Adapter Design Pattern

By: Sajjad Patel

# Overview

The Adapter Design Pattern also known as a Wrapper Design Pattern is a Class and Object Structural Classification. It’s used to convert the interface of a class into an interface the client expects. It gets classes which have incompatible interfaces to work together.

# Motivation

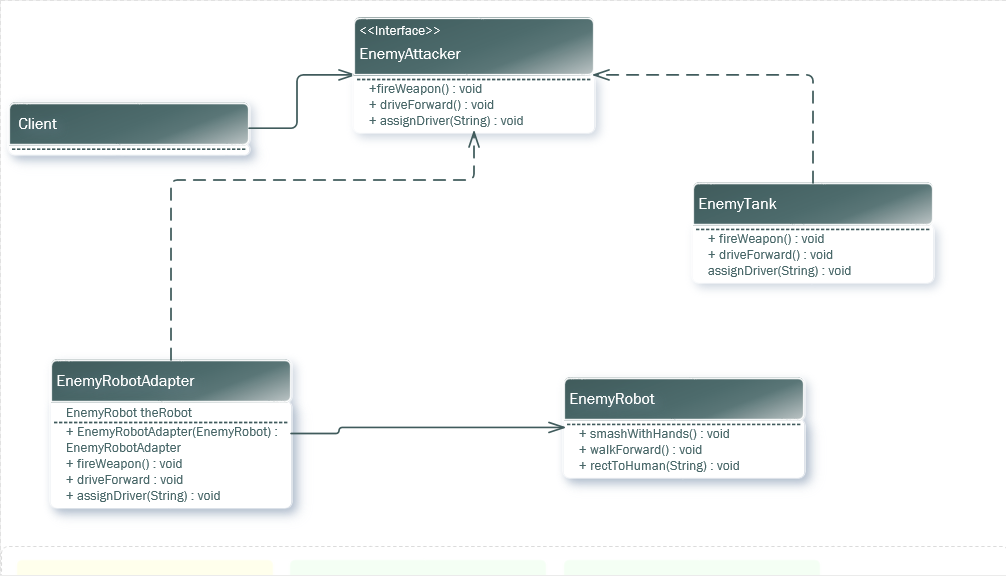
Software for remote controlling a machine is sometimes incompatible with other machines. The problem boils down to the interface requirements for the equipment.

Let’s take for instance a remote-control for enemy robot and enemy tank. Both machines have similar functionality as in they both move forward and back, left and right, and they both attack. Let’s say I make software for the remote-control enemy tank, it won’t be compatible with the enemy robot. The first thing that comes to mind is that they are two completely different machines and the software will not work on both. That’s right but the reason for this problem is that the interface of enemy tank is incompatible with enemy robot. Enemy tank uses an interface named EnemyAttacker, it has methods designed for enemy tank and not enemy robot. An enemy robot walks forward and back while an enemy tank drives forward and back. An enemy robot smashes while an enemy tank shoots.

We can make enemy robot implement EnemyAttacker interface, but this does not make any sense. Methods such as driveForward() and driveBackward() should not be implemented in the enemy robot class because an enemy robot does not drive.

We could however have another class like RobotAdapter which is an adapter that will make the enemy robot interface compatible with the EnemyAttacker. There are two ways of using the Adapter class. You can inherit both EnemyAttacker (interface) and enemy robot’s implementation. Or you can make an enemy robot instance inside the RobotAdapter and implement it per enemy robot’s interface.

In summary the Adapter class is responsible for the functionality the adapted class does not provide in this case EnemyAttacker, and making it work with the Adaptee which in this case is enemy robot.



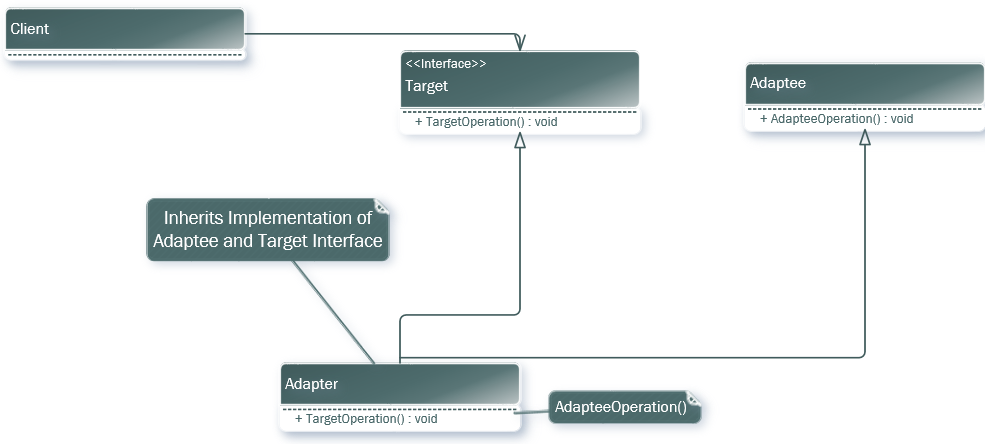
# Applicability

The Adapter Design Pattern is supposed to be used under three conditions. First when you want to use an existing class which has compatibility issues with the interface. Second when you want to create a class that will be reusable by classes that will have compatibility issues with its interface. Lastly when you want to use several subclasses. This is only for Object Adapters as it has the capability of adapting the interface of the parent class.

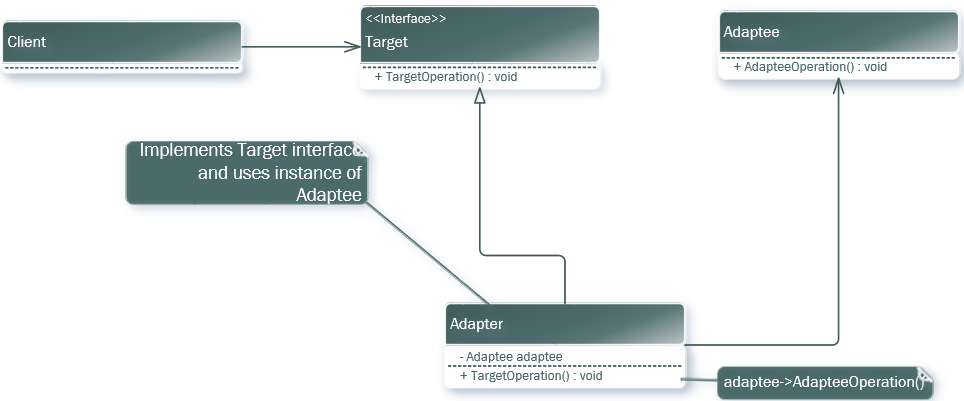
# Structure

There are many parts to the structure of an Adapter Design Pattern. You have the Target, Client, Adaptee and Adapter. You also have Class Adapter and Object Adapter structures which use all the same parts ,however the Class Adapter inherits from both the Adapter and Adaptee which the Object Adapter implements from the Adapter and uses an instance of the Adaptee.

## Class Adapter



## Object Adapter



The Target defines the interface that the client uses. The Client works with objects that implement the Target Interface. The Adaptee defines an interface that needs an Adapter to make it work with the Target interface. The Adapter takes the Adaptee, fixes the compatibility issues it has with the Target interface and makes them work together.

# Consequences

The consequences of using an Adapter Design Pattern is that the Adapter gets a concrete Adaptee to work with the Target interface, so an Adapter will not work when you want to adapt both class and subclass. You can also override the Adaptee’s functionality because the Adapter is a subclass of Adaptee.

# Implementation

## Class Adapters

When implementing the Adapter Design Pattern in C++ using the Class Adapter structure the Adapter will inherit publicly from the Target and privately from the Adaptee. This will make the Adapter a subtype of Target not Adaptee.

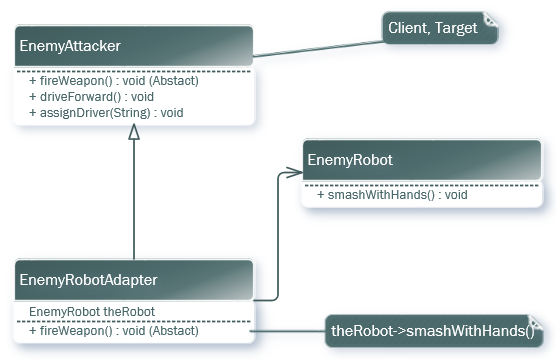
## Pluggable Adapters

When implementing Pluggable Adapter, the first thing to do is find an interface which is small for Adaptee. This helps when making the Adapter because when you have a small amount of operations it is easier to do the adapting.

This will then open three different implementation paths. Abstract Operation, Delegate Operation and Parameterized Adapters.

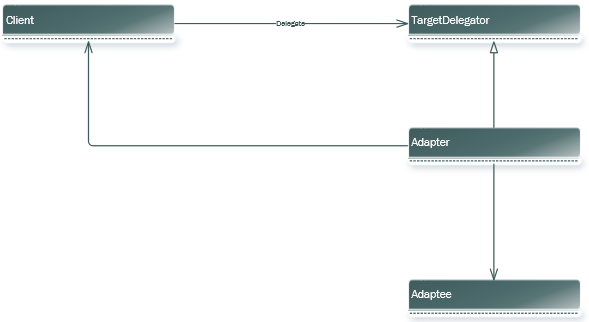
### Abstract Operation

When using the Adapter Operation approach, you need to define Abstract Operations for the Adaptee interface. Any class that then inherit from this class which is both the Target and Client then implements these operations using an instance of the Adaptee.



### Delegate Object

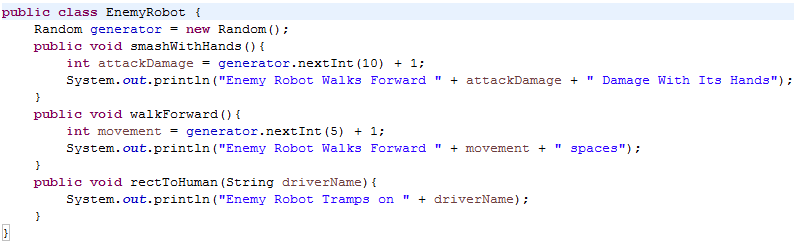
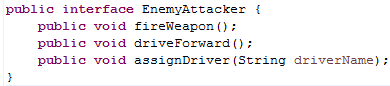
When using Delegate Object approach, the client will forward all requests for using the hierarchical structure to a delegate object. The way this is done is that the Target class is an abstract class that defines the small amount of abstract methods the client needs which the Adapter inherits from. The Adapter will then use a client object to implement the methods using an instance of the Adaptee.



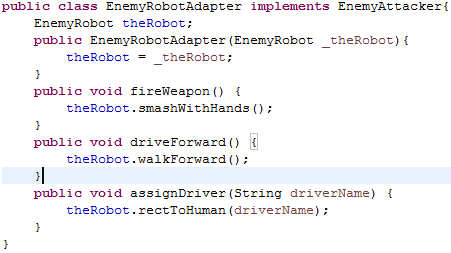
### Parameterized Adapters

When using Parameterized Adapters, you would need to use small talk as this is usually the way of supporting pluggable adapters in it. In this approach, you use one or more blocks. A block will adapt a request and an Adapter stores the blocks for each requests.

# Sample Code



In this sample code, you can see that the EnemyAttacker which in this case is the Target declares methods that will be used in a enemy tank. Now when looking at EnemyRobot which in this case is the Adaptee you see methods that are incompatible with EnemyAttacker.



The EnemyRobotAdapter which is the Adapter uses the Object Adapter structure. Where in it implements EnemyAttacker (Target) and has an instance of EnemyRobot(Adaptee). It then takes the interface of EnemyRobot and makes it compatible with EnemyAttacker.

# Related Patterns

The related patterns to the Adapter Design Pattern are the Bridge Design Pattern, the Decorator Design Pattern and the Proxy Design Pattern.

## Bridge Design Pattern

The Bridge Design Pattern is similar, but it separates the interface from the implementation. Whereas an Adapter is used to change the existing interface.

## Decorator Design Pattern

The Decorator Design Pattern is different because it enhances objects without changing its interface. It is also more transparent.

## Proxy Design Pattern

The Proxy Design Pattern surrogates another object without changing its interface.

# References

<http://www.newthinktank.com/2012/09/adapter-design-pattern-tutorial/>

Design Patterns: Elements of Reusable Object-Oriented Software

Elements of Reusable Object-Oriented Software

by [Erich Gamma,](https://www.kobo.com/ca/en/search?query=Erich%20Gamma&fcsearchfield=Author) [Richard Helm,](https://www.kobo.com/ca/en/search?query=Richard%20Helm&fcsearchfield=Author) [Ralph Johnson,](https://www.kobo.com/ca/en/search?query=Ralph%20Johnson&fcsearchfield=Author) [John Vlissides](https://www.kobo.com/ca/en/search?query=John%20Vlissides&fcsearchfield=Author)