sería muy sencillo añadir nuevos vehículos o nuevos talleres concretos y no afectaría para nada el funcionamiento de Main, permitiendo que las abstracciones y las implementaciones crezcan de manera independiente.

**Ventajas**

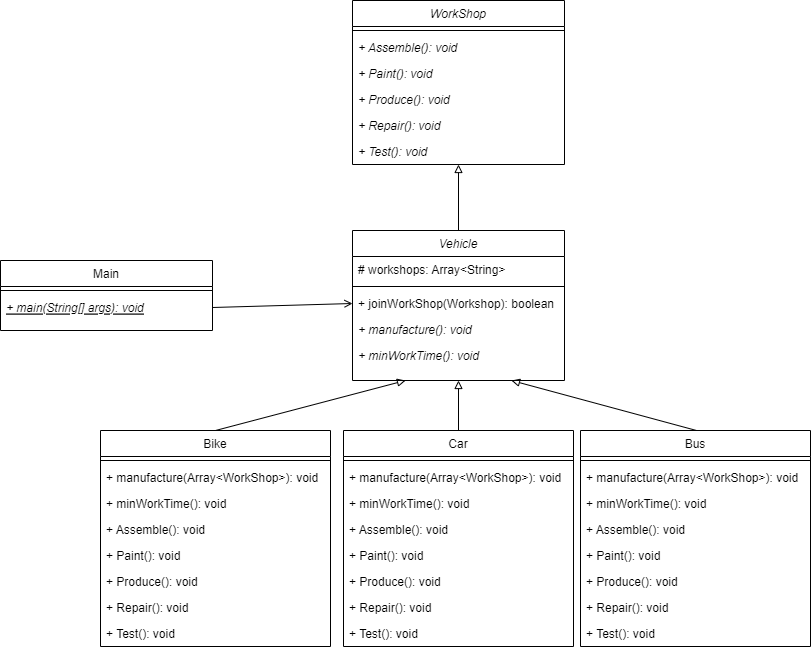
* Desacomplamiento de las abstracciones y las implementaciones
* Crecimiento independiente de abstracciones e implementaciones
* Mayor reusabilidad de código
* Flexibildad en el código
* Adaptabilidad del sistema
* Se facilitan las pruebas de cada componente por aparte
* Código sencillo de leer y entender
* Clara jerarquía de clases

**Desventajas**

* Código más difícil de escribir
* Un paso extra por la jerarquía
* Mayor tiempo y esfuerzo en diseño
* Mayor tiempo y esfuerzo en desarrollo
* No aplica en todos los escenarios (inútil si el código no se extenderá mucho en el futuro)

**Posibles cambios**

Este proyecto utilizó clases abstractas para las abstracciones, aunque también podría utilizar interfaces por si en el futuro alguna implementación debe implementar 2 o más abstracciones (ya que no puede extender 2 clases a la vez), por ejemplo, que un Bus implemente Vehicle y también Enterprise y que así implemete métodos relacionados a su fabricación y también a su productividad en una empresa.

Otras formas de resolver el problema habrían sido simplemente usar herencia y/o composición. Por ejemplo: 

En este diseño, Vehicle es una clase abstracta que extiende la clase abstracta WorkShop. Como WorkShop no implementa sus métodos por ser abstractos, y Vehicle tampoco lo hace, entonces es cada implementación de vehículo la que se encarga de hacer sus propias acciones. El Main instanciaría los vehículos particulares y según los workshops de cada vechículo particular, el método manufacture se encargaría de hacer los respectivos a llamados a Assemble, Paint, Produce, Repair y Test según corresponda, y a su vez, cada uno de esos métodos haría uso de minWorkTime, o mejor aún, la constante de minWorkTime podría ser un atributo de cada vehículo que se use en cada taller.

Otra posible solución sería usar un patrón de diseño MVC en donde Main sea la única clase que interactue con el usuario y pueda hacer todos los posibles llamados a un controlador, y que el controlador pueda llamar concretamente a cada clase (abstracta o concreta), siendo cada vehículo un information-holder de minWorkTime y siendo cada taller un service-provider que reciba los vehículos.