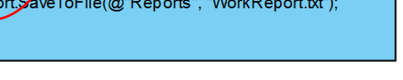
Single-Responsibilty

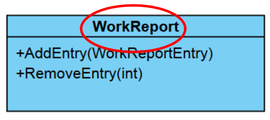
- Es sollte niemals mehr als einen Grund geben, eine Klasse zu verändern
- Viele kleine Klassen sind besser als wenige große
- Umsetzung um Fehler zu vermeiden und eine klare Strukturierung zu bekommen

Annahme: Es sollen Reporte geschrieben und als Files gespeichert werden.

```
// BEFORE
static void Main(string[] args)
    {
        var report = new WorkReport();
        report.AddEntry(new WorkReportEntry{ ... });
        report.AddEntry(new WorkReportEntry{ ... });

        report.SaveToFile(@"Reports", "WorkReport.txt");
    }
```



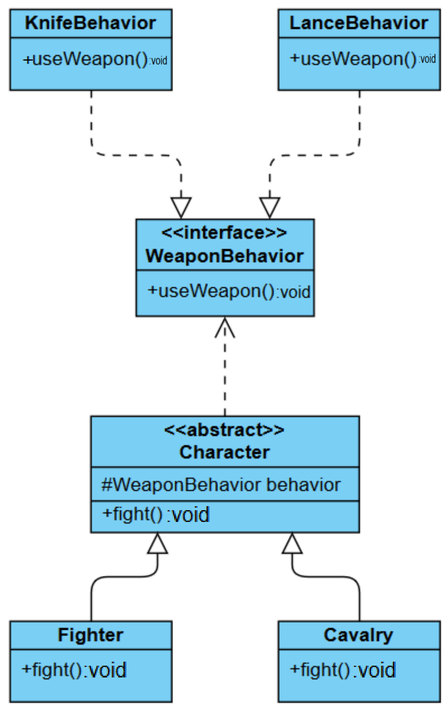

```
// AFTER
static void Main(string[] args)
    {
        var report = new WorkReport();
        report.AddEntry(new WorkReportEntry{ ... });
        report.AddEntry(new WorkReportEntry{ ... });

        var saver = new FileSaver();
        saver.SaveToFile(@"Reports", "WorkReport.txt", report);
    }
```



Open-Closed

- Eine Klasse soll offen für Erweiterung, aber geschlossen für Veränderung sein
- Umsetzung um Fehler in fertig implementierten Klassen zu vermeiden

Annahme: Es gibt verschiedene Charaktere, welche eine Waffe besitzen. Die Waffen haben eine Funktion zum Ausführen, welche je nach Waffe ein anderes Verhalten hat.



Veränderungen in den bestehenden Klassen müssen nicht mehr vorgenommen werden. Soll das Verhalten der Charaktere im Kampf geändert werden, kann man eine andere Waffe über das Interface auswählen oder neue Waffen mit eigenem Verhalten hinzufügen

Liskovsche-Substitution

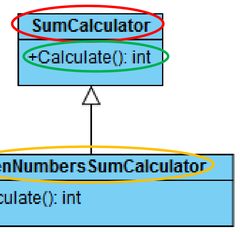
- Eine Unterklasse muss immer alle Eigenschaften der Oberklasse erfüllen und darf sie nicht verändern
- Abgeleitete Klassen müssen immer anstelle der Basisklasse einsetzbar sein

Annahme: Es sollen Werte zusammengerechnet werden. Weiter sollen die dafür geltenden Werte z.B. nach geraden Zahlen gefiltert werden.

Aufgrund der Vererbung sollte es möglich sein, eine Referenz der Unterklasse in der Oberklasse zu speichern:

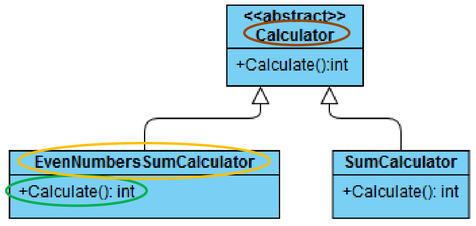
```
SumCalculator evenSum = new EvenNumbersSumCalculator(numbers);
Console.WriteLine($"The sum of all the even numbers: {evenSum.Calculate()}");
```

Hierbei würde ein falsches Ergebnis herauskommen, da die Methode der Oberklasse anstatt der Unterklasse aufgerufen werden würde.



Durch die Verwendung einer abstrakten Oberklasse bleiben die Eigenschaften der Klassen identisch und die Speicherung der Unterklassenreferenz in einem Objekt der Oberklasse ist möglich:

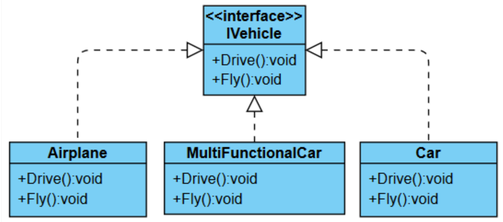
```
Calculator evenSum = new EvenNumbersSumCalculator(numbers)
Console.WriteLine($"The sum of all the even numbers: {evenSum.Calculate()}");
```



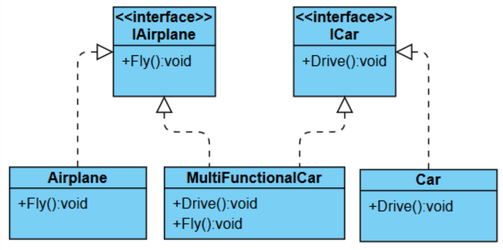
Interface-Segregation

- Zu große Interfaces sollen in mehrere aufgeteilt werden
- Ein Interface soll nur Funktionen enthalten, die eng zusammengehören

Annahme: Verschiedene Transportmittel die Fliegen und/oder Fahren können.



Bei einem MultiFunctionalCar wäre diese Variante akzeptabel. Für ein normales Car oder Airplane wären überflüssige Funktionen mit einem NotImplementedError o.ä. zu implementieren.

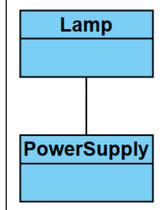


Wendet man das ISP an, so sind nur die Funktionen in den Interfaces enthalten, welche tatsächlich auf die Klassen anwendbar sind. Im Falle des MultiFunctionalCar werden mehrere Interfaces implementiert.

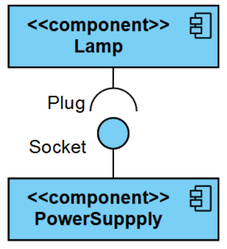
Dependancy-Inversion

- Hohe Module sollen nicht von niedrigeren Modulen abhängig sein
- Stabile Abstraktionen > flüchtige Konkretionen

Annahme: Es existiert eine Lampe, welche mit dem Stromnetz verbunden sein soll.



In disem Fall hätten wir eine Konkretion, welche sich aufgrund von Änderungen der Verbindung zum Netz ändern kann (z. B. Umzug).



In diesem Fall hätten wir eine Abstraktion, durch welche man das Stromnetz bzw. die Steckdose flexibel wechseln kann.

Quellen:
https://code-maze.com/solid-principles/
https://de.wikipedia.org/wiki/Prinzipien_objektorientierten_Designs

Github Link: