SOLID für .NET und JavaScript

BINARYGEARS
Software Made Simple

www.binarygears.de | Stefan Dirschnabel

Twitter: @StefanDirschnab | #dncgn

BINARYGEARS
Software Made Simple

SOFTWARE ENTWICKLUNG

BERATUNG

SCHULUNG

5 Prinzipien für das Design von Software

Wartbarer Code

Leicht erweiterbarer Code

Weniger Bugs

BINARYGEARS
Software Made Simple

SOLID

BINARYGEARS
Software Made Simple

Single Responsibility Principle

Open Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Vorgehensweise

BINARYGEARS
Software Made Simple

- Für jedes Prinzip
  - Theorie
  - Beispiel
- Erkenntnisse: Erkennen + Lösen
- Demo: durchgängiges Beispiel / Problemstellung

Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

DIP - Theorie

BINARYGEARS
Software Made Simple

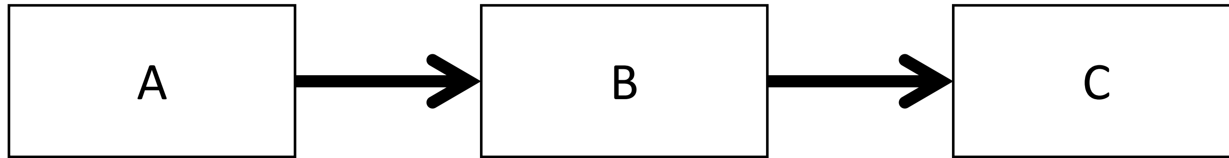
Zur Entkopplung von Programmteilen sollten alle Teile einer Software immer nur von Abstraktionen abhängig sein.

Dies ermöglicht auch die Wiederverwendbarkeit von High-Level Komponenten aus Sicht von Low-Level Komponenten.

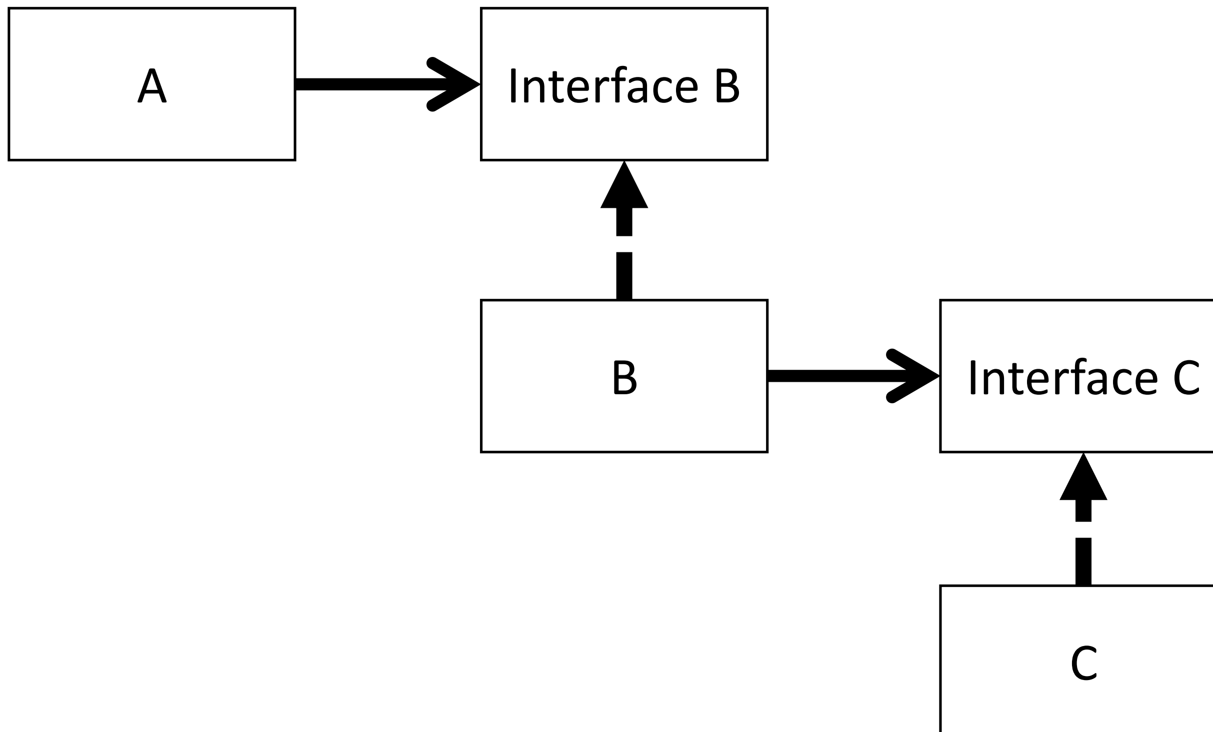
- Abhängigkeiten nur zu Interfaces („*depend on abstractions*“)

- Dependency Inversion Principle != Dependency Injection
- Dependendy Inversion Principle != Inversion of Control
- Aber: DI / IoC wird in diesem Zusammenhang verwendet um die Abhängigkeiten aufzulösen

DIP - Theorie



Top Down



Abstraktionen auf Ebene des Verwenders

BINARYGEARS
Software Made Simple

DIP - Beispiel

BINARYGEARS
Software Made Simple

Microsoft
.NET

JS

Liskov Substitution Principle

Sei q(x) eine beweisbare Eigenschaft von Objekten x des Typs T, dann soll q(y) des Typs S wahr sein, wobei S ein Untertyp von T ist.

Liskov Substitution Principle

BINARYGEARS
Software Made Simple

Eine Unterklasse darf keine logischen Probleme erzeugen wenn Sie anstelle Ihrer Superklasse verwendet wird

Liskov Substitution Principle

BINARYGEARS
Software Made Simple

- Einfacher ausgedrückt: „Man sollte anstelle einer Superklasse immer eine Subklasse verwenden können“
- C#.NET: Kovarianz und Kontravarianz Restriktionen per default (Compiler lässt nichts anderes zu)
- JS: Entwickler muss Kovarianz und Kontravarianz Restriktionen einhalten

- Sehr leicht zu verletzen

LSP

BINARYGEARS
Software Made Simple

- D.h. folgendes ist erlaubt:
  - Kovarianz der Methoden-Rückgabewerte in der Subklasse
  - Kontravarianz der Methoden-Parameter in der Subklasse
  - Nur Exceptions werfen die Ableitungen von Exceptions sind die die Superklasse sowieso schon wirft

  - Kontravarianz Parameter: die Subklasse darf in ihren Methoden nur Parameter akzeptieren die Basis-Typen derjenigen Typen sind, welche die Superklasse an dieser Stelle akzeptiert hat. Logisch: ansonsten könnte man B nicht anstelle von A verwenden. Alles was an die Methoden der Superklasse übergeben wurde muss auch an die Methoden der Subklasse übergeben werden können.
  - Kovarianz Rückgabewerte: der Rückgabewerte der Methode darf vom gleichen Typ derjenige der Superklassen-Methode oder einer Ableitung davon sein

LSP - Beispiel

BINARYGEARS
Software Made Simple

Microsoft
.NET

JS

Interface Segregation Principle

Clients should not be forced to depend on methods that they do not use.

ISP - Theorie

BINARYGEARS
Software Made Simple

Mehrere kleine Interfaces sind besser als ein großes Interface, da der Client nicht von Schnittstellen abhängig sein sollte die er nicht verwendet.

- Bei der Realisierung muss nichts unnötig implementiert werden
- Übersichtlicher und sauberer bei der Verwendung

ISP - Beispiel

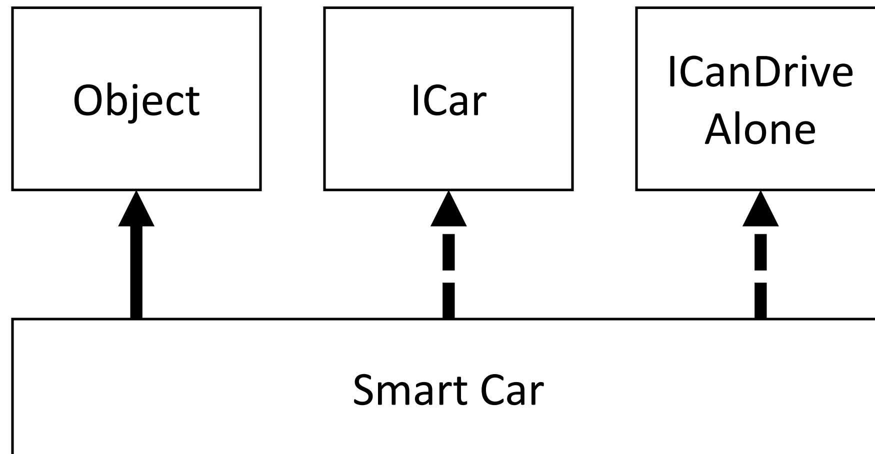
BINARYGEARS
Software Made Simple

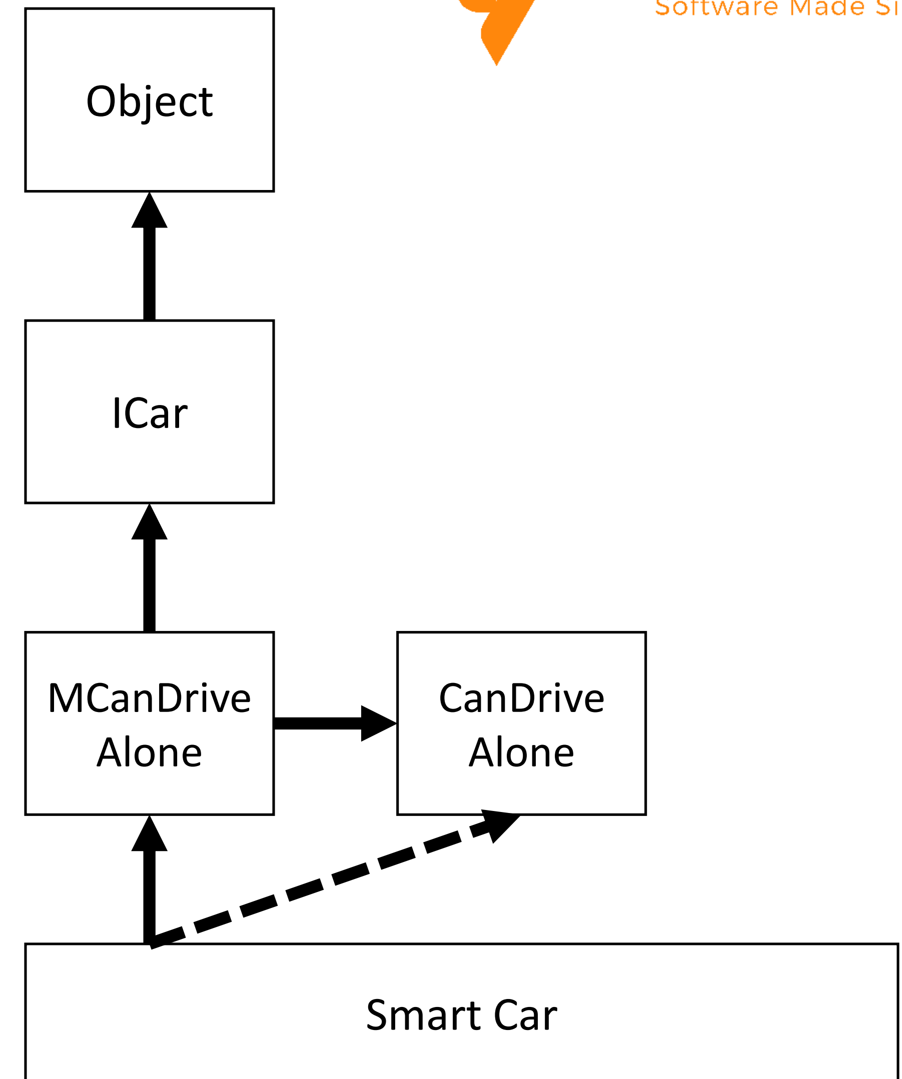
Microsoft
.NET

JS

ISP – Erkennen + Lösen

- C# (typisierte): Interfaces
- JS (dynamisch typisiert): Mixins





BINARYGEARS
Software Made Simple

BINARYGEARS
Software Made Simple

Open Closed Principle

A class should be open for extensions but closed for modifications

OCP - Theorie

BINARYGEARS
Software Made Simple

Einmal getestete Klasse soll nicht immer wieder verändert werden müssen

„Once a class is done, it is done“

OCP - Beispiel

BINARYGEARS
Software Made Simple

Microsoft
.NET

JS

BINARYGEARS
Software Made Simple

Single Responsibility

There is one, and only one reason to change a class

SRP - Theorie

Bei Änderung einer Anforderung sollen sich die Änderungen auch nur auf eine Klasse beschränken.

Wenn eine Klasse mehrere Verantwortlichkeiten hat

- isoliertes Testen von Anforderungen nicht möglich
- Änderungen bewirken leichter Bugs in anderen Features

SRP - Beispiel

BINARYGEARS
Software Made Simple

Microsoft
.NET

JS

BINARYGEARS
Software Made Simple

SOLID schnell erkennen + lösen

SRP – Erkennen + Lösen

BINARYGEARS
Software Made Simple

Erkennen

- Manager Klassen
- Controller Klassen
- Methoden mit „And“ bzw. „Und“ im Namen
- Grenzfall: Repository/ Adapter Klassen

Lösen

- Refactoring: Extract Class
- Refactoring: Extract Method

OCP – Erkennen + Lösen

BINARYGEARS
Software Made Simple

Erkennen

Erweiterungen an Klassen

Hinzufügen von Tests

Typisch bei Validierungs-Klassen

Lösen

Neue Klasse erzeugen

Aggregation / Komposition

Polymorphie / Vererbung

LSP – Erkennen + Lösen

BINARYGEARS
Software Made Simple

Erkennen

Vererbungshierarchien

Vermischung von Daten + Logik

Default Rückgabewerte

Lösen

Aggregation / Komposition

Daten & Logik trennen

Funktionale SW-Architektur

ISP – Erkennen + Lösen

BINARYGEARS
Software Made Simple

Erkennen

Single Responsibility verletzt

Anzahl der Methoden im Interface

Aggregations-Interfaces

Lösen

Extract Interface / Mixin

DIP – Erkennen + Lösen

BINARYGEARS
Software Made Simple

Erkennen

Keine Komponentenorientierung (getrennte Kontrakt Projekte / Bibliotheken)

Abhängigkeiten zu konkreten Typen

Switch-Case

Lösen

Komponentenorientierung einführen

Abhängigkeiten „umkehren“ → besser „umlenken“

Konsequent DI verwenden

Switch-Case mit Klasse / Polymorphie auflösen

Fazit

BINARYGEARS
Software Made Simple

Komplexität unkompliziert gestalten

Software so komplex wie nötig aber
so einfach wie möglich

Fazit

BINARYGEARS
Software Made Simple

Single Responsibility Principle

Open Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

BINARYGEARS
Software Made Simple

SOFTWARE ENTWICKLUNG

BERATUNG

SCHULUNG