# 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](#_Toc80714779)

[Bridges everywhere 1](#_Toc80714780)

[The Pattern 1](#_Toc80714781)

[Bridge design pattern UML 4](#_Toc80714782)

[Code-specific example 5](#_Toc80714783)

[Benefits of the Bridge Design pattern 14](#_Toc80714784)

[Disadvantages of the Bridge Design Pattern 15](#_Toc80714785)

[Consequences of bridge design pattern 15](#_Toc80714786)

[Implementation of bridge design pattern 15](#_Toc80714787)

[When to use the bridge design pattern 16](#_Toc80714788)

[Bridge vs Adapter 17](#_Toc80714789)

[References 17](#_Toc80714790)

[Table of Figures 17](#_Toc80714791)

# 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.

# Bridge design pattern UML

Diagram

Description automatically generated

Figure 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.

# Code-specific example

In the software industry change is inevitable. A piece of software might need to be changed to incorporate a new feature or if the owner decides they would like to go a different way with it. In a perfect situation all the future requirements of a piece of software would have being realised but this is unrealistic. Certain changes will have to be made to most pieces of software at some point.

Code refactoring allows the developer to make changes to your code without changing the software functionality. A code refactor can make your code cleaner and easier to read. Refactoring allows you to remove unnecessary pieces of code and simplify the way it is written without changing what the code does. The most important benefits of code refactoring are for the developers who use the code or the future developers who might inherit the code. End users of the software might not feel the benefits of a code refactor immediately, but they too can benefit from code refactoring. Principles like You Aren’t Going To Need It (YAGTI) and SOLID will provide a good start so refactoring is not going to be as much of a hassle.

The main goals of refactoring in software projects are

* to avoid technical dept
* to avoid code rot
* to save the development time and money

Technical dept

Technical dept is a term that describes what results when development teams prioritise speedy delivery of a piece of software over perfect code. Technical dept is a catchall that covers everything from bugs to missing documentation. If you do not have time to test and rework your code until its clean you are accumulating technical dept. If software must be tested and reworked at sometimes you are only pushing the problem to the future.

Code rot

A software team can consist of several different developers with different coding styles. While developing a piece of software a developer might introduce correct but ugly code hacks as part of a quick solution for example when trying to meet a deadline. Without no routine for refactoring Code rot can creep in. Code rot is when the software’s code is becoming cluttered and losses integrity. Examples of Code rot include duplicate code, unused variables, unnecessary dependencies or too many parameters. Code refactoring should be done early to avoid bugs and errors.

Saving the development time and money

When the code is clear and easy to understand, existing and future developers can understand the code much quicker. This means that you spend less time explaining your code and providing training for your team before they can add new features. Easily readable code saves a lot of time and money and is key to streamlining your projects development.

To demonstrate the use of the Bridge design pattern to refactor your code lets take an overly simplified example of an account whose function is to provide a basic transaction. Let us assume this is the initial requirement and that the client was implemented. The code given to you is a savings account with some basic functions that allow you to deposit and withdraw money.

Diagram

Description automatically generated

Figure 10 Initial Solution

IAccount is the abstraction which the client would use

Text, letter

Description automatically generated

Figure 11 IAccount

SavingAccount provides concrete implementation of IAccount

Graphical user interface, text, application

Description automatically generated

Figure 12 SavingAccount

The client is tightly bound to IAccount

Graphical user interface, text, application

Description automatically generated

Figure 13 Program

When the program is run it outputs the results of a set of transactions. These are from SavingAccount which holds the programs behaviour.

Down the line the inevitable happens and we gain a whole set of new requirements. The software now needs to handle processing transactions depending on whether the accounts are personal or corporate. Then you see that changes to your software must be made, its time to consider whether this is a good time to refactor your code. You must change your code to incorporate the new changes anyway and there is a cue that more changes might have to be made. Refactor now rather than struggle later.

One solution to handling this new requirement would be to derive from the existing class and provide the 2 new types of accounts personal and corporate. The client code is tightly bound to the IAccount abstraction. If the instance arrives that you need a second credit account as well as your savings account, it will also have to support the personal and corporate accounts. As you might need to add more account types in the future, there would be an explosion of classes due to inheritance. Therefore, you should favour composition over inheritance while also retaining the abstraction.

Diagram

Description automatically generated

Figure 14 Solution with Inheritance

Using the Bridge Pattern here would allow you to separate the abstraction from the Implementation. Like the Bridge Pattern states it will allow us to introduce a new level of abstraction to the implementation. You will continue to inherit IAccount to provide different account types, but their implementations will be abstracted away through ITransaction.

The benefits of using the Bridge Design Pattern here are we are removing the permanent binding between the functional abstraction and its implementation. You will be able to make changes into implementing classes without affecting the abstraction and client code. You will be able to extend the abstraction and its implementations using sub classes.

The classes and objects participating in this newly refactored diagram:  
**Abstraction** (IAccount)

* It defines the abstraction’s interface.
* It maintains a reference to an object of type Implementor.

**RefinedAbstraction** (SavingAccount, CreditAccount)

* It extends the interface defined by Abstraction.

**Implementor** (ITransaction)

* It defines the interface for implementation classes. Typically, the Implementation interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.

**ConcreteImplementor** (Personal, Corporate)

* It implements the Implementor interface and defines its concrete implementation (java, 2017).

**Diagram

Description automatically generated**

Figure 15 Refactor with Bridge Design Pattern

As the client still goes through the interface implemented earlier IAccount this ensures that none of the refactoring impacts the client. The implementors of IAccount would have concrete instance of ITransaction. This means it is not exposed to the client or assigned by the client. This means that the client is not bound to ITransaction. We can now port the client to the underlying concrete implementations without recompiling it. Only the bridge and its implementor will need to be compiled. You can now add the new type CreditAccount while still using the same implementations. As you have decoupled your abstraction IAccount from the implementation ITransaction both can now vary independently.

Abstractions for implementations

Graphical user interface, text, application, email

Description automatically generated

Figure 16 ITransaction

The two types of accounts Personal and Corporate. The Personal and Corporate class was created to handle the two types of accounts. The type of account is chosen by the switch method in AccountBase. The two types of accounts allow the user to withdraw and deposit cash into the account. This class implements the ITransaction interface.

Text

Description automatically generated

Figure 17 Personal and Corporate

AccountBase is the base class which will be used to select the required implementer. There are other ways in which this could be implemented but the point is not to configure the concrete implementor by the client. A factory method could have been used to return the required instance.

The AccountBase class implements the IAccount interface. This class includes a switch statement to handle the corporate and personal accounts, getters, and setters and enum AccountType for choosing the different types of account.

Text

Description automatically generated

Figure 18 AccountBase

Refactored SavingAccount. The methods for withdrawing and depositing cash was removed from the class and added to the Personal class. SavingAccount now extends the base class AccountBase. This allows the user to choose if the savings account the wish to access is corporate or personal.

Graphical user interface, text, application

Description automatically generated

Figure 19 SavingAccount

The client class. This class still uses the same interface and has not being changed. The program class remains unchanged. The user can choose from accessing there personal or corporate saving account and credit account.

Graphical user interface, text, application

Description automatically generated

Figure 20 Program

Running the client will produce the same output as before but when you have a new instance you can continue to use the same implementations

The CreditAccount was created similar to the SavingAccount so the user could choose to access the Personal or Corporate accounts from their credit accounts.

Graphical user interface, text, application

Description automatically generatedFigure 21 CreditAccount

In this example I have take three classes IAccount, SavingAccount and Program and refactored them using the Bridge Design Pattern. The point of the refactor was to facilitate the need for a user to choose between different types of Personal and Corporate accounts. The strength of refactoring code is the ability to change the underlining code without changing how the program works. With this refactor the code was changed, and new classes were added to facilitate the different types of accounts, but the previous functions were not changed.

An AccountBase class was created to handle the need to switch between the Personal and Corporate accounts. This class implements the abstraction IAccount and means that the IAccount interface was not needed to be changed. The refined abstractions CreditAccount and SavingAccount are accessed through the IAccount abstraction. His allows more account types to be added in the future if the need for them should arise. As the IAccount abstraction is not coupled to the SavingAccount implementation, CreditAccount and SavingAccount can easily be modified or deleted if needed without effecting the whole program. This makes the software code more maintainable.

The implementor ITransaction was created to access the concrete implementor classes Personal and Corporate. With the newly refactored code the client Program is only aware of the abstraction IAccount not the implementor ITransaction. A new concrete implementor such as a DebitAccount can be added but still use the same implementor.

# Benefits of the Bridge Design pattern

The bridge design pattern allows you to introduce new abstractions and implementations independently from each other which supports the open closed principle.

The bridge design pattern allows you to create platform independent classes and apps.

The client code is protected with high level abstraction.

The bridge design pattern allows you to focus on high level logic in the abstraction and on platform details on the implementation. Thus, enforcing the single responsibility principle.

# 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 you 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.

java, D. o., 2017. *Bridge Design Pattern – Structural Patterns.* [Online]   
Available at: https://www.dineshonjava.com/bridge-design-patterns/  
[Accessed 11 08 2021].

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

# Table of Figures

[Figure 1 Steering Wheel 3](#_Toc80714692)

[Figure 15 Bridge Pattern UML 4](#_Toc80714693)

[Figure 10 Initial Solution 6](#_Toc80714694)

[Figure 11 IAccount 6](#_Toc80714695)

[Figure 12 SavingAccount 7](#_Toc80714696)

[Figure 13 Program 7](#_Toc80714697)

[Figure 14 Solution with Inheritance 8](#_Toc80714698)

[Figure 15 Refactor with Bridge Design Pattern 9](#_Toc80714699)

[Figure 16 ITransaction 9](#_Toc80714700)

[Figure 17 Personal and Corporate 10](#_Toc80714701)

[Figure 18 AccountBase 12](#_Toc80714702)

[Figure 19 SavingAccount 13](#_Toc80714703)

[Figure 20 Program 13](#_Toc80714704)

[Figure 21 CreditAccount 14](#_Toc80714705)

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