# Single-Responsibilty

Es sollte niemals mehr als einen Grund geben, eine Klasse zu verändern.

Viele kleine Klassen sind besser als wenige große.

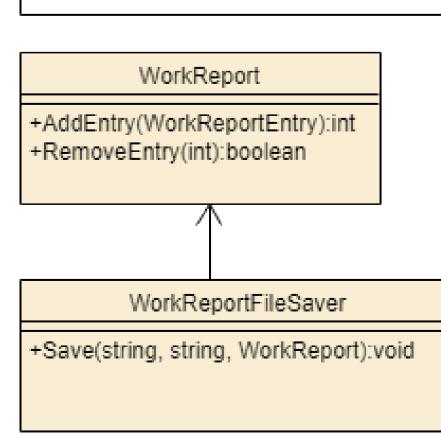
Umsetzung um Fehler zu vermeiden und eine klare Strukturierung zu bekommen.

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

#### //BEFORE \$report = new WorkReport(); \$report.AddEntry(new WorkReportEntry(...)); \$report.AddEntry(new WorkReportEntry(...)); \$report.SaveToFile("reports", "workReport.txt");

#### +AddEntry(WorkReportEntry):int +RemoveEntry(int):boolean +SaveToFile(string, string): void

```
//AFTER
$report = new WorkReport();
$report.AddEntry(new WorkReportEntry(...));
$report.AddEntry(new WorkReportEntry(...));
$saver = new WorkReportFileSaver();
$saver.Save("reports", "workReport.txt", $report);
```

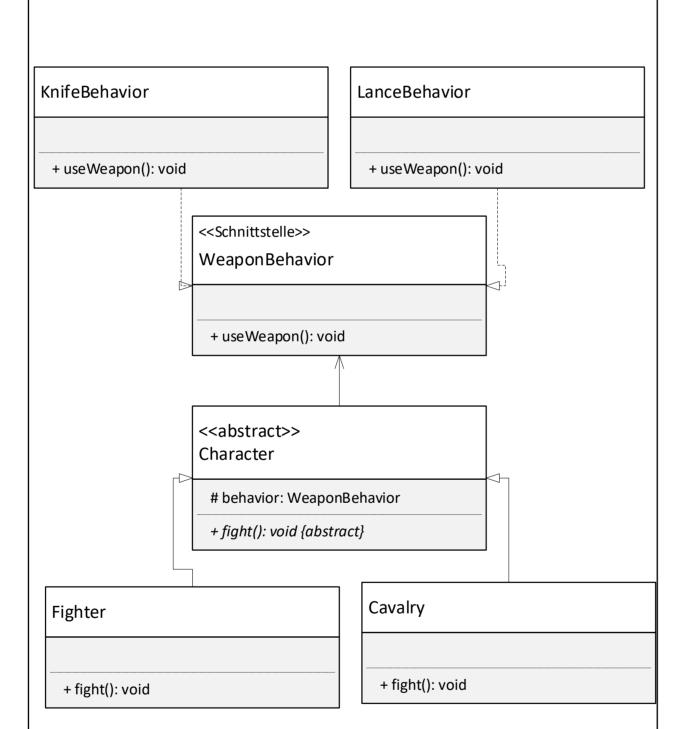


### Open-Closed

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.

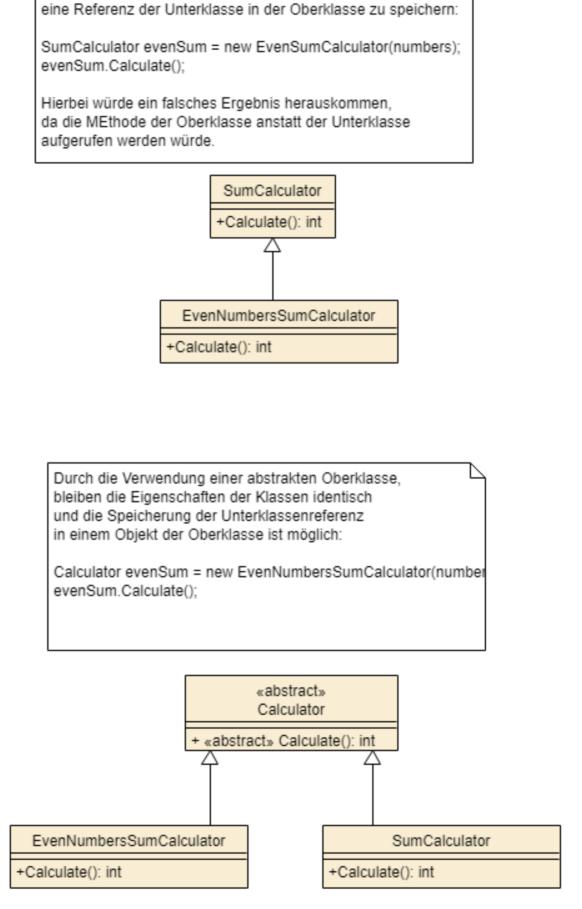
#### Liskov-Substitution

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.

ufgrund der Vererbung sollte es möglich sein,

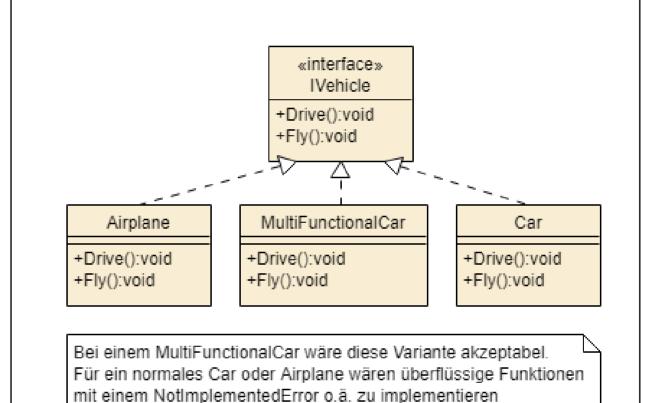


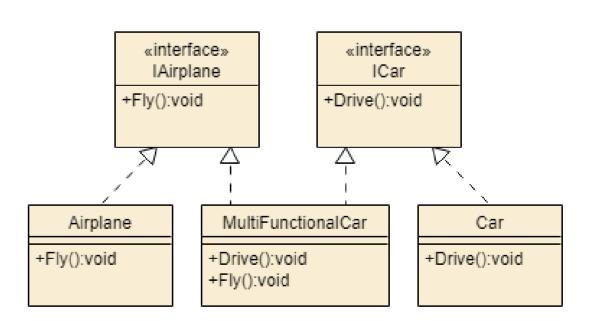
## nterface-Segregation

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.





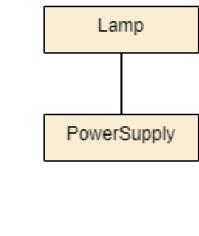
Wendet man das ISP an, so sind nur die Funktionen in den Interfaces enthalten, welche tatsächlich auf die Klassen anwendbar sind. Im Falle des MultiFunctionalCar werden mehrere Interfaces implementiert.

#### Dependancy-Inversion

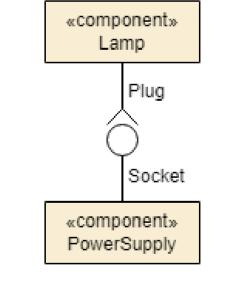
Hohe Module sollen nicht von niedrigeren Modulen abhängig sein.

Stabile Abstraktionen > flüchtige Konkretionen.

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



In diesem 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.

