Open/Closed Principle

Definition

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification“

Source: Meyer, Bertrand: Object-Oriented Software Construction

Why I chose Open/Closed Principle

- Definition sounds deceptively simple, but what does it imply?
- It was the most violated principle in our code smell exercise
- Jon Skeet finds it hard to understand

Source: https://codeblog.jonskeet.uk/2013/03/15/the-open-closed-principle-in-review/

Alternative definition

- It should be possible to change the behavior of a method without editing its source code

Source: https://en.wikipedia.org/wiki/Open%E2%80%93closed_principle

Why Should Code Be Closed to Modification?

- Less likely to introduce bugs in code we don’t touch or deploy
- Less likely to break dependent code when we don’t have to deploy updates
- Fewer conditionals in code that is open to extension results in simpler code
- Bug fixes are ok
- Modifications during development are ok

Source: https://www.pluralsight.com/courses/csharp-solid-principles

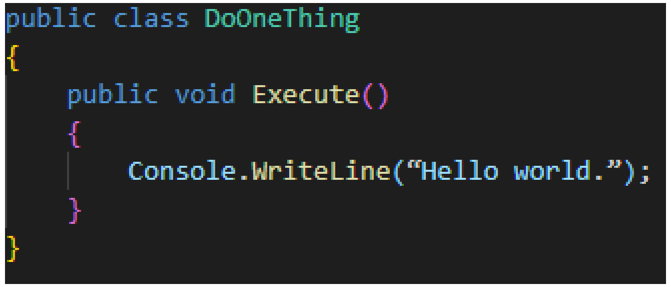
What does “Open for extension” mean?

- New functionality can be added as modules
- Pluggable code
- Similar in a way to extensions in Chrome, Visual Studio

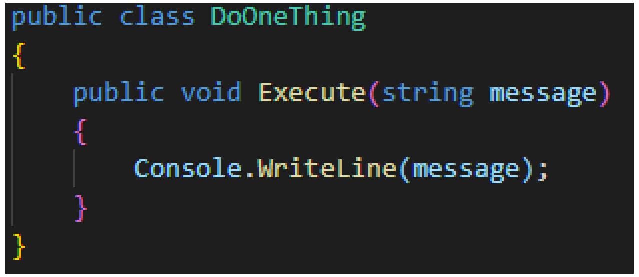
Fixing OCP Violations

- Parameters
- Inheritance
- Composition / injection

Parameters



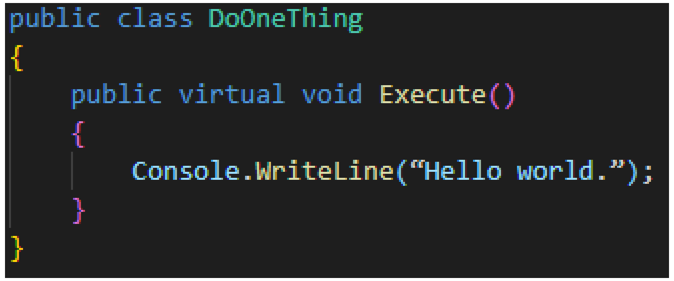
```
public class DoOneThing
{
    public void Execute()
    {
        Console.WriteLine(“Hello world.”);
    }
}
```



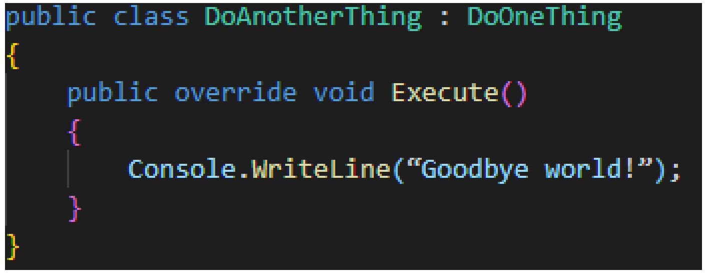
```
public class DoOneThing
{
    public void Execute(string message)
    {
        Console.WriteLine(message);
    }
}
```

Source: https://www.pluralsight.com/courses/csharp-solid-principles

Inheritance



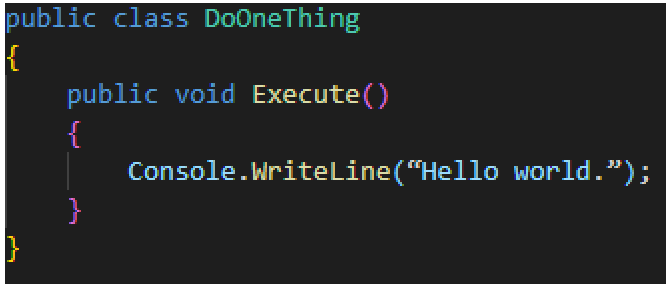
```
public class DoOneThing
{
    public virtual void Execute()
    {
        Console.WriteLine(“Hello world.”);
    }
}
```



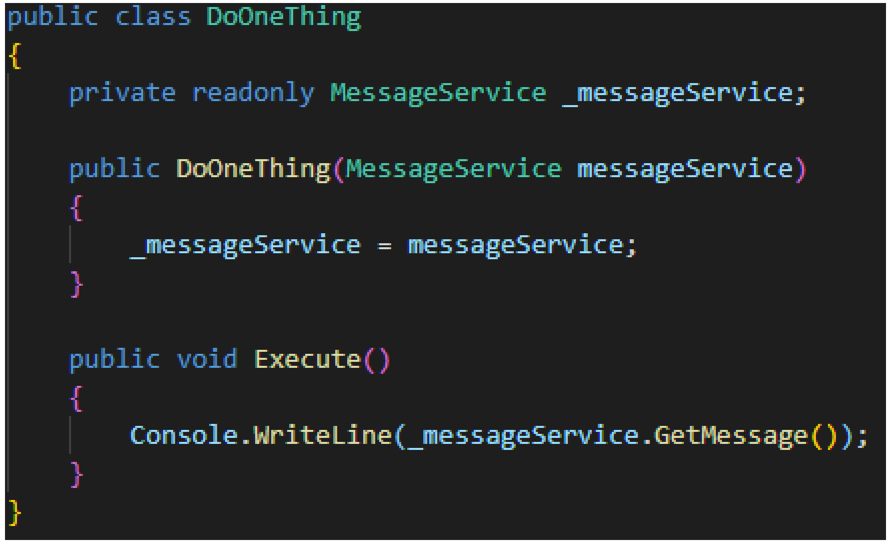
```
public class DoAnotherThing : DoOneThing
{
    public override void Execute()
    {
        Console.WriteLine(“Goodbye world!”);
    }
}
```

Source: https://www.pluralsight.com/courses/csharp-solid-principles

Composition / Injection

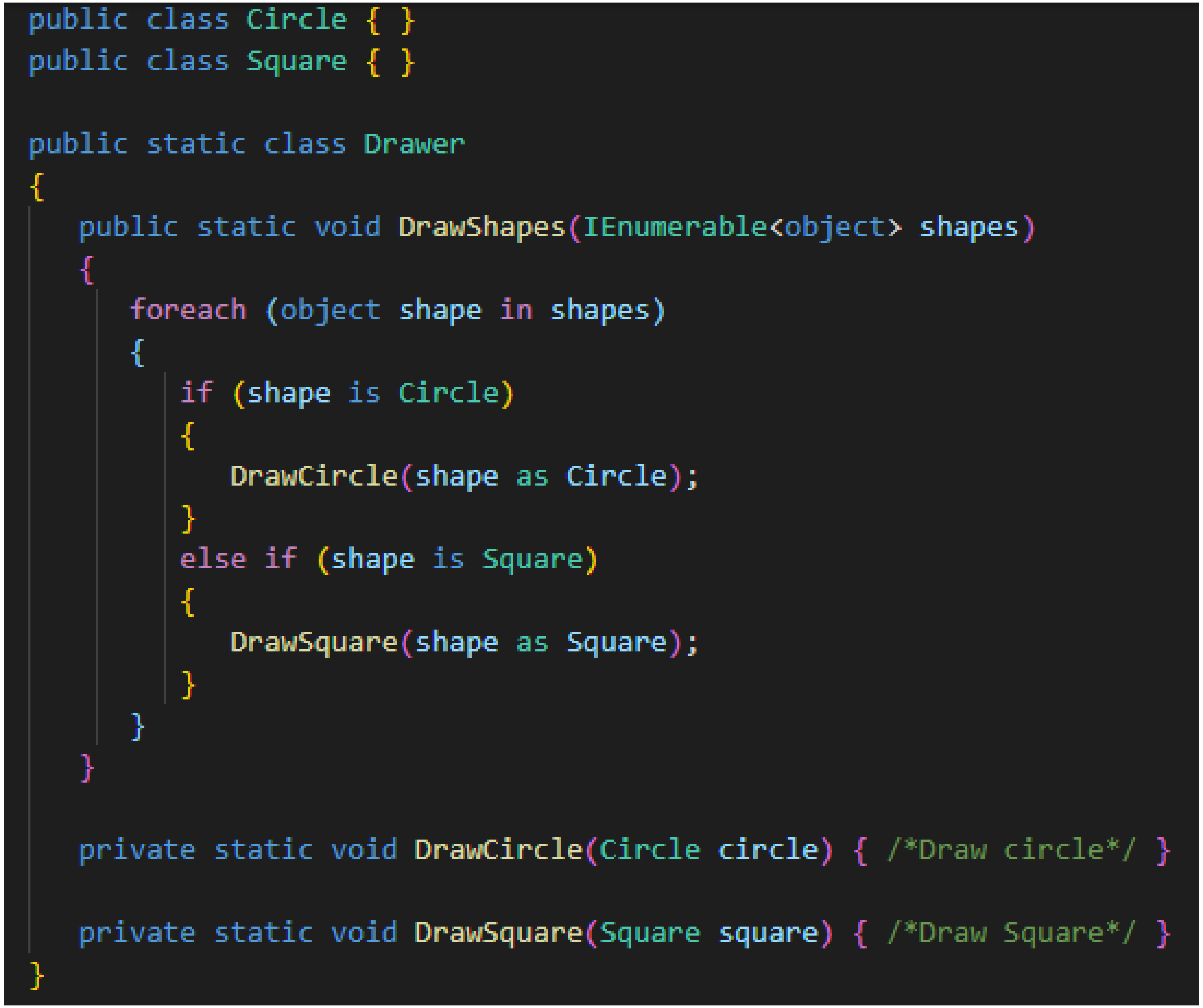


```
public class DoOneThing
{
    public void Execute()
    {
        Console.WriteLine(“Hello world.”);
    }
}
```



```
public class DoOneThing
{
    private readonly MessageService _messageService;

    public DoOneThing(MessageService messageService)
    {
        _messageService = messageService;
    }

    public void Execute()
    {
        Console.WriteLine(_messageService.GetMessage());
    }
}
```

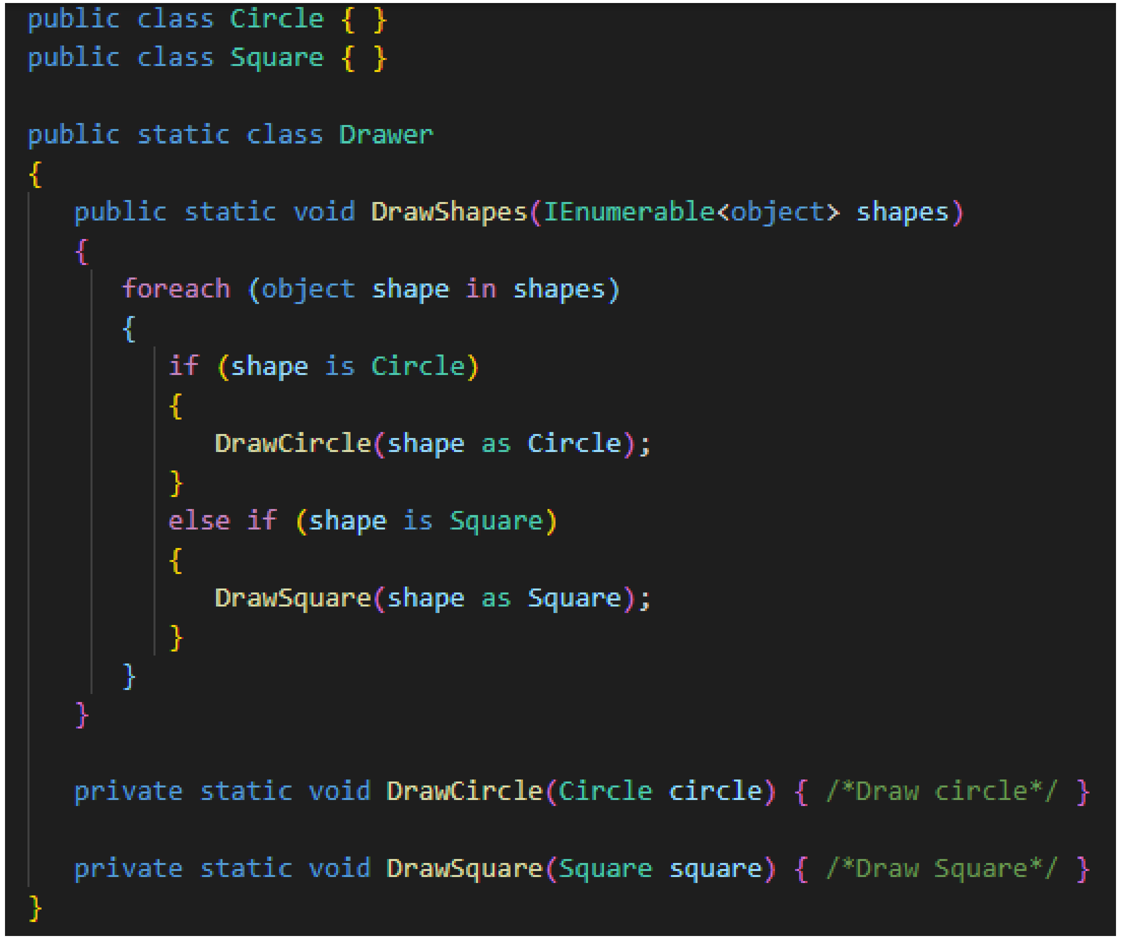
Source: https://www.pluralsight.com/courses/csharp-solid-principles



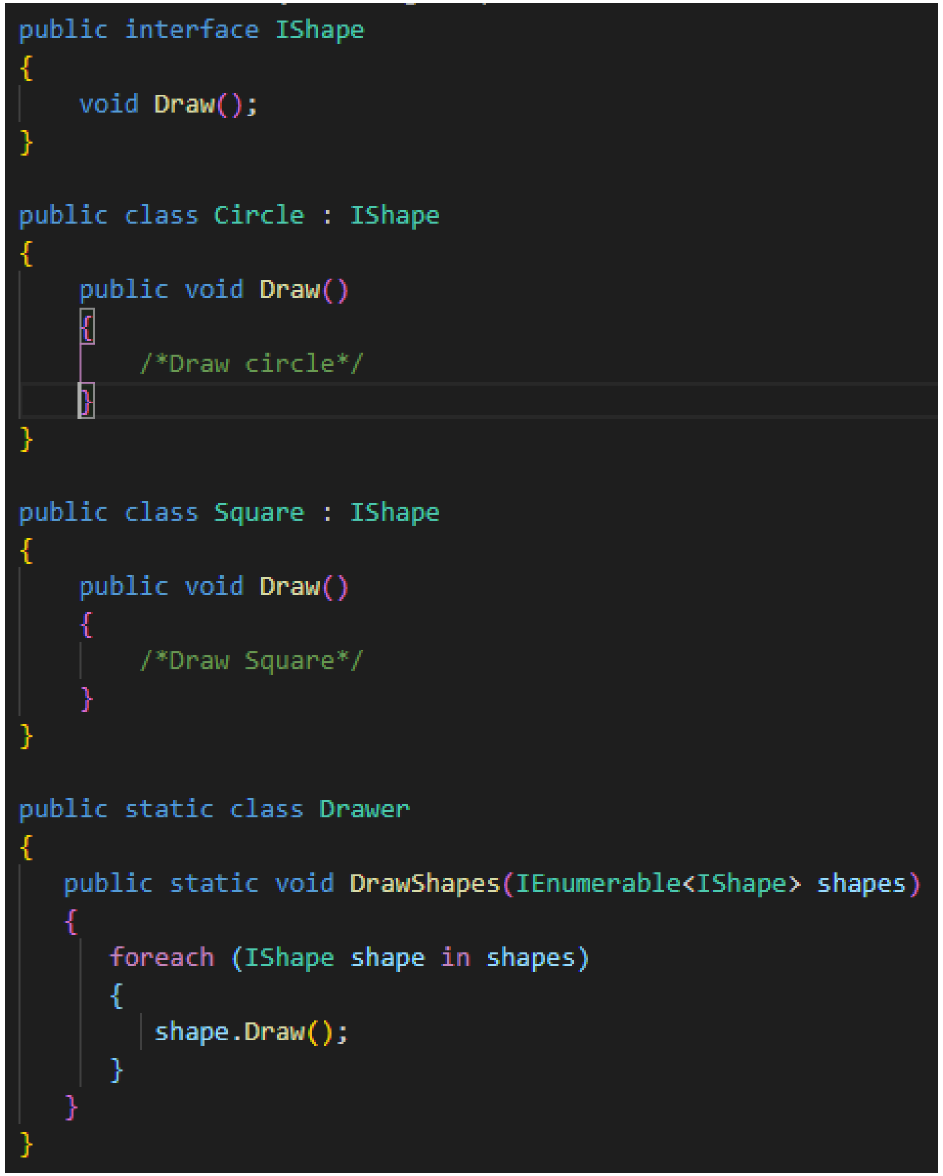
```
public class Circle { }
public class Square { }

public static class Drawer
{
    public static void DrawShapes(IEnumerable<object> shapes)
    {
        foreach (object shape in shapes)
        {
            if (shape is Circle)
            {
                DrawCircle(shape as Circle);
            }
            else if (shape is Square)
            {
                DrawSquare(shape as Square);
            }
        }
    }

    private static void DrawCircle(Circle circle) { /*Draw circle*/ }

    private static void DrawSquare(Square square) { /*Draw Square*/ }
}
```

Source: https://blog.ndepend.com/solid-design-the-open-close-principle-ocp/

Strategy pattern



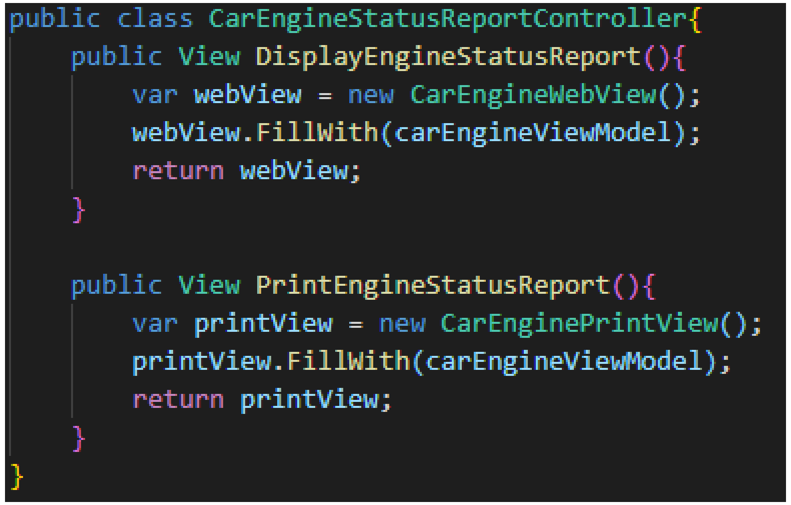
```
public class Circle { }
public class Square { }

public static class Drawer
{
    public static void DrawShapes(IEnumerable<object> shapes)
    {
        foreach (object shape in shapes)
        {
            if (shape is Circle)
            {
                DrawCircle(shape as Circle);
            }
            else if (shape is Square)
            {
                DrawSquare(shape as Square);
            }
        }
    }

    private static void DrawCircle(Circle circle) { /*Draw circle*/ }

    private static void DrawSquare(Square square) { /*Draw Square*/ }
}
```



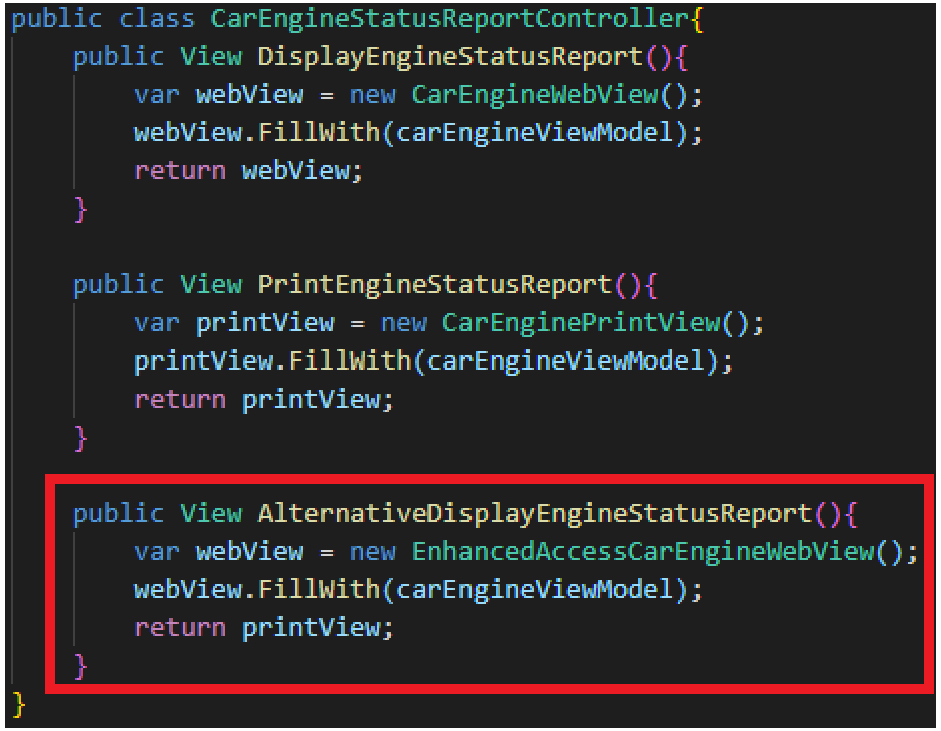
```
public interface IShape
{
    void Draw();
}

public class Circle : IShape
{
    public void Draw()
    {
        /*Draw circle*/
    }
}

public class Square : IShape
{
    public void Draw()
    {
        /*Draw Square*/
    }
}

public static class Drawer
{
    public static void DrawShapes(IEnumerable<IShape> shapes)
    {
        foreach (IShape shape in shapes)
        {
            shape.Draw();
        }
    }
}
```

Source: https://blog.ndepend.com/solid-design-the-open-close-principle-ocp/

Strategy – Example II



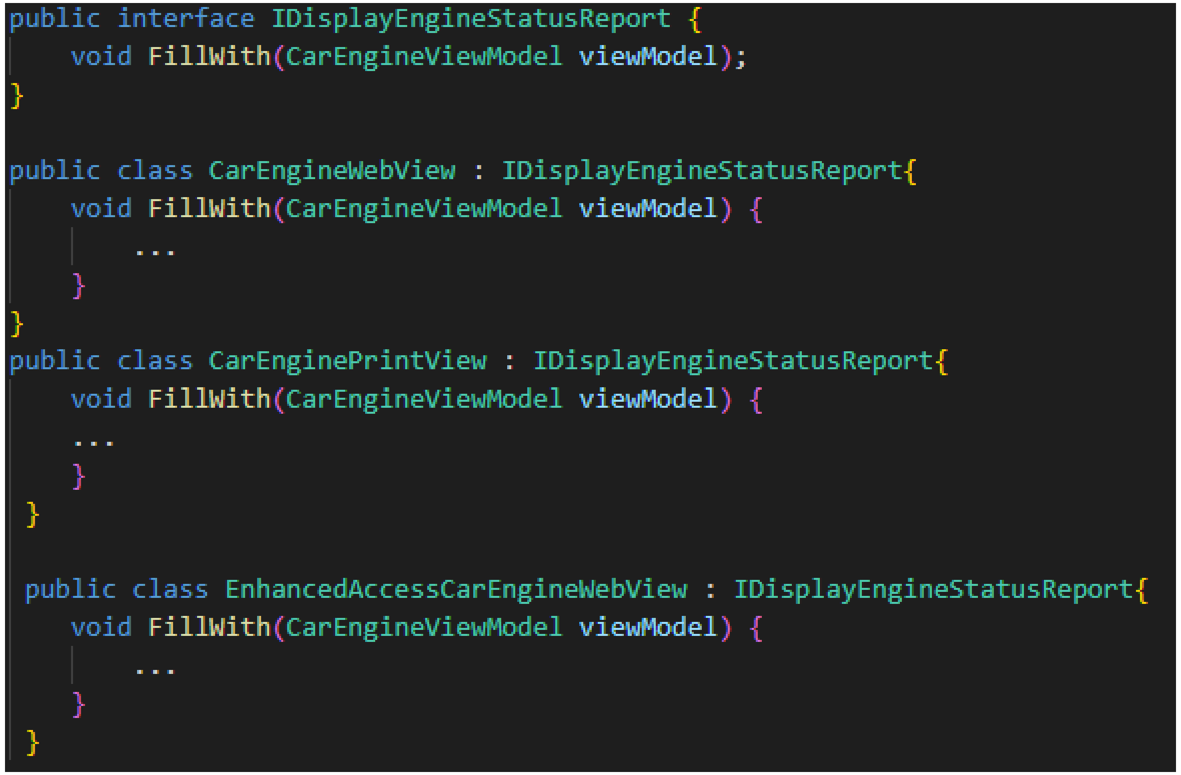
```
public class CarEngineStatusReportController{
    public View DisplayEngineStatusReport(){
        var webView = new CarEngineWebView();
        webView.FillWith(carEngineViewModel);
        return webView;
    }

    public View PrintEngineStatusReport(){
        var printView = new CarEnginePrintView();
        printView.FillWith(carEngineViewModel);
        return printView;
    }
}
```



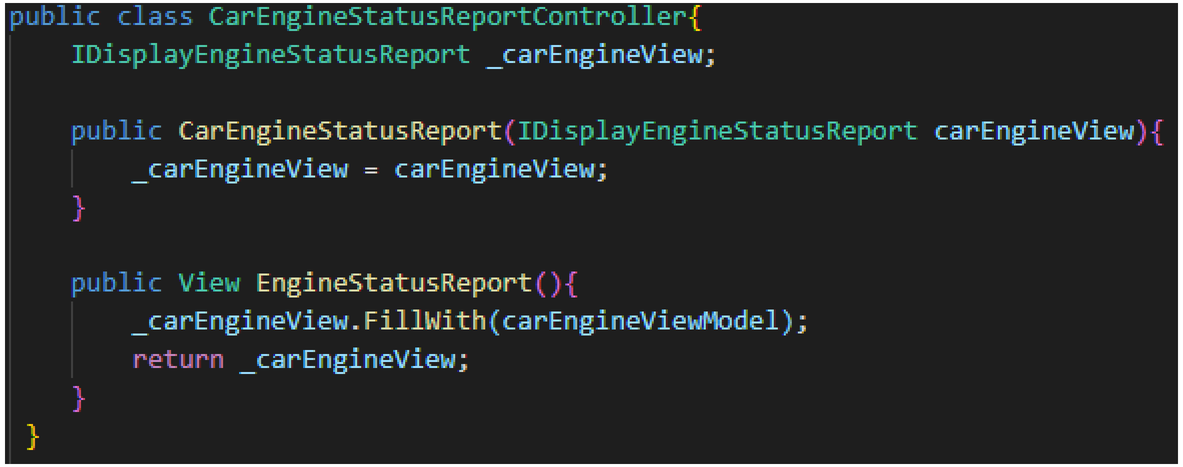
```
public class CarEngineStatusReportController{
    public View DisplayEngineStatusReport(){
        var webView = new CarEngineWebView();
        webView.FillWith(carEngineViewModel);
        return webView;
    }

    public View PrintEngineStatusReport(){
        var printView = new CarEnginePrintView();
        printView.FillWith(carEngineViewModel);
        return printView;
    }

    public View AlternativeDisplayEngineStatusReport(){
        var webView = new EnhancedAccessCarEngineWebView();
        webView.FillWith(carEngineViewModel);
        return printView;
    }
}
```

Source: Agile Technical Practices Distilled (Santos, Consolaro, Di Gioia)

Making it OCP compliant



```
public interface IDisplayEngineStatusReport {
    void FillWith(CarEngineViewModel viewModel);
}

public class CarEngineWebView : IDisplayEngineStatusReport{
    void FillWith(CarEngineViewModel viewModel) {
        ...
    }
}
public class CarEnginePrintView : IDisplayEngineStatusReport{
    void FillWith(CarEngineViewModel viewModel) {
    ...
    }
 }

 public class EnhancedAccessCarEngineWebView : IDisplayEngineStatusReport{
    void FillWith(CarEngineViewModel viewModel) {
        ...
    }
 }
```



```
public class CarEngineStatusReportController{
    IDisplayEngineStatusReport _carEngineView;

    public CarEngineStatusReport(IDisplayEngineStatusReport carEngineView){
        _carEngineView = carEngineView;
    }

    public View EngineStatusReport(){
        _carEngineView.FillWith(carEngineViewModel);
        return _carEngineView;
    }
 }
```

Source: Agile Technical Practices Distilled (Santos, Consolaro, Di Gioia)

Why Use a New Class?

- Design class to suit problem at hand
- Nothing in current system depends on it
- Can add behavior without touching existing code
- Can follow Single Responsibility Principle
- Can be unit-tested

Source: https://www.pluralsight.com/courses/csharp-solid-principles

OCP vs YAGNI

- YAGNI prohibits changing the existing functionality to account for possible new features in the future
- OCP is about accounting for possible new features in the future

- Risk of over-engineering a wrong abstraction
- But failure to identify variation over a common pattern may lead to lots of breaking changes and code smells like Shotgun Surgery and Divergent change

- We have to balance these two forces

Sources: https://enterprisecraftsmanship.com/posts/ocp-vs-yagni/,
Agile Technical Practices Distilled (Santos, Consolaro, Di Gioia)

OCP vs YAGNI - Rule of Three

- Start concrete
- Modify the code the first time or two
- By the third modification, consider making the code open to extension for that axis of change

Source: https://www.pluralsight.com/courses/csharp-solid-principles

Protected Variation Pattern

“Identify points of predicted variation and create a stable interface around them”

Source: https://martinfowler.com/ieeeSoftware/protectedVariation.pdf

Review - Why use Open/Closed Principle

- Reduces risk when introducing new functionality
- Simplifies introducing new functionality
- Future developments of new functionality becomes much faster than before

- Beware of YAGNI and Rule of Three