

# Vorlesung Softwareentwicklung 2021

<https://github.com/SebastianZug/CsharpCourse>

André Dietrich    Christoph Pooch    Fabian Bär    Fritz Apelt    Galina Rudolf  
JohannaKlinke    Jonas Treumer    KoKoKotlin    Lesestein    LinaTeumer  
MMachel    Sebastian Zug    Snikker123    Yannik Höll    Florian2501    DEVensiv  
fb89zila

## Modellierung von Software

Parameter	Kursinformationen
<b>Veranstaltung:</b>	Vorlesung Softwareentwicklung
<b>Semester:</b>	Sommersemester 2021
<b>Hochschule:</b>	Technische Universität Freiberg
<b>Inhalte:</b>	Motivation der Modellierung von Software
<b>Link auf den GitHub:</b>	<a href="https://github.com/TUBAF-Ifi-LiaScript/VL_Softwareentwicklung/blob/master/13_UML_Modellierung.md">https://github.com/TUBAF-Ifi-LiaScript/VL_Softwareentwicklung/blob/master/13_UML_Modellierung.md</a>
<b>Autoren</b>	@author

## Neues aus Github

Die erste Aufgabe unter Zuhilfenahme von git / GitHub ist angelaufen. Setzen Sie sich in dieser Woche, soweit das noch nicht geschehen ist, intensiv damit auseinander! Die Techniken sind von zentraler Bedeutung für die weiteren Aufgabenblätter.

Anmerkungen:

- Versehen Sie Ihre Commits mit aussagekräftigen Bezeichnungen -> [Anleitung](#)
- Beenden Sie das Projekt mit der Veröffentlichung eines Release!

Tragen Sie bitte Ihre Fragebogenschlüssel in die Datei team.config ein. Dies hilft bei der wissenschaftlichen Aufbereitung der Daten ungemein.

## Motivation des Modellierungsgedankens

Um gedanklich wieder in die C# Entwicklung einzutauchen, finden Sie in dem Ordner [code](#) zwei Beispiele für die:

- Nutzung abstrakter Klassen
- Verwendung von Interfaces

Überlegen Sie sich alternative Lösungsansätze mit Vor- und Nachteilen für die beschriebenen Implementierungen.

## Prinzipien des (objektorientierten) Softwareentwurfs

**Merke:** Software lebt!

- Prinzipien zum Entwurf von Systemen
- Prinzipien zum Entwurf einzelner Klassen
- Prinzipien zum Entwurf miteinander kooperierender Klassen

Robert C. Martin fasste eine wichtige Gruppe von Prinzipien zur Erzeugung wartbarer und erweiterbarer Software unter dem Begriff “SOLID” zusammen <sup>1</sup>. Robert C. Martin erklärte diese Prinzipien zu den wichtigsten Entwurfsprinzipien. Die SOLID-Prinzipien bestehen aus:

- **S**ingle Responsibility Prinzip
- **O**pen-Closed Prinzip
- **L**iskovsches Substitutionsprinzip
- **I**nterface Segregation Prinzip
- **D**ependency Inversion Prinzip

Die folgende Darstellung basiert auf den Referenzen <sup>2</sup>. Eine sehr gute, an einem Beispiel vorangetriebene Erläuterung ist unter <sup>3</sup> zu finden.

### Prinzip einer einzigen Verantwortung (Single-Responsibility-Prinzip SRP)

In der objektorientierten Programmierung sagt das SRP aus, dass jede Klasse nur eine fest definierte Aufgabe zu erfüllen hat. In einer Klasse sollten lediglich Funktionen vorhanden sein, die direkt zur Erfüllung dieser Aufgabe beitragen.

“There should never be more than one reason for a class to change.” <sup>4</sup>

- Verantwortlichkeit = Grund für eine Änderung (multiple Veränderungen == multiple Verantwortlichkeiten)

```
public class Employee
{
    public Money calculatePay() ...
    public void save() ...
    public String reportHours() ...
}
```

- Mehrere Verantwortlichkeiten innerhalb eines Software-Moduls führen zu zerbrechlichem Design, da Wechselwirkungen bei den Verantwortlichkeiten nicht ausgeschlossen werden können

```
public class SpaceStation{
    public initialize() ...
    public void run_sensors() ...
    public void show_sensors() ...
    public void load_supplies(type, quantity) ...
    public void use_supplies(type, quantity) ...
    public void report_supplies () ...
    public void load_fuel(quantity) ...
    public void report_fuel() ...
    public void activate_thrusters() ...
}
```

Eine mögliche separate Realisierung findet sich unter [Link](#)

**Merke:** Vermeiden Sie “God”-Objekte, die alles wissen.

### Verallgemeinerung

Eine Verallgemeinerung des SRP stellt Curly’s Law [CodingHorror](#) dar, welches das Konzept “methods should do one thing” bis “single source of truth” zusammenfasst und auf alle Aspekte eines Softwareentwurfs anwendet. Dazu gehören nicht nur Klassen, sondern unter anderem auch Funktionen und Variablen.

```
var numbers = new [] { 5,8,4,3,1 };
numbers = numbers.OrderBy(i => i);

var numbers = new [] { 5,8,4,3,1 };
var orderedNumbers = numbers.OrderBy(i => i);
```

Da die Variable `numbers` zuerst die unsortierten Zahlen repräsentiert und später die sortierten Zahlen, wird Curly’s Law verletzt. Dies lässt sich auflösen, indem eine zusätzliche Variable eingeführt wird.

<sup>1</sup>Robert C. Martin, Webseite “The principles of OOD”, <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

<sup>2</sup>Markus Just, IT Designers Gruppe, “Entwurfsprinzipien”, Foliensatz Fachhochschule Esslingen [Link](#)

<sup>3</sup>Andre Krämer, “SOLID - Die 5 Prinzipien für objektorientiertes Softwaredesign”, [Link](#)

<sup>4</sup>Robert C. Martin, Webseite “The principles of OOD”, <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

## Open-Closed Prinzip

Bertrand Meyer beschreibt das Open-Closed-Prinzip durch: *Module sollten sowohl offen (für Erweiterungen) als auch verschlossen (für Modifikationen) sein.*<sup>5</sup>

Eine Erweiterung im Sinne des Open-Closed-Prinzips ist beispielsweise die Vererbung. Diese verändert das vorhandene Verhalten der Einheit nicht, erweitert aber die Einheit um zusätzliche Funktionen oder Daten. Überschriebene Methoden verändern auch nicht das Verhalten der Basisklasse, sondern nur das der abgeleiteten Klasse.

Gegenbeispiel: Ausgangspunkt ist eine Klasse `Employee`, die für unterschiedliche Angestelltentypen um verschiedenen Algorithmen zur Bonusberechnung versehen werden soll. Intuitiv ist der Ansatz ein weiteres Feld einzufügen, dass den Typ des Angestellten erfasst und dazu eine entsprechende Verzweigung zu realisieren ... ein Verstoß gegen das OCP, der sich über eine Vererbungshierarchie deutlich wartungsfreundlicher realisieren lässt!

```
using System;

public class Employee
{
    public string Name {set; get;}
    public int ID {set; get;}

    public Employee(int id, string name){
        this.ID = id; this.Name = name;
    }

    public decimal CalculateBonus(decimal salary){
        return salary * 0.1M;
    }
}

public class Program
{
    public static void Main(string[] args){
        Employee Bernhard = new Employee(1, "Bernhard");
        Console.WriteLine($"Bonus = {Bernhard.CalculateBonus(11234)}");
    }
}
```

Achtung: Die Einbettung der `CalculateBonus()` Methode in die jeweiligen `Employee` Klassen ist selbst fragwürdig! Dadurch wird eine Funktion an verschiedenen Stellen realisiert, so dass pro Klasse unterschiedliche "Zwecke" umgesetzt werden. Damit liegt ein Verstoß gegen die Idee des SRP vor.

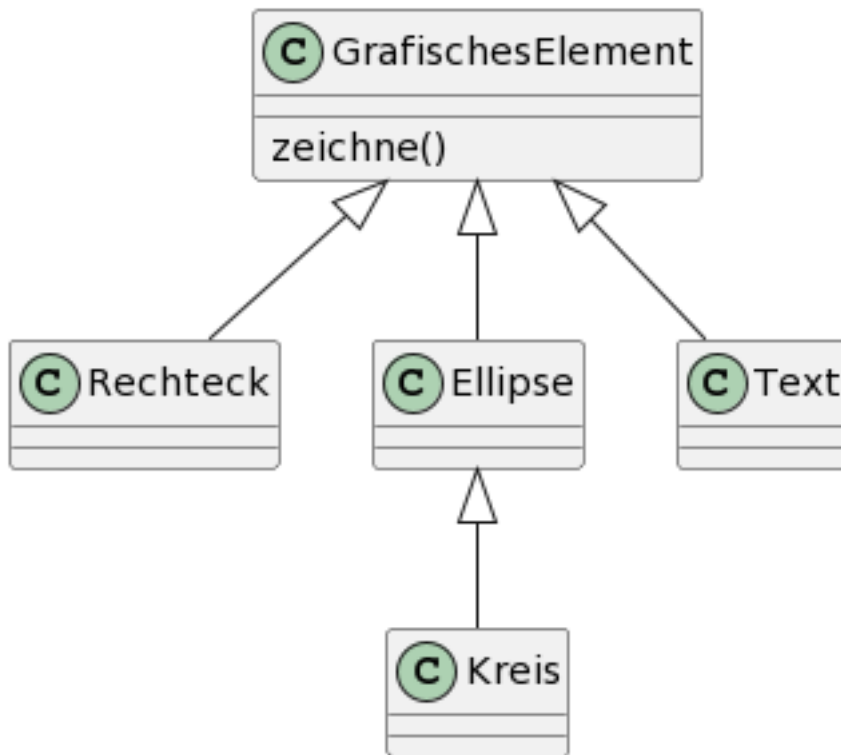
## Liskovsche Substitutionsprinzip (LSP)

Das Liskovsche Substitutionsprinzip (LSP) oder Ersetzbarkeitsprinzip besagt, dass ein Programm, das Objekte einer Basisklasse *T* verwendet, auch mit Objekten der davon abgeleiteten Klasse *S* korrekt funktionieren muss, ohne dabei das Programm zu verändern: *"Sei  $q(x)$  eine beweisbare Eigenschaft von Objekten  $x$  des Typs  $T$ . Dann soll  $q(y)$  für Objekte  $y$  des Typs  $S$  wahr sein, wobei  $S$  ein Untertyp von  $T$  ist."* [Liskov]

Beispiel: Grafische Darstellung von verschiedenen Primitiven

---

<sup>5</sup>Bertrand Meyer, "Object Oriented Software Construction" Prentice Hall, 1988,



Entsprechend sollte eine Methode, die **GrafischesElement** verarbeitet, auch auf **Ellipse** und **Kreis** anwendbar sein. Problematisch ist dabei allerdings deren unterschiedliches Verhalten. **Kreis** weist zwei gleich lange Halbachsen. Die zugehörigen Membervariablen sind nicht unabhängig von einander.

### Interface Segregation Prinzip

Zu große Schnittstellen sollten in mehrere Schnittstellen aufgeteilt werden, so dass die implementierende Klassen keine unnötigen Methoden umfasst. Schnittstellen aufgeteilt werden, falls implementierende Klassen unnötige Methoden haben müssen. Nach erfolgreicher Anwendung dieses Entwurfprinzips würde ein Modul, das eine Schnittstelle benutzt, nur die Methoden implementieren müssen, die es auch wirklich braucht.

```

public interface IVehicle
{
    void Drive();
    void Fly();
}

public class MultiFunctionalCar : IVehicle
{
    public void Drive()
    {
        //actions to start driving car
        Console.WriteLine("Drive a multifunctional car");
    }

    public void Fly()
    {
        //actions to start flying
        Console.WriteLine("Fly a multifunctional car");
    }
}

public class Car : IVehicle
{
    public void Drive()

```

```

{
    //actions to drive a car
    Console.WriteLine("Driving a car");
}

public void Fly()
{
    throw new NotImplementedException();
}
}

```

Lösung unter Beachtung des Interface Segregation Prinzip

```

public interface ICar
{
    void Drive();
}

public interface IAirplane
{
    void Fly();
}

public class Car : ICar
{
    public void Drive()
    {
        //actions to drive a car
        Console.WriteLine("Driving a car");
    }
}

public class MultiFunctionalCar : ICar, IAirplane
{
    public void Drive()
    {
        //actions to start driving car
        Console.WriteLine("Drive a multifunctional car");
    }

    public void Fly()
    {
        //actions to start flying
        Console.WriteLine("Fly a multifunctional car");
    }
}

```

Man könnte jetzt sogar ein Highlevel Interface realisieren, dass beide Aspekte integriert.

```

public interface IMultiFunctionalVehicle : ICar, IAirplane
{
}

public class MultiFunctionalCar : IMultiFunctionalVehicle
{
}

```

### Vorteil

- übersichtlichere kleinere Schnittstellen, die flexibler kombiniert werden können
- Klassen umfassen keine Methoden, die sie nicht benötigen

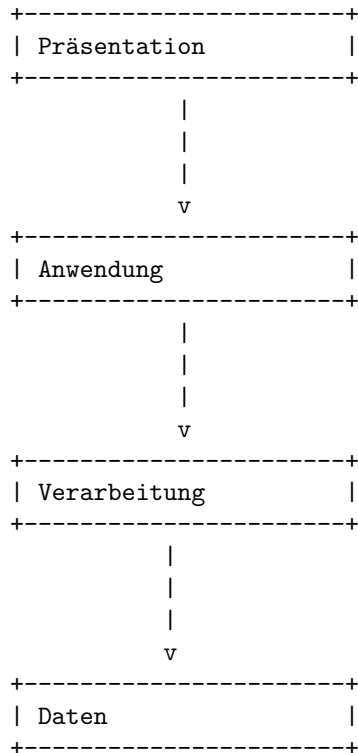
-> Das Prinzip der Schnittstellentrennung verbessert die Lesbarkeit und Wartbarkeit unseres Codes.

## Dependency Inversion Prinzip

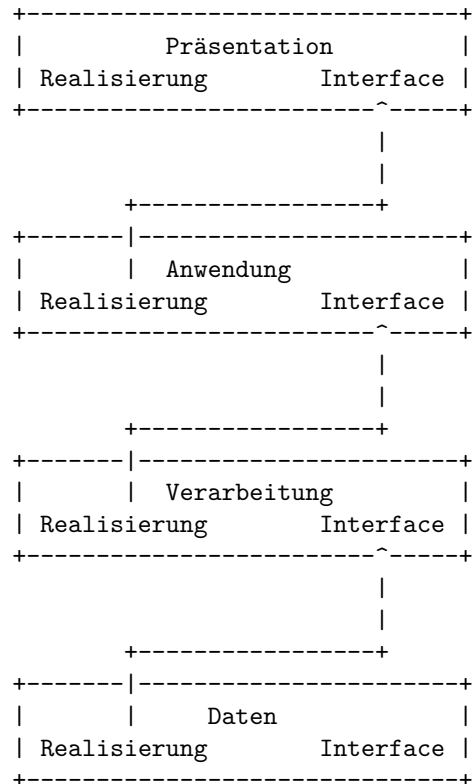
“High-level modules should not depend on low-level modules. Both should depend on abstractions.  
Abstractions should not depend upon details. Details should depend upon abstractions” [UncleBob]

Lösungsansatz für die Realisierung ist eine veränderte Sicht auf die klassischerweise hierarchische Struktur von Klassen.

Traditionelle Sicht



Objektorientierte Perspektive



Das folgende Beispiel entstammt der Webseite <https://exceptionnotfound.net/simply-solid-the-dependency-inversion-principle/>

Beachten Sie, dass die Benachrichtigungsklasse, eine übergeordnete Klasse, eine Abhängigkeit sowohl von der E-Mail-Klasse als auch von der SMS-Klasse hat, bei denen es sich um untergeordnete Klassen handelt. Mit anderen Worten, die Benachrichtigung hängt von der konkreten Implementierung von E-Mail und SMS ab und nicht von einer Abstraktion der Implementierung. Da DIP verlangt, dass sowohl Klassen der höheren als auch der unteren Ebenen von Abstraktionen abhängen, verstoßen wir derzeit gegen das Prinzip der Abhängigkeitsinversion.

```
public class Email
{
    public string ToAddress { get; set; }
    public string Subject { get; set; }
    public string Content { get; set; }
    public void SendEmail()
    {
        //Send email
    }
}

public class SMS
{
    public string PhoneNumber { get; set; }
    public string Message { get; set; }
    public void SendSMS()
    {
        //Send sms
    }
}
```

```

    }
}

public class Notification
{
    private Email _email;
    private SMS _sms;

    public Notification()
    {
        _email = new Email();
        _sms = new SMS();
    }

    public void Send()
    {
        _email.SendEmail();    // Abhängigkeit von Email
        _sms.SendSMS();        // Abhängigkeit von SMS
    }
}

```

Lösung

*// Schritt 1: Interface Definition*

```

public interface IMessage
{
    void SendMessage();
}

```

*// Schritt 2: Die niederwertigeren Klassen implmentieren das Interface*

```

public class Email : IMessage
{
    public string ToAddress { get; set; }
    public string Subject { get; set; }
    public string Content { get; set; }
    public void SendMessage()
    {
        //Send email
    }
}

```

```

public class SMS : IMessage
{
    public string PhoneNumber { get; set; }
    public string Message { get; set; }
    public void SendMessage()
    {
        //Send sms
    }
}

```

*// Schritt 3: Die höherwertige Klasse wird gegen das Interface implementiert*

```

public class Notification
{
    private IMessage _message;

    public Notification(IMessage messages)
    {
        this._message = message;
    }
    public void Send()
    {

```

```

        message.SendMessage();
    }
}

// Variante für multiple Messages
//public class Notification
//{
//    private ICollection<IMessage> _messages;
//    public Notification(ICollection<IMessage> messages)
//    {
//        this._messages = messages;
//    }
//    public void Send()
//    {
//        foreach(var message in _messages)
//        {
//            messages.SendMessage();
//        }
//    }
//}

```

Beispiel aus <https://exceptionnotfound.net/simply-solid-the-dependency-inversion-principle/>

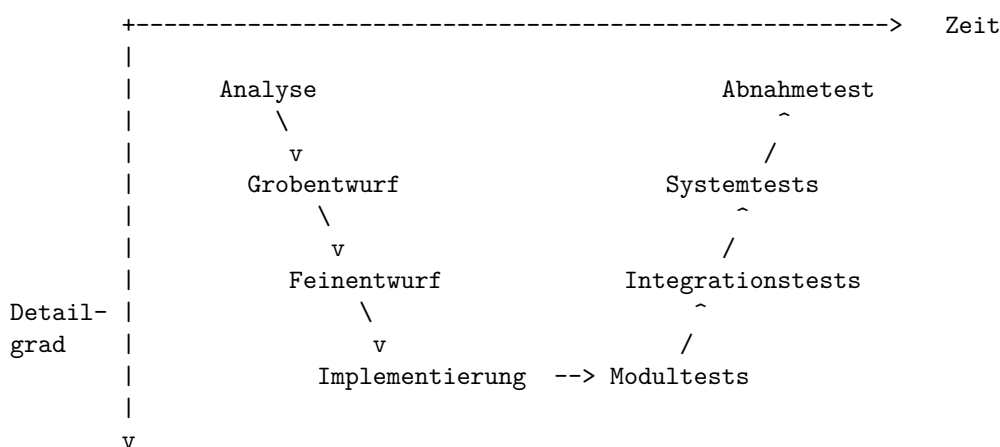
## Herausforderungen bei der Umsetzung der Prinzipien

Kunde TU Freiberg: *Entwickeln Sie für mich ein webbasiertes System, mit dem Sie die Anmeldung und Bewertung von Prüfungsleistung erfassen.*

Welche Fragen sollten Sie dem Kunden stellen, bevor Sie sich daran machen und munter Code schreiben?

Betrachten Sie die Darstellung auf der [Webseite](#). Welche hier scherzhaft beschriebenen Herausforderungen sehen Sie im Projekt?

Wie verzahnen wir den Entwicklungsprozess? Wie können wir sicherstellen, dass am Ende die erwartete Anwendung realisiert wird?



Das V-Modell ist ein Vorgehensmodell, das den Softwareentwicklungsprozess in Phasen organisiert. Zusätzlich zu den Entwicklungsphasen definiert das V-Modell auch das Evaluationsphasen, in welchen den einzelnen Entwicklungsphasen Testphasen gegenüber gestellt werden.

vgl. zum Beispiel [Link](#)

Achtung: Das V-Modell ist nur eine Variante eines Vorgehensmodells, moderne Entwicklungen stellen eher agile Methoden in den Vordergrund.

vgl. zum Beispiel [Link](#)



# Unified Modeling Language

Die Unified Modeling Language, kurz UML, dient zur Modellierung, Dokumentation, Spezifikation und Visualisierung komplexer Softwaresysteme unabhängig von deren Fach- und Realisierungsgebiet. Sie liefert die Notationselemente gleichermaßen für statische und dynamische Modelle zur Analyse, Design und Architektur und unterstützt insbesondere objektorientierte Vorgehensweisen. <sup>6</sup>

UML ist heute die dominierende Sprache für die Softwaresystem-Modellierung. Der erste Kontakt zu UML besteht häufig darin, dass Diagramme in UML im Rahmen von Softwareprojekten zu erstellen, zu verstehen oder zu beurteilen sind:

- Projektauftraggeber prüfen und bestätigen die Anforderungen an ein System, die Business Analysten in Anwendungsfalldiagrammen in UML festgehalten haben;
- Softwareentwickler realisieren Arbeitsabläufe, die Wirtschaftsanalytiker in Aktivitätsdiagrammen beschrieben haben;
- Systemingenieure implementieren, installieren und betreiben Softwaresysteme basierend auf einem Implementationsplan, der als Verteilungsdiagramm vorliegt.

UML ist dabei Bezeichner für die meisten bei einer Modellierung wichtigen Begriffe und legt mögliche Beziehungen zwischen diesen Begriffen fest. UML definiert weiter grafische Notationen für diese Begriffe und für Modelle statischer Strukturen und dynamischer Abläufe, die man mit diesen Begriffen formulieren kann.

**Merke:** Die grafische Notation ist jedoch nur ein Aspekt, der durch UML geregelt wird. UML legt in erster Linie fest, mit welchen Begriffen und welchen Beziehungen zwischen diesen Begriffen sogenannte Modelle spezifiziert werden.

Was ist UML nicht:

- vollständige, eindeutige Abbildung aller Anwendungsfälle
- keine Programmiersprache
- keine rein formale Sprache
- kein vollständiger Ersatz für textuelle Beschreibungen
- keine Methode oder Vorgehensmodell

## Geschichte

UML (aktuell UML 2.5) ist durch die Object Management Group (OMG) als auch die ISO (ISO/IEC 19505 für Version 2.4.1) genormt.

## UML Werkzeuge

- Tools zur Modellierung - Unterstützung des Erstellungsprozesses, Überwachung der Konformität zur graphischen Notation der UML

*Herausforderungen:* Transformation und Datenaustausch zwischen unterschiedlichen Tools

- Quellcoderzeugung - Generierung von Sourcecode aus den Modellen

*Herausforderungen:* Synchronisation der beiden Repräsentationen, Abbildung widersprüchlicher Aussagen aus verschiedenen Diagrammtypen

(Beispiel mit Visual Studio folgt am Ende der Vorlesung.)

- Reverse Engineering / Dokumentation - UML-Werkzeug bilden Quelltext als Eingabe liest auf entsprechende UML-Diagramme und Modelldaten ab

*Herausforderungen:* Abstraktionskonzept der Modelle führt zu verallgemeinernden Darstellungen, die ggf. Konzepte des Codes nicht reflektieren.

## Darstellung von UML im Rahmen dieser Vorlesung

Die Vorlesungsunterlagen der Veranstaltung “Softwareentwicklung” setzen auf die domainspezifische Beschreibungssprache plantUML auf, die verschiedene Aspekte in einer

<http://plantuml.com/de/>

```
@startuml
class Car
```

---

<sup>6</sup>Mario Jeckle, Christine Rupp, Jürgen Hahn, Barbara Zengler, Stefan Queins, UML 2 glasklar, Hanser Verlag, 2004

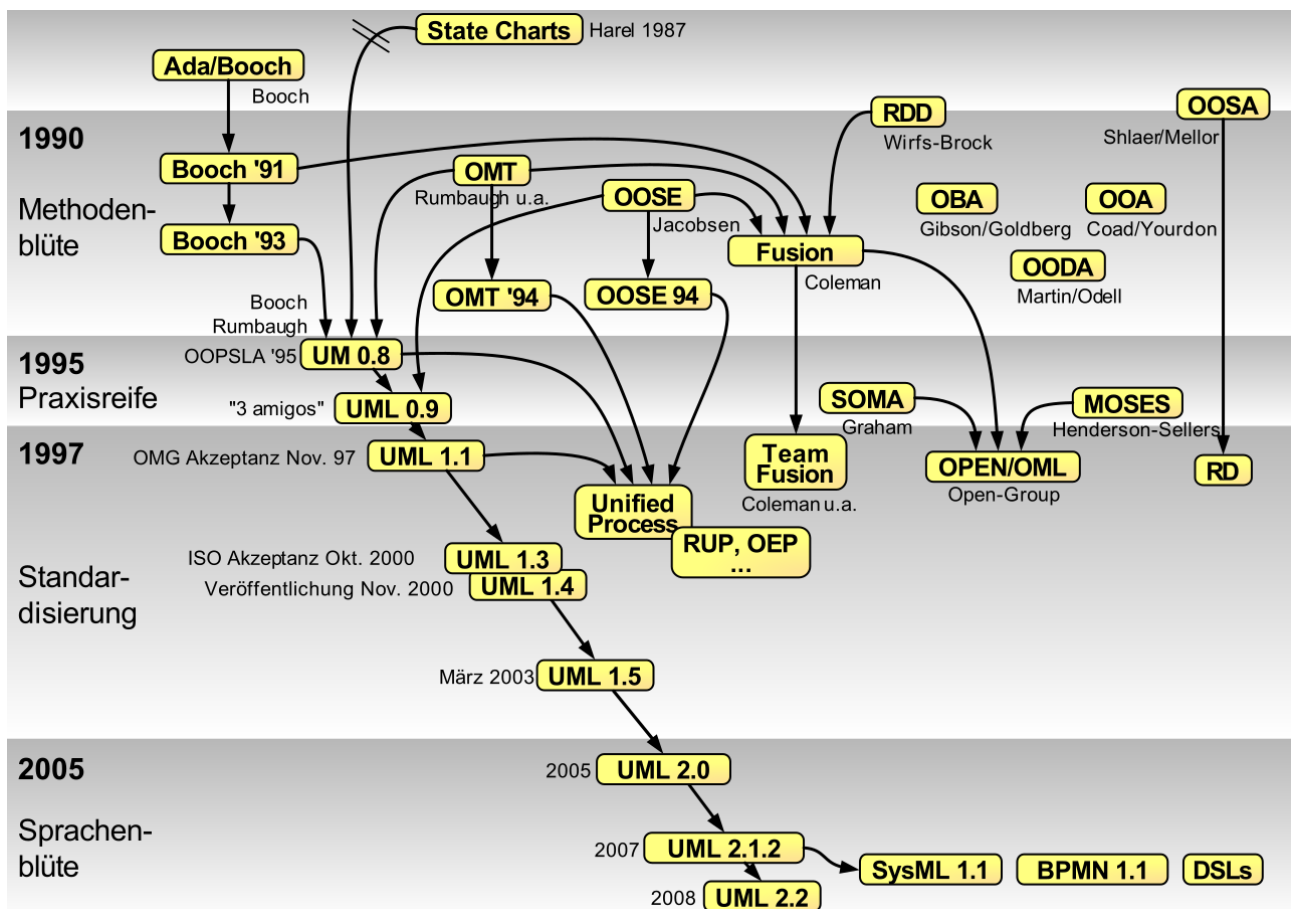


Figure 1: OOPGeschichte

Main Page

Namespaces

Classes

Files

Class List

Class Index

Class Hierarchy

Class Members

libsocket

selectset

Public Member Functions

Private Attributes

List of all members

libsocket::selectset Class Reference

Collaboration diagram for libsocket::selectset:

```

graph TD
    int -- sfd --> libsocket_socket[libsocket::socket]
    libsocket_socket -- elements --> int
    libsocket_socket -- keys --> std_map_int_socket_ptr[std::map<int, socket*>]
    std_map_int_socket_ptr -- fdsockmap --> libsocket_selectset[libsocket::selectset]
    libsocket_selectset -- elements --> libsocket_socket
    libsocket_selectset -- filedescriptors --> std_vector_int[std::vector<int>]
    std_vector_int -- elements --> libsocket_selectset
    libsocket_selectset -- set_up --> bool
    bool -- readset --> fd_set
    fd_set -- writeset --> bool

```

[legend]

Public Member Functions

void **add\_fd** (**socket** &sock, int method)  
std::pair< std::vector< **socket** \* > , std::vector< **socket** \* > > **wait** (long long microseconds=0)

Private Attributes

std::vector< int > **filedescriptors**  
std::map< int, **socket** \* > **fdsockmap**  
bool **set\_up**  
fd\_set **readset**  
fd\_set **writeset**

The documentation for this class was generated from the following files:

Figure 2: OOPGeschichte

```

Driver - Car : drives >
Car *- Wheel : have 4 >
Car -- Person : < owns

@enduml

@plantUML.eval(png)

@startuml
robust "Web Browser" as WB
concise "Web User" as WU

@0
WU is Idle
WB is Idle

@100
WU is Waiting
WB is Processing

@300
WB is Waiting
@enduml

@plantUML.eval(png)

```

## Diagramm-Typen

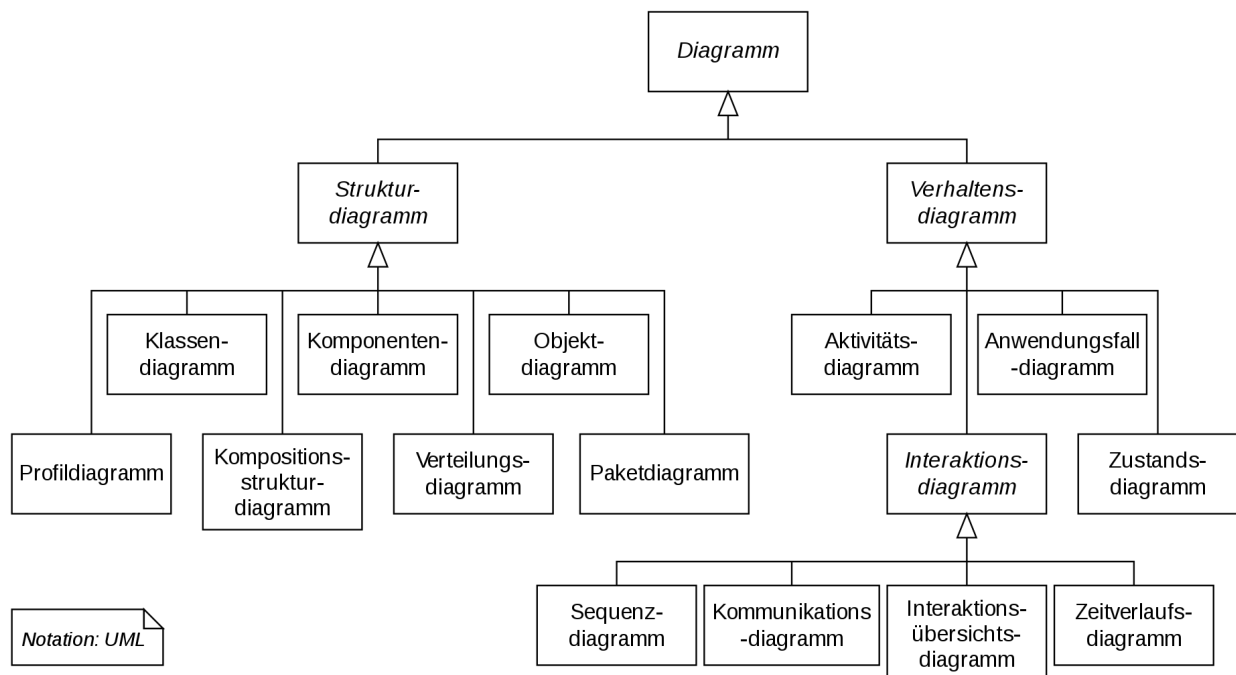


Figure 3: OOPGeschichte

## Strukturdiagramme

Diagrammtyp	Zentrale Frage
Klassendiagramm	Welche Klassen bilden das Systemverhalten ab und in welcher Beziehung stehen diese?
Paketdiagramm	Wie kann ich mein Modell in Module strukturieren?
Objektdiagramm	Welche Instanzen bestehen zu einem bestimmten Zeitpunkt im System?

Diagrammtyp	Zentrale Frage
Kompositionsstrukturdiagramm	Welche Elemente sind Bestandteile einer Klasse, Komponente, eines Subsystems?
Komponentendiagramm	Wie lassen sich die Klassen zu wiederverwendbaren Komponenten zusammenfassen und wie werden deren Beziehungen definiert?
Verteilungsdiagramm	Wie sieht das Einsatzumfeld des Systems aus?

## Verhaltensdiagramme

Diagrammtyp	Zentrale Frage
Use-Case-Diagramm	Was leistet mein System überhaupt? Welche Anwendungen müssen abgedeckt werden?
Aktivitätsdiagramm	Wie lassen sich die Stufen eines Prozesses beschreiben?
Zustandsautomat	Welche Abfolge von Zuständen wird für eine eine Sequenz von Eingangsinformationen realisiert
Sequenzdiagramm	Wer tauscht mit wem welche Informationen aus? Wie bedingen sich lokale Abläufe untereinander?
Kommunikationsdiagramm	Wer tauscht mit wem welche Informationen aus?
Timing-Diagramm	Wie hängen die Zustände verschiedener Akteure zeitlich voneinander ab?
Interaktionsübersichtsdiagramm	Wann läuft welche Interaktion ab?

## Begrifflichkeiten

Ein UML-Modell ergibt sich aus der Menge aller seiner Diagramme. Entsprechend werden verschiedene Diagrammtypen genutzt um unterschiedliche Perspektiven auf ein realweltliches Problem zu entwickeln.

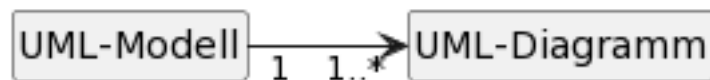


Figure 4: Modelle

## Aufgaben

- ☐ Bearbeiten Sie die Aufgabe 3 im GitHub Classroom
- ☐ Machen Sie sich mit den SOLID Prinzipien weiter vertraut:

!/?alt-text