Will Shepherd
CSC 403
10/14/2023

Bridge as a Structural Pattern

The bridge is a structural design pattern that allows one to decouple an abstraction or a high-level interface, which is less complicated and detailed than its implementations but nonetheless lays a framework that describes how they should be or work, from its implementations. This means that implementations of that abstraction are able to vary independently, which is particularly useful whenever there is a need to change or extend an implementation without affecting other implementations of the same abstraction. Furthermore, by encapsulating code in an abstraction, it provides a way to more easily manage complex hierarchies of code or different variations of code as implementations of that abstraction. In this way, the abstraction serves almost like a template and its implementations are fleshed out versions of that template; you can think of a form that is filled out by different people – all are of the same abstraction, but their details vary.

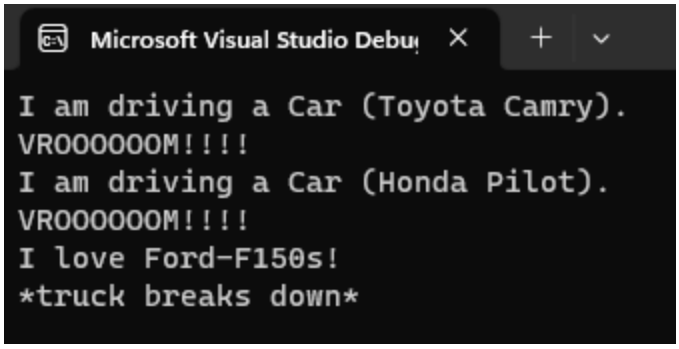
A good example of the bridging design pattern can be seen in how chargers have evolved. Before, chargers were mostly a specific cable fixed to a specific power adapter, usually designed exclusively for just one device. Since then, though, we've begun to see chargers as separated charging cables and power adapters. They all work more or less in the same fashion, connecting one USB-2.0, USB-3.0, USB-C, or similar connective interface to the power adapter, but their implementations vary; some charging cables may support only USB-C to USB-C, while others may support only micro-USB to USB-2.0. This allows a variety of cables to be bridged to a wide range of devices while still adhering to the abstraction that is a charger; overall, this has increased the versatility and functionality of chargers. However, note that for this example, you would likely create multiple abstractions, one for charging cables, power adapters, and chargers, which includes a combination of a compatible charging cable and power adapter.

One obvious advantage to bridging is that – as it was previously mentioned – it allows for the decoupling of abstractions from their implementations; this makes it easier to make changes to one implementation without affecting others. It also increases flexibility and extensibility; more abstractions and implementations can be made to accommodate evolving needs or requirements. Similarly, abstractions can be reused as new implementations that do not already exist or repurposed as new abstractions for eventual implementations. Furthermore, it improves the maintainability of code; in most cases, a problem lies in either the implementation of its abstraction, and bridging makes it easier to identify which may be the culprit long-term.

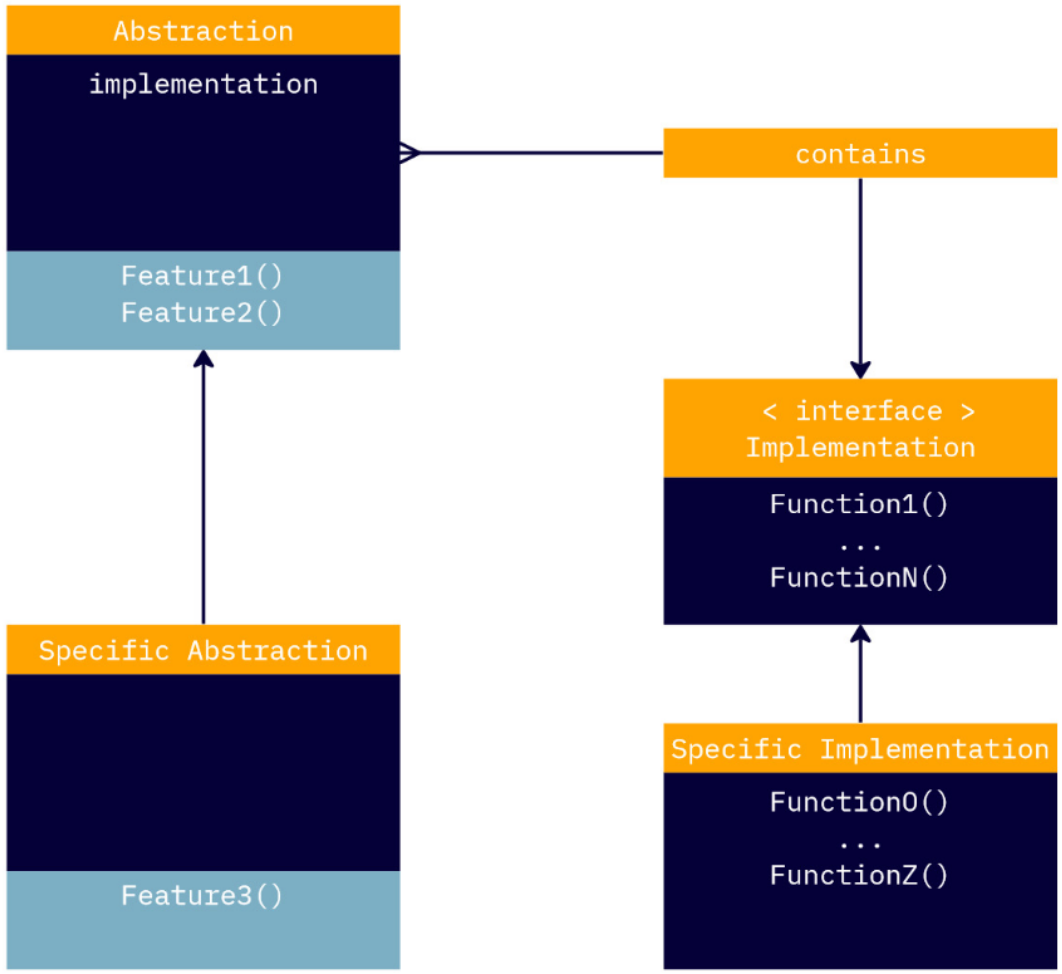
Bridging also has its disadvantages. One such disadvantage is that it may add a layer of complexity to a project; it may not always be easy to identify which abstractions are needed, or it may not even be feasible to create abstractions, especially for smaller projects, wherein there is not a need for many different similar implementations. It also requires more forethought and technical overhead long-term, especially if multiple abstractions are being used in any one implementation. Finally, it can – over time – contribute to an excessive volume of code, which can be tedious for developers to both maintain and keep track of.

Example Code with Output

```
// imports
using System;
using System.Reflection;

// abstraction -- the 'template,' if you will for other types of vehicles, or implementations
// of this abstraction
3 references
public abstract class Vehicle
{
    public string vehicleType;
    public string vehicleMake;
    public string vehicleModel;

    // get vehicle-specific information heres
    2 references
    public Vehicle(string type, string make, string model)
    {
        vehicleType = type;
        vehicleMake = make;
        vehicleModel = model;
    }

    // make sure vehicles have a drive method
    5 references
    public abstract void Drive();
}

// implementation of a vehicle -- car
3 references
public class Car : Vehicle
{
    // send make and model
    2 references
    public Car(string make = "", string model = "") : base("Car", make, model) { }

    // implementation of the abstract drive method from the vehicle abstraction
    4 references
    public override void Drive()
    {
        Console.WriteLine("I am driving a " + vehicleType + " (" + vehicleMake + " " + vehicleModel + ").");
        Console.WriteLine("VROOOOOOM!!!!");
    }
}



// implementation of a vehicle -- truck
2 references
public class Truck : Vehicle
{
    // send make and model
    1 reference
    public Truck(string make = "", string model = "") : base("Truck", make, model) { }

    // implementation of the abstract drive method from the vehicle abstraction
    4 references
    public override void Drive()
    {
        // they're always ford-f150s -- let's be real
        Console.WriteLine("I love Ford-F150s!");
        Console.WriteLine("*truck breaks down*");
    }
}

// main part of the program for testing
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        // instantiation of different types of vehicles --
        // 2 cars and 1 truck
        Vehicle carOne = new Car("Toyota", "Camry");
        Vehicle carTwo = new Car("Honda", "Pilot");
        Vehicle truck = new Truck("Ford", "F-150");

        // call drive method for each vehicle
        carOne.Drive();
        carTwo.Drive();
        truck.Drive();
    }
}
```


Microsoft Visual Studio Debu
I am driving a Car (Toyota Camry).
VROOOOOOM!!!!
I am driving a Car (Honda Pilot).
VROOOOOOM!!!!
I love Ford-F150s!
truck breaks down

UML Diagram


Abstraction
implementation
Feature1()
Feature2()
contains
< interface >
Implementation
Function1()
...
FunctionN()
Specific Abstraction
Feature3()
Specific Implementation
Function0()
...
FunctionZ()