# Bridge design pattern

In this chapter we are going to look at the bridge design pattern. The bridge design pattern is used to separate objects from there implementation.

Contents

[Bridge design pattern 1](#_Toc57396027)

[Bridges everywhere 1](#_Toc57396028)

[The Pattern 1](#_Toc57396029)

[Code-specific example 3](#_Toc57396030)

[Code refactoring 6](#_Toc57396031)

[Bridge design pattern UML 9](#_Toc57396032)

[Benefits of the Bridge Design pattern 10](#_Toc57396033)

[Disadvantages of the Bridge Design Pattern 10](#_Toc57396034)

[Consequences of bridge design pattern 10](#_Toc57396035)

[Implementation of bridge design pattern 11](#_Toc57396036)

[When to use the bridge design pattern 12](#_Toc57396037)

[Bridge vs Adapter 12](#_Toc57396038)

[References 12](#_Toc57396039)

# Bridges everywhere

Let us look at a simple household switch that was designed to turn on a light. When the switch is turned on the light turns on and when the switch is turned off the light turns off. The switch only has one job, to turn the light on and off. The light has its own job to connect to electricity and power the bulb. Let us say we see the potential of the switch to turn on and off other objects in our house. Let us say we want to make a copy of this switch and use it to turn our radio on and off at the power plug. The switch as it is, is coupled with the light bulb and electricity components that make the light work. To begin to use that switch on other appliances we will need to separate the switch from the light components. This is where the bridge pattern comes in, the bridge patter will allow us to decouple the switch from the light bulb components. Once we have used the bridge pattern the switch and light components are separate, the switch turns something on and off and the light component handles the powering of the light. With the switch separated from the light components we are free to use this switch on any number of applications in our house.

# The Pattern

The definition of the bridge design pattern according to the Gang of Four book Design Patterns is **“Decouples an abstraction from its implementation so that the two can vary independently.”**

Let us begin our understanding of the bridge design pattern by breaking down what this definition means. Let us start by understanding what is meant by abstraction and implementation. The meaning of abstraction and implementation can differ depending on which programming language you use. An abstract in java is usually considered to be and abstract class or an interface. An implementation in java is usually considered to be an implementation of that interface or concrete class that inherits from that abstract class.

The definition given by the Gang of Four book Design Patterns is not language pacific. This is mostly confusing due to the word interface. Let us not focus on what these words mean in a pacific programming language but try to consider what these words are trying to convey. Let us start by understanding what is meant by abstraction. An abstraction is a way of looking at something without understanding how that thing works. A car steering wheel is an abstraction, it allows us to turn the car without ever understanding what makes the car change direction. The fact that the car wheel is an abstraction allows us to turn the car without having to worry about the mechanics behind the car turning process. An implementation is the mechanics that are working under the hood to make the car turn. The implementation is taking the abstract wheel and making it real. In most circumstances the abstract and the implementation are coupled. The car steering wheel is coupled to the car axel.

The car steering wheel in this example might be discovered to have the potential to be used in other types of vehicles. The car steering wheel manufacturer might want this steering wheel to be used in and aeroplane or a boat. The steering wheel of the car might work to turn left or right but as it is coupled with the car axel it will behave very differently than it would in a boat or a plane. What the manufacturer would have to do is decouple the steering wheel from the car axel so that the steering wheel can act independently of the car axel. This is where the bridge pattern comes in. The Gang of Four description of the bridge design pattern should be some bit clearer now. The bridge pattern in this example would decouple the abstract steering wheel from the car axel implementation so that both the wheel and the axel can work independently.

To create a bridge pattern for the steering wheel we must first decouple the abstract car steering wheel from the car axel implementation. We then create a higher-level abstraction of that steering wheel and call it a vehicle steering wheel. The vehicle steering wheel becomes our abstract and the car steering wheel becomes and implementation of that abstraction. When we have a car steering wheel implementation, we can create implementations of other vehicle such as boats or aeroplanes.

Diagram

Description automatically generated

Figure Steering Wheel

In Figure 1 Steering Wheel We can see that the vehicle steering wheel is now a high level abstraction and the car steering , boat steering and plane steering wheels are implantations of the abstract vehicle steering wheel. The Steering system has become an implementor of the abstraction and the axle, rudder and wing steering have become concrete implementations of that implementation.

Now that the steering wheel is not coupled to a pacific steering system, we are free to apply the steering wheel to other steering systems.

# Code-specific example

Now that I have explained the bridge design pattern, we can look at the pattern in a more code pacific way. The example I am going to use will be of a paint shop. The paint shop has a car painting booth which paints a car blue or black. The owner of the paint shop would like to be able to paint trucks as well as cars. He has thought about building a new paint boot for the trucks but would like to use the same paint booth to paint the cars and trucks.

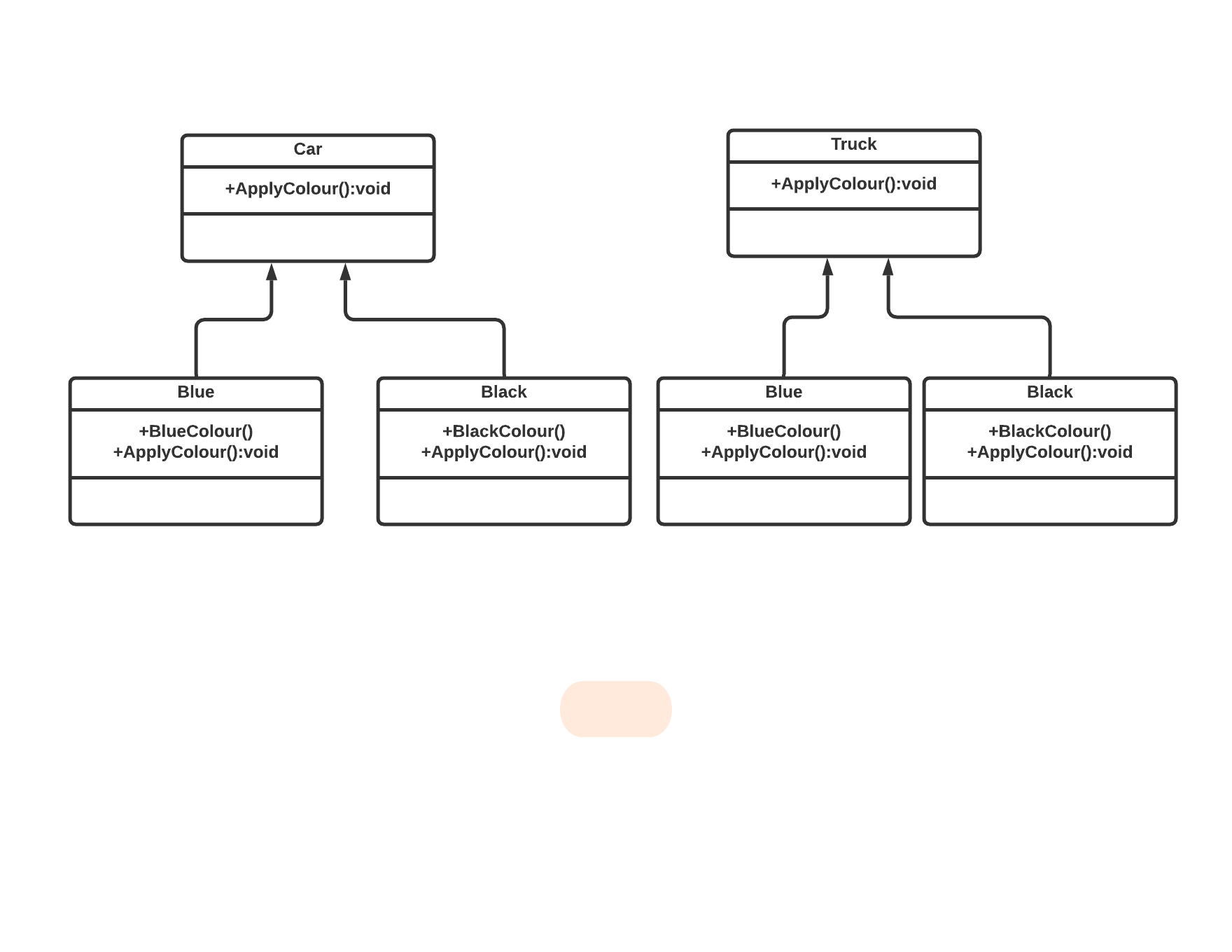


Figure Paint booths

Graphical user interface, text, application

Description automatically generated

Figure 3 Car class

Graphical user interface, text, application

Description automatically generated

Figure 4 Black class

Graphical user interface, text, application

Description automatically generated

Figure 5 Blue class

Figure 2 Paint booths shows the two paint booths for the car and truck. The solution above has a problem, if you need to change the car class you might end up having to change Blue and Black as well. IF this change is paint booth related you may need to change the Truck class as well. Imagine if the manager of the booth wanted to add a van vehicle. The paint shop owner would have to create a new booth and the classes blue and black paint would have to be added also. This could make the number of classes needed grow exponentially. Without the bridge pattern you would have to create a new class and new colour sub classes. Ideally the paint shop owner would have one class for paint and another for vehicles. This would allow the owner to add several vehicles and paint them anything from the colour class. We will now apply the bridge pattern to create a higher-level colour class and an implementor class for the car and truck called vehicle.

# Code refactoring

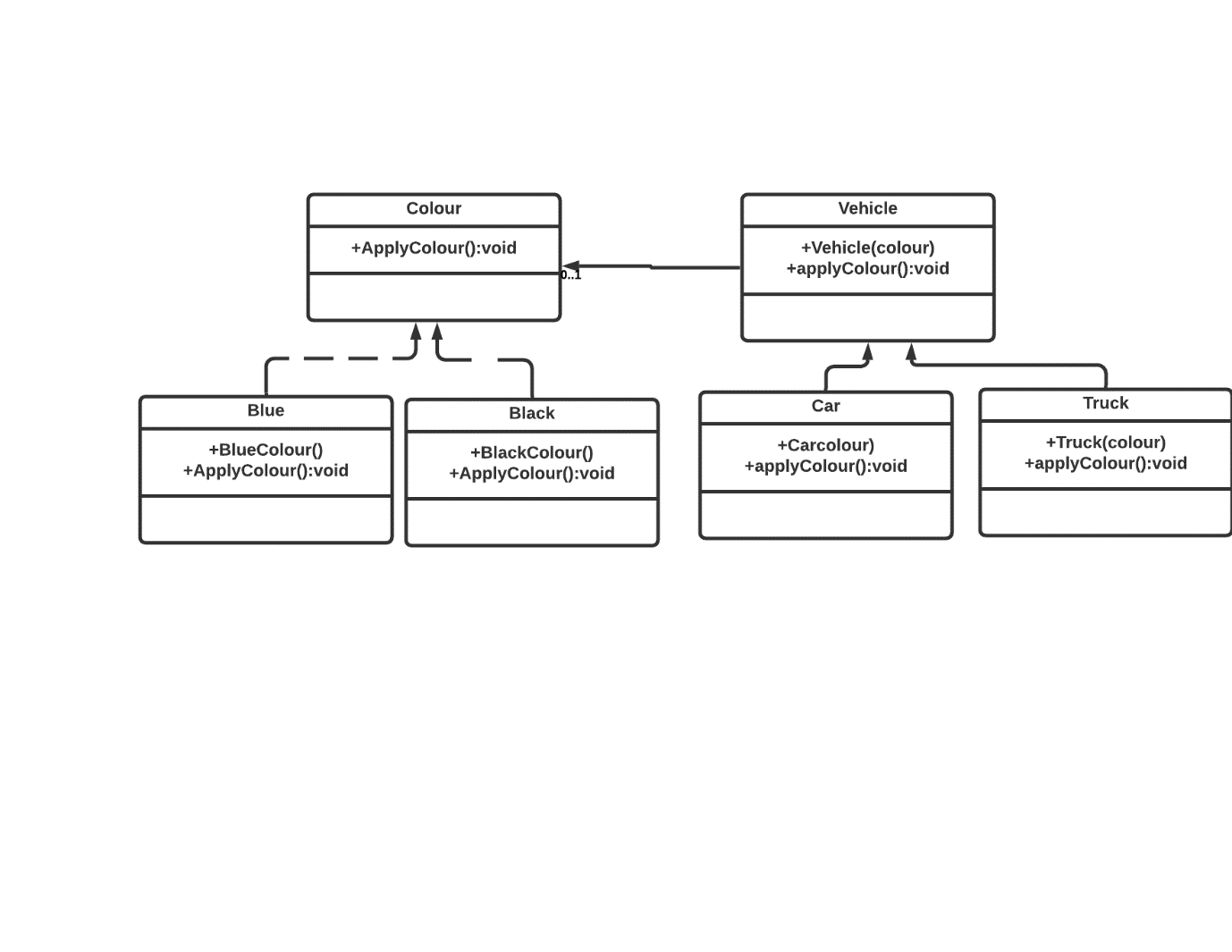


Figure 6 Bridge Pattern

After refactoring the previous code Figure 6 Bridge Pattern shows the colour class has being decoupled from the car and truck class and is now able to operate free and independently. The car and truck class are now in a new class called vehicle; this class is bridged with the colour class. If new colours or vehicles should need to be added they will also be able to be painted anything from the colour class and any new colours that are added can be used by the vehicle class. Instead of having specific classes of car and truck that have their own colour sub class we merge the car and truck class into one vehicle class that can share the colour interface. Now the vehicle class can choose any colours from the colour class to apply to any vehicle in the vehicle class. The colour classes before were sub class to the type of vehicle if we added ten new colours the car class would have had 10 new colour subclass and so would the truck. This would have gotten out of hand quickly.

Here is the code for the colour and vehicle interfaces



Figure 7 Colour Interface

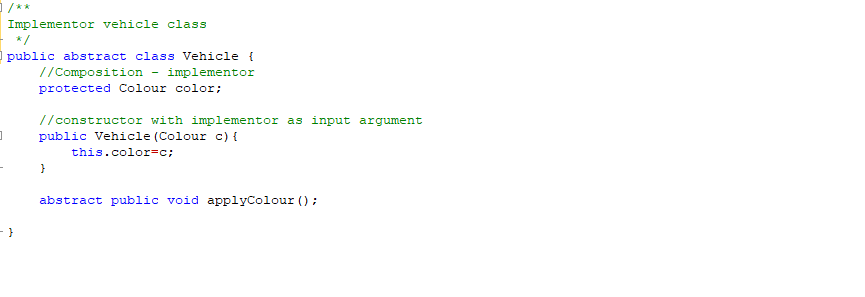


Figure 8 Vehicle Interface

We have a car and truck implementation

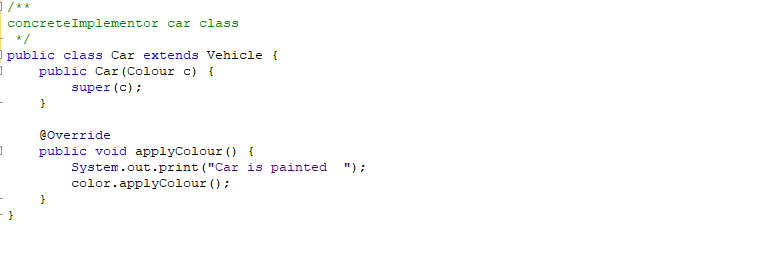


Figure 9 Car implementation

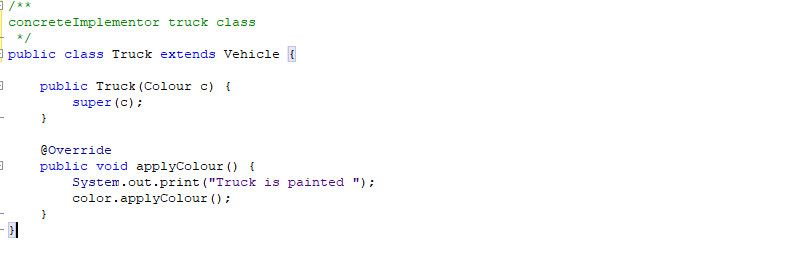


Figure 10 Truck Implementation

Here are the implementations of Black and Blue

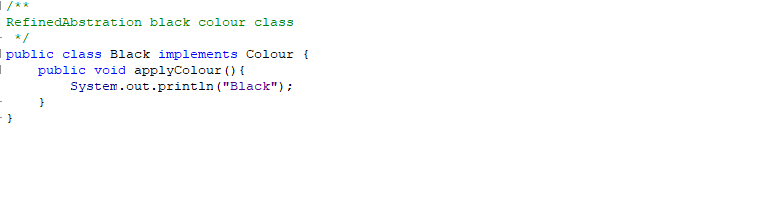


Figure 11 Black implementation

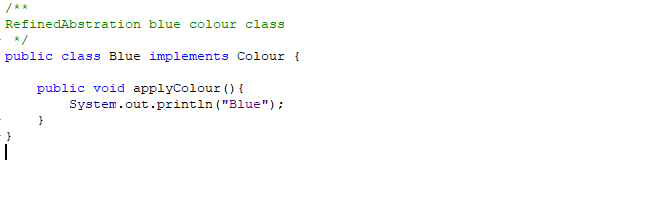


Figure 12 Blue Implementation

Here we have a test for the bridge design pattern

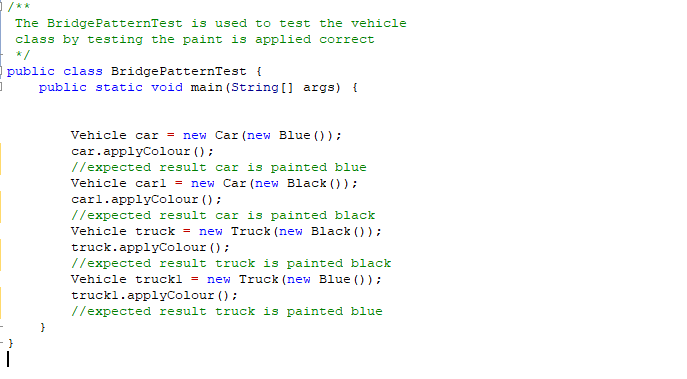


Figure 13 Test

Here is the output



Figure 14 Output

# Bridge design pattern UML

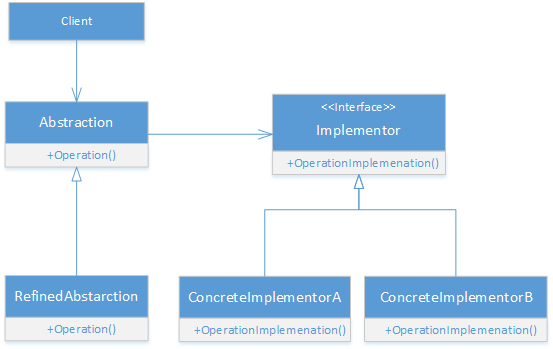


Figure 15 Bridge Pattern UML

#### Abstraction (abstract class)

* Most important part of the bridge design. The abstraction references the implementor.

#### RefinedAbstraction (normal class)

* The RefinedAbstraction extends the abstraction and extends in greater detail one level below. The RefinedAbstraction hides the greater details from the implementor

#### Implementer (interface)

* The Implementer defines the interface for the Implementer classes. The interface can differ from the abstraction interface and does not need to correspond to the abstraction interface.

#### ConcreteImplementor (normal class)

* The ConcreteImplementor implements the Implementer by providing concrete implementation.

In Figure 15 Bridge Pattern UML we can see a UML diagram. (sunny94, 2018) This diagram represents the bridge design pattern, but this UML is not final and the design pattern can vary depending on user or use. Let us apply this to my paint shop example.

The colour class is the abstraction, the abstraction would act like a high-level controller with the implementation doing all the work. In this example the implementor has a method to apply paint. The implementor is the vehicle class the car and truck class would be concreteImplementor’s of that class. They would call on the implementor to do some type of function such as apply paint. Lastly, we have the RefinedAbstraction which would be the colours that can be applied to the vehicles.

# Benefits of the Bridge Design pattern

Code reusability: As we have seen in the example above the colours in the colour class are not sub class by the vehicle’s in the vehicle class. The bridge design pattern has helped with code reusability.

Reduce duplicate code: because we are not using the colours in the car and truck class we are cutting down on duplicate code.

Increase code maintainability: If we need to add or remove a colour, we now do not need to change them in the car and truck class only the colour class.

Increase productivity: We do not have to write code for the colours for separate vehicles.

Respects single responsibility principle: This states that every class or function must be responsible for a single part of the software’s functionality. The colour class is only responsible for the colours in its class and the vehicle class is only responsible for the vehicles in that class.

If something breaks, then it does not break everything: If the truck class breaks the car class can still be painted. If the truck class had an orange colour sub classed and then broke the car class would not have access to the orange colour.

# Disadvantages of the Bridge Design Pattern

The design increases the programs complexity.

# Consequences of bridge design pattern

The consequences of the bridge design pattern according to the design patterns book are. (Erich Gamma, 1994)

*Decoupling interface and implementation. An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time. It's even possible for an object to change its implementation at run-time. Decoupling Abstraction and Implementor also eliminates compile-time dependencies on the implementation. Changing an implementation class doesn't require recompiling the Abstraction class and its clients. This property is essential when you must ensure binary compatibility between different versions of a class library. Furthermore, this decoupling encourages layering that can lead to a better-structured system. The high-level part of a system only has to know about Abstraction and Implementor.* (Erich Gamma, 1994)

This talks about the implementation not being bound to the interface. This has advantages such as if you create and interface for a system and that interface is not bound to that system the interface can be taken and used in another application. This could be useful if your project fails and you would like to cut useful features and interfaces to be used in other projects. The consequences in the book talks about the implementation being able to be configured at run time and how changing the implementation does not require recompiling the abstraction at run time. This makes the implementation easier to change and the code more versatile.

*Improved extensibility*. *You can extend the Abstraction and Implementor hierarchies independently.* (Erich Gamma, 1994)

The bridge design pattern improves extensibility. Because the pattern decouples the code the abstraction and implementor can be extended independently this gives you added control over your program and will allow you to make changes to the program easier.

*Hiding implementation details from clients. You can shield clients from implementation details, like the sharing of implementor objects and the accompanying reference count mechanism (if any).* (Erich Gamma, 1994)

Keeping the user from needing to know how something works is and important part of software design. The user should be able to steer a car without knowing how the axle system works.

# Implementation of bridge design pattern

The implementation issues of the bridge design pattern according to the book design patterns are. (Erich Gamma, 1994)

*Only one Implementor. In situations where there's only one implementation, creating an abstract Implementor class isn't necessary. This is a degenerate case of the Bridge pattern; there's a one-to-one relationship between Abstraction and Implementor. Nevertheless, this separation is still useful when a change in the implementation of a class must not affect its existing clients—that is, they shouldn't have to be recompiled, just relinked. Carolan [Car89] uses the term "Cheshire Cat" to describe this separation. In C++, the class interface of the Implementor class can be defined in a private header file that isn't provided to clients. This lets you hide an implementation of a class completely from its clients. When to use the Bridge design pattern.* (Erich Gamma, 1994)

The relationship between the abstract and the implementor is that the abstract acts like a controller and the implementation does all the work. There is a one to one relationship between the abstract and the implementor, if in the instance there is only one implementation creating an abstract implementor is not necessary. Instead you can hide the implementation in a private header file that will keep the privacy and access of the program intact.

*Creating the right Implementor object. How, when, and where do you decide which Implementor class to instantiate when there's more than one? If Abstraction knows about all ConcreteImplementor classes, then it can instantiate one of them in its constructor; it can decide between them based on parameters passed to its constructor. If, for example, a collection class supports multiple implementations, the decision can be based on the size of the collection. A linked list implementation can be used for small collections and a hash table for larger ones. Another approach is to choose a default implementation initially and change it later according to usage. For example, if the collection grows bigger than a certain threshold, then it switches its implementation to one that's more appropriate for a large number of items. It's also possible to delegate the decision to another object altogether. In the Window/WindowImp example, we can introduce a factory object (see Abstract Factory (99)) whose sole duty is to encapsulate platform-specifics. The factory knows what kind of WindowImp object to create for the platform in use; a Window simply asks it for a WindowImp, and it returns the right kind. A benefit of this approach is that Abstraction is not coupled directly to any of the Implementor classes.* (Erich Gamma, 1994)

This discusses how to choose the right implementor object when there are multiple implementors. If the abstraction knows all the concreteImplementor classes, then it is possible to instantiate them in the constructor. This paragraph discusses how to deal with large amounts of data collections and how to delegate decisions to other objects. The bridge design pattern is useful for keeping a program smaller and making code more reusable.

# When to use the bridge design pattern

Use the bridge design pattern when you have a class that contains multiple components that are not connected logically or are coupled together pointlessly. When a class becomes too big it makes it difficult to figure out how it works and harder to make changes to the class. The bridge design pattern lets you split classes into several different hierarchies. Once the classes are split in hierarchies these classes can be change independently, this makes code more maintainable and has less risk of braking.

Use the bridge design pattern if you want to switch implementations at runtime. The implementation of an abstraction can be changed as easily as assigning a new value.

The bridge design pattern favours composition over inheritance. Use the bridge design pattern when you must subclass different time in ways that are orthogonal with one another.

# Bridge vs Adapter

The adapter and bridge pattern are often mistaken for each other. The different between these two patterns are when they are implemented. The adapter pattern is implemented after your program has being created like when you need to get your existing code to work on a new interface.

The bridge pattern is implemented when your are creating your program or refactoring and existing program. The adapter pattern makes things work after they have being design whereas the bridge design pattern wakes thing works before they are designed.

# References

Erich Gamma, R. H. R. J. J. V., 1994. *Design Patterns.* United States: Addison-Wesley.

sunny94, N., 2018. *Bridge Design Pattern.* [Online]   
Available at: https://www.geeksforgeeks.org/bridge-design-pattern/  
[Accessed 19 11 2020].

GitHub link https://github.com/brendanit/Bridge