Modellierung von Software

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Teil:	`13/27`
Semester	`Sommersemester 2025`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	Prinzipien des (objektorientierten) Softwareentwurfs, Motivation der Modellierung von Software, Unified Modeling Language
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/15_UML_Modellierung.md
Autoren	Sebastian Zug, Galina Rudolf & André Dietrich

Organisatorisches

Projektarbeiten als Leistungsnachweis für die Vorlesung Softwareentwicklung.

https://github.com/ComputerScienceLecturesTUBAF/SoftwareentwicklungSoSe2025_Projektaufgaben

Motivation des Modellierungsgedankens

ugly_version.py

```python
class Report:
    def __init__(self, title, content):
        self.title = title
        self.content = content

    def generate_report(self):
        return f"Report: {self.title}\n{self.content}"

    def save_to_file(self, filename):
        with open(filename, 'w') as f:
            f.write(self.generate_report())

    def send_via_email(self, email_address):
        print(f"Sending report to {email_address}...")
        # (Hier würde tatsächliche E-Mail-Logik stehen)


# Verwendung
report = Report("Monatsbericht", "Umsatz ist gestiegen.")
report.save_to_file("bericht.txt")
report.send_via_email("chef@firma.de")
```

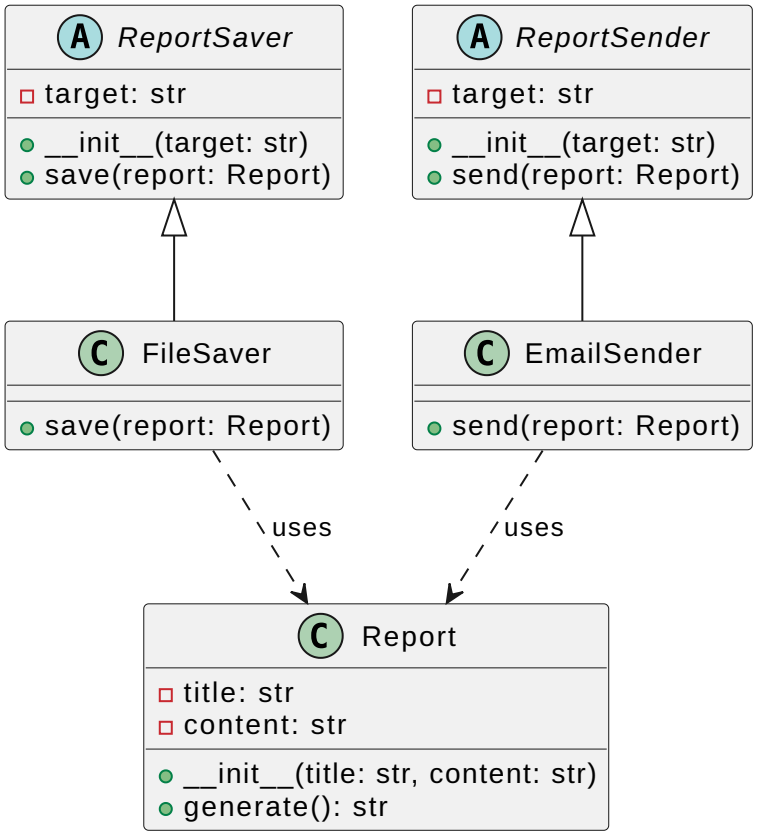
Was gefällt Ihnen nicht an diesem Code?

improved_version.py

```python
from abc import ABC, abstractmethod

# ----- Domänenmodell -----
class Report:
    def __init__(self, title, content):
        self.title = title
        self.content = content

    def generate(self):
        return f"Report: {self.title}\n{self.content}"


# ----- Abstrakte Klassen -----
class ReportSaver(ABC):
    def __init__(self, target: str):
        self.target = target

    @abstractmethod
    def save(self, report: Report):
        pass


class ReportSender(ABC):
    def __init__(self, target: str):
        self.target = target

    @abstractmethod
    def send(self, report: Report):
        pass


# ----- Implementierungen -----
class FileSaver(ReportSaver):
    def save(self, report: Report):
        with open(self.target, 'w') as f:
            f.write(report.generate())
        print(f"Report saved to {self.target}")


class EmailSender(ReportSender):
    def send(self, report: Report):
        print(f"Sending report to {self.target}...")
        print(report.generate())


if __name__ == "__main__":
    report = Report("Monatsbericht", "Umsatz ist gestiegen.")
```

```
saver = FileSaver("bericht.txt")
saver.save(report)

sender = EmailSender("chef@firma.de")
sender.send(report)
```

Was stört Sie, wenn wir uns einen so langen Code anschauen? Die übersichtlichkeit leidet!



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!

Gibt es für ein Problem mehrere Lösungen? Sind sie alle gleich gut? Ist das einfachste und nahliegende nach KISS-Prinzip (Keep It Simple, Stupid) immer am besten?

- Prinzipien zum Entwurf von Systemen: Modularität, Trennung von Zuständigkeiten (Separation of Concerns), Schichtenarchitektur, lose Kopplung und hohe Kohäsion von Modulen
- Prinzipien zum Entwurf einzelner Klassen: Single Responsibility (einzige Verantwortlichkeit) Principle, Kapselung, Immutable Objects
- Prinzipien zum Entwurf miteinander kooperierender Klassen: O, L, I, D, Law of Demeter (Kommunikation nur unter "verwandten" Klassen, keine langen Aufrufketten)

Robert C. Martin fasste eine wichtige Gruppe von Prinzipien zur Erzeugung wartbarer und erweiterbarer Software unter dem Begriff "SOLID" zusammen [UncleBob]. 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 [Just]. Eine sehr gute, an einem Beispiel vorangetrieben Erläuterung ist unter [Krämer] zu finden.

[Just] Markus Just, IT Designers Gruppe, "Entwurfsprinzipien", Foliensatz Fachhochschule Esslingen Link

[Krämer] Andre Krämer, "SOLID - Die 5 Prinzipien für objektorientiertes Softwaredesign", Link

[UncleBob] Robert C. Martin, Webseite "The principles of OOD", http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

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. [UncleBob]

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

```
public class Book
{
    // nur eine Verantwortlichkeit: die Verwaltung der Buchdetails
    public string Title { get; private set; }
    public string Author { get; private set; }
    public string ISBN { get; private set; }

    public Book(string title, string author, string isbn)
    {
        Title = title;
        Author = author;
        ISBN = isbn;
    }

    public string GetBookDetails()
    {
        return $"Title: {Title}, Author: {Author}, ISBN: {ISBN}";
    }
    //Speichern von Büchern in einer Datenbank, das Drucken der Buchdetails
      erfolgt in den separaten Klassen
}
```

- Mehrere Verantwortlichkeiten innerhalb eines Software-Moduls führen zu zerbrechlichem Design, da Wechselwirkungen bei den Verantwortlichkeit 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 separaten 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.

> Jedes Wissenselement muss eine einzigartige, eindeutige Darstellung innerhalb eines Systems haben.

```
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.

Open-Closed Prinzip

Bertrand Meyer beschreibt das Open-Closed-Prinzip durch: *Module sollten sowohl offen (für Erweiterungen) als auch verschlossen (für Modifikationen) sein.* [Meyer]

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.

PriceCalculator.cs

```
using System;
using System.Collections.Generic;

// Basisklasse mit abstrakter Methode
public abstract class Product
{
    public string Name { get; set; }
    public abstract decimal GetPrice();
}

// Erweiterung 1: Einfaches Produkt
public class StandardProduct : Product
{
    public decimal BasePrice { get; set; }

    public override decimal GetPrice()
    {
        return BasePrice;
    }
}

// Erweiterung 2: Produkt mit Rabatt
public class DiscountedProduct : Product
{
    public decimal BasePrice { get; set; }
    public decimal DiscountPercent { get; set; }

    public override decimal GetPrice()
    {
        return BasePrice * (1 - DiscountPercent / 100m);
    }
}

// Erweiterung 3: Premium-Produkt mit Aufschlag
public class PremiumProduct : Product
{
    public decimal BasePrice { get; set; }
    public decimal PremiumFee { get; set; }

    public override decimal GetPrice()
    {
        return BasePrice + PremiumFee;
    }
}

// Verwenderklasse
public class PriceCalculator
{
```

```
    public decimal CalculateTotalPrice(List<Product> products)
    {
        decimal total = 0;
        foreach (var product in products)
        {
            total += product.GetPrice();
        }
        return total;
    }
}

// Testprogramm
class Program
{
    static void Main()
    {
        var products = new List<Product>
        {
            new StandardProduct { Name = "Buch", BasePrice = 20m },
            new DiscountedProduct { Name = "Stift", BasePrice = 5m,
              DiscountPercent = 10 },
            new PremiumProduct { Name = "Laptop", BasePrice = 1000m,
              PremiumFee = 150m }
        };

        var calculator = new PriceCalculator();
        var total = calculator.CalculateTotalPrice(products);

        Console.WriteLine($"Gesamtpreis: {total} EUR");
    }
}
```

```
Gesamtpreis: 1174.5 EUR
Gesamtpreis: 1174.5 EUR
```

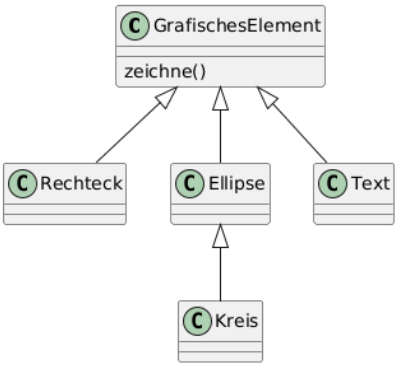
Die Klasse PriceCalculator muss nicht geändert werden, wenn ein neuer Produkttyp hinzukommt. Stattdessen kann man durch Vererbung und Polymorphie neue Klassen hinzufügen (GiftProduct, Leihprodukt, etc.). Die Erweiterung erfolgt über neue Klassen, nicht durch Änderung des bestehenden Codes.

[Meyer] Bertrand Meyer, "Object Oriented Software Construction" Prentice Hall, 1988,

Liskovsche Substitutionsprinzip (LSP)

> *"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] 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.

Beispiel: Grafische Darstellung von verschiedenen Primitiven



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 auf. Die zugehörigen Membervariablen sind nicht unabhängig voneinander.

[Liskov] Liskov, Barbara H., and Jeannette M. Wing. "A Behavioral Notion of Subtyping." ACM Transactions on Programming Languages and Systems, vol. 16, no. 6, 1994, pp. 1811–41. doi:10.1145/197320.197383

Interface Segregation Prinzip

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

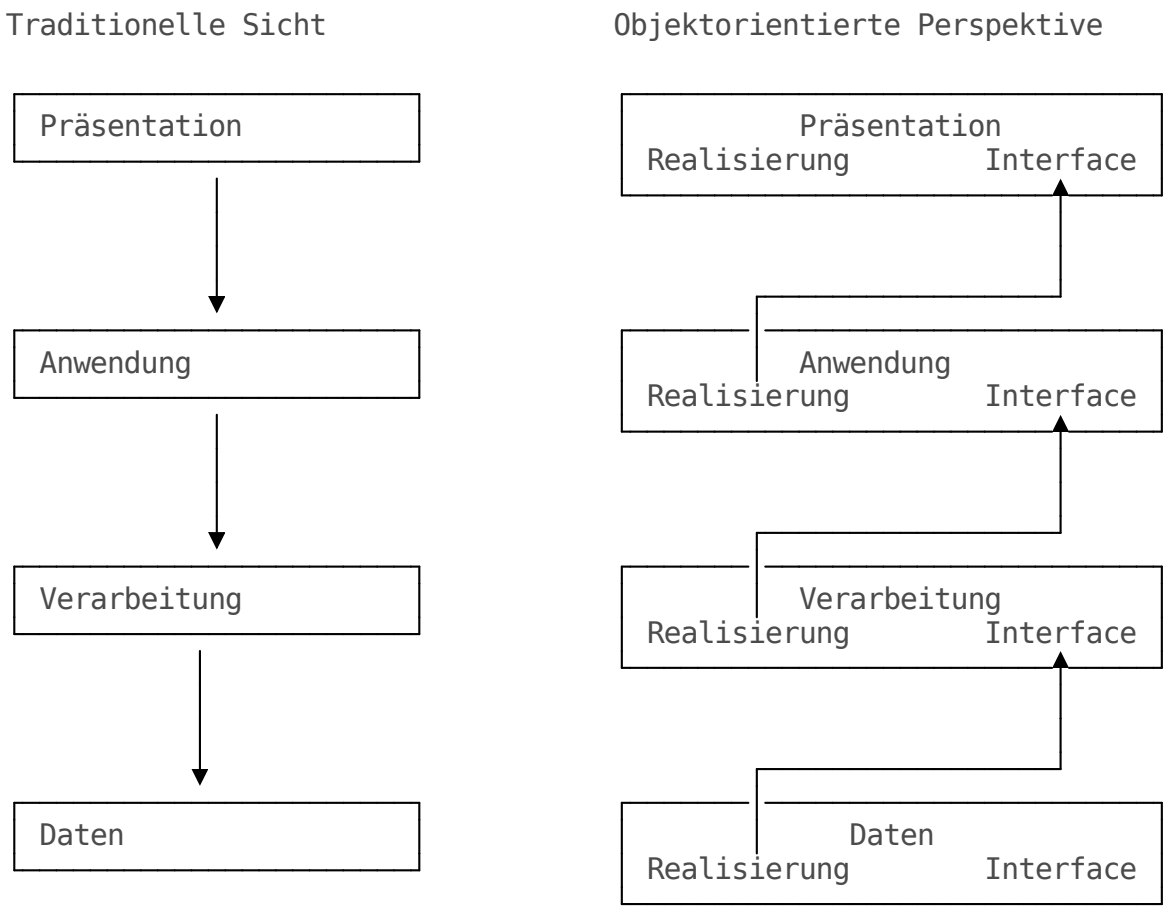
- übersichtlichere kleinere Schnittstellen, die flexibler kombiniert werden können
- Klassen umfassen keine Methoden, die sie nicht benötigen

→ 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 hierachische Struktur von Klassen.



Hohe Abstraktionsebenen (High-level modules) sind die Komponenten des Systems, die höhere Logik oder Funktionalität implementieren (z.B. Benutzeroberfläche).

Niedrige Abstraktionsebenen (Low-level modules) sind Komponenten, die konkrete Aufgaben wie Datenzugriff, Netzwerkkommunikation oder hardwarebezogene Operationen ausführen.

> Die High-level Module sollen nicht direkt von den Low-level Modulen abhängig sein. Stattdessen sollten beide Arten von Modulen von Abstraktionen abhängen, wie Schnittstellen oder abstrakte Klassen.

Beispiel: High-level Modul ist ein Rechnungsservice, und ein Low-level Modul ist ein konkretes Datenrepository. Rechnungsservice soll nicht direkt vom Datenrepository abhängen, sondern von einer Abstraktion (IDataRepository). Das konkrete Datenrepository implementiert dann diese Schnittstelle.

```
public interface IDataRepository
{
    void SaveInvoice(Invoice invoice);
}

public class DatabaseRepository : IDataRepository //Low-level Modul
{
    public void SaveInvoice(Invoice invoice)
```

```
    public void SaveInvoice(Invoice invoice)
    {
        // Konkrete Implementierung zum Speichern der Rechnung in der Daten
    }
}

public class InvoiceService //High-level Modul
{
    private IDataRepository repository;

    public InvoiceService(IDataRepository repository)
    {
        this.repository = repository;
    }

    public void CreateInvoice(Invoice invoice)
    {
        // Geschäftslogik zur Erstellung einer Rechnung
        repository.SaveInvoice(invoice);
    }
}

class Program
{
    static void Main()
    {
        IDataRepository repository = new DatabaseRepository();
        InvoiceService invoiceService = new InvoiceService(repository);

        Invoice invoice = new Invoice();
        invoiceService.CreateInvoice(invoice);
    }
}
```

Abstraktionen (wie Schnittstellen oder abstrakte Klassen) sollen nicht von konkreten Implementierungen abhängig sein, sondern nur von (anderen) Abstraktionen.

```
public interface INotificationSender
{
    void Send(string message);
}

//EmailSender, SmsSender

public interface INotificationService //abhängige Abstraktion
{
    void Notify(string message, INotificationSender sender);
}
```

```
public class NotificationService : INotificationService
{
    public void Notify(string message, INotificationSender sender)
    {
        // Geschäftslogik zur Benachrichtigung
        sender.Send(message);
    }
}

class Program
{
    static void Main()
    {
        INotificationSender emailSender = new EmailSender();
        INotificationSender smsSender = new SmsSender();

        INotificationService notificationService = new NotificationService(

        notificationService.Notify("Notification via Email.", emailSender);
        notificationService.Notify("Notification via SMS.", smsSender);
    }
}
```

```
from abc import ABC, abstractmethod

# Definition des Interface INotificationSender
class INotificationSender(ABC):
    @abstractmethod
    def send(self, message: str):
        pass

# Definition des Interface INotificationService
class INotificationService(ABC):
    @abstractmethod
    def notify(self, message: str, sender: INotificationSender):
        pass

# Implementierung des EmailSender
class EmailSender(INotificationSender):
    def send(self, message: str):
        print(f"Sending email: {message}")

# Implementierung des SmsSender
class SmsSender(INotificationSender):
    def send(self, message: str):
        print(f"Sending SMS: {message}")

# Implementierung des NotificationService
```

```
# Implementierung des NotificationService
class NotificationService(INotificationService):
    def notify(self, message: str, sender: INotificationSender):
        # Geschäftslogik zur Benachrichtigung
        sender.send(message)

# Hauptprogramm
if __name__ == "__main__":
    email_sender = EmailSender()
    sms_sender = SmsSender()
    notification_service = NotificationService()
    notification_service.notify("Notification via Email.", email_sender)
    notification_service.notify("Notification via SMS.", sms_sender)
```

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
    }
}
```

Warum ist dieser Code nicht DIP konform?

```
// 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 implementier
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)
//        {
//            message.SendMessage();
//        }
//    }
//}
```

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

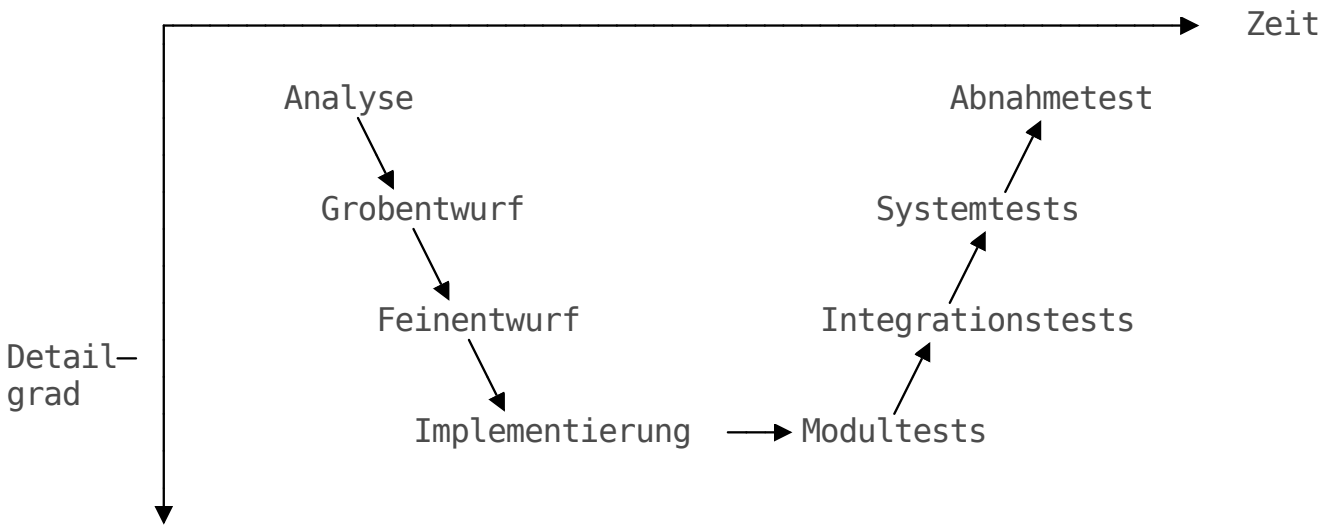
Herausforderungen bei der Umsetzung der Prinzipien

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 die Evaluationsphasen, in welchen den einzelnen Entwicklungsphasen Testphasen gegenübergestellt 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

Objektorientierte Analyse, objektorientiertes Design

Die **objektorientierte Analyse (OO-Analyse)** ist der Prozess der Analyse von Anforderungen aus der Perspektive von Objekten und deren Interaktionen. Die Anforderungen werden in verschiedene Klassen (Objekte) zerlegt, die Daten und Verhalten gemeinsam haben, typische Benutzungsabläufe (Use Cases) werden dokumentiert, um das Verhalten des Systems aus Sicht der Benutzer darzustellen. Ziel ist es, ein **Modell** zu erstellen, das das System und seine Eigenschaften klar darstellt.

Das **objektorientierte Design (OO-Design)** setzt das Modell aus der Analyse in eine detaillierte Softwarearchitektur um. Dabei werden die verschiedenen Klassen, ihre Methoden und Interaktionen spezifiziert.

Als Standardnotation für OOA/OOD wird UML (Unified Modeling Language) verwendet.

Unified Modeling Language

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

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 enthält dabei Bezeichner (Begriffe) für die meisten Elemente der Modellierung und legt mögliche Beziehungen zwischen diesen Elementen fest.

UML definiert weiterhin grafische Notationen für diese Begriffe und für **statische Strukturen** und **dynamische 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.

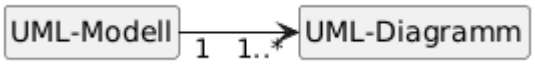
Was ist UML nicht:

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

UML-Modell und Diagramme

UML-Modell: ist eine abstrakte Darstellung eines Systems, das alle relevanten Informationen über die Struktur und das Verhalten des Systems enthält. Es umfasst nicht nur Diagramme, sondern auch die (nicht darstellbaren) Beziehungen, Constraints und anderen Metadaten, die die Modellierung ausmachen.

UML-Diagramme: sind verschiedene grafische Darstellungen, die unterschiedliche Aspekte des Systems betonen. Es gibt mehrere Arten von UML-Diagrammen, die um unterschiedliche Perspektiven auf ein realweltliches Problem zeigen. Ein UML-Modell beinhaltet die Menge aller seiner Diagramme.

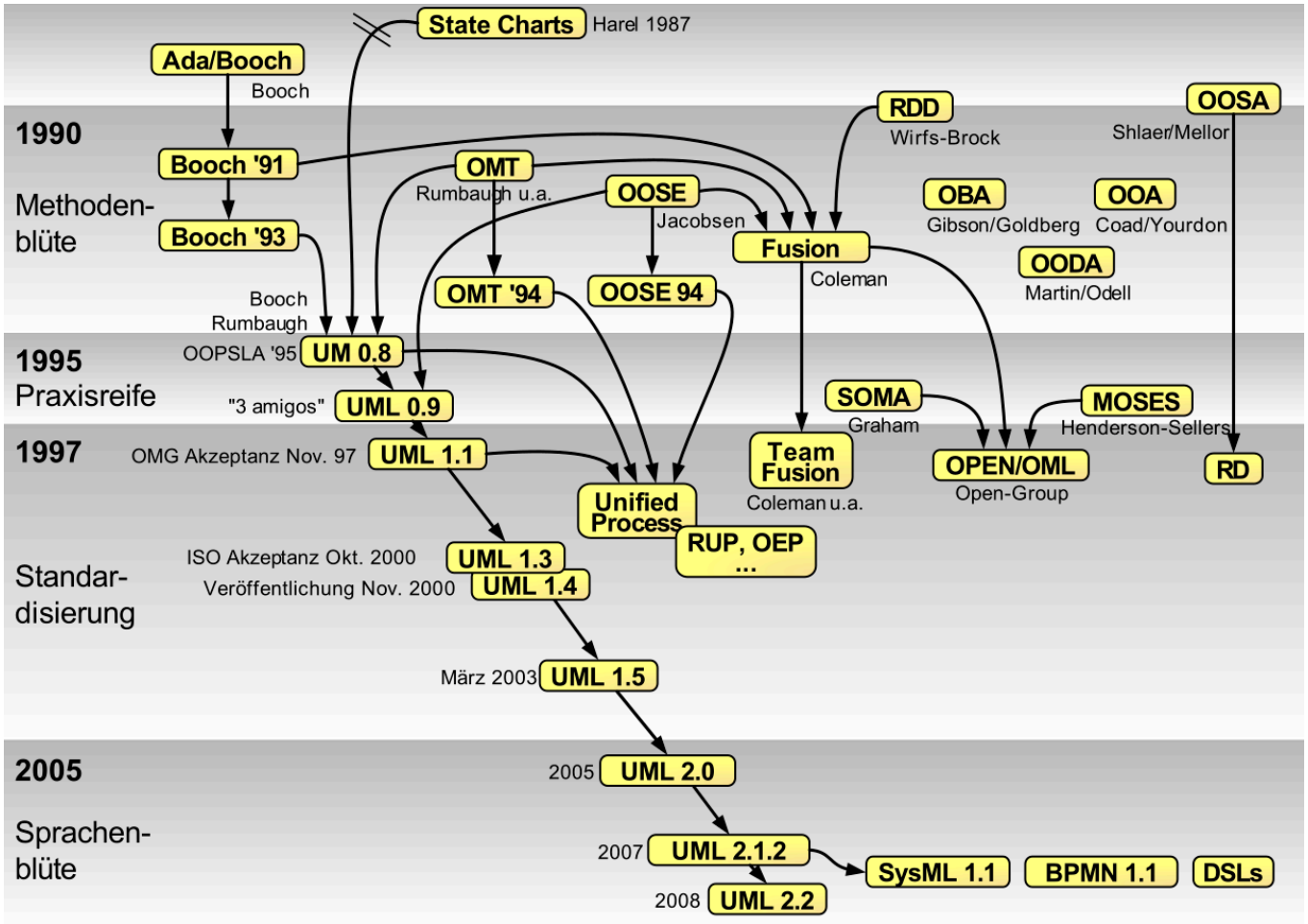


Modell vs. Diagramm

[Jeckle] Mario Jeckle, Christine Rupp, Jürgen Hahn, Barbara Zengler, Stefan Queins, UML 2 glasklar, Hanser Verlag, 2004

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.



Darstellung der Historie von UML [WikiUMLHist]

Booch-Methode (*Grady Booch*, 1984) umfasste die ersten formalen Methoden für das objektorientierte Software-Engineering, Konzepte wie Klassen, Vererbung, Assoziationen und Aggregationen mit graphischen Elementen.

OMT - Object Modeling Technique (*James Rumbaugh* et al., Ende der 1980er) war eine Methode für das objektorientierte Modellieren mit der grafischen Notation für die Analyse und das Design von Systemen.

OOSE - Object-Oriented Software Engineering (*Ivar Jacobson*, 1992) betonte die Verwendung von Anwendungsfällen (Use Cases) zur Spezifikation von Systemanforderungen und war eine der ersten Methoden, die diese Konzept einführte.

UML (3 amigos, Rational Software, Mitte 1990er) vereinheitlichtes Modellierungssystem, das die verschiedenen Ansätze und Diagrammtypen der Objektorientierten Analyse und Design (OOA/OOD) vereint.

UML 1 (1997) umfasst eine Reihe von Diagrammen, etabliert sich als Standard für die Modellierung von Software- und Systemarchitekturen.

Übernahme durch die OMG - Object Management Group (1997) leitete den Beginn als offenen Industriestandard ein.

UML 2 (2005) eine aktualisierte und erweiterte Version der UML mit weiteren Diagrammtypen und Verbesserungen in der Semantik und der Modellierungssprache.

[WikiUMLHist] https://commons.wikimedia.org/w/index.php?curid=7892951, Autor GuidoZockoll, Mitarbeiter der oose.de Dienstleistungen für Innovative Informatik GmbH - derivative work: File:OO-historie.svg : AxelScheithauer, oose.de Dienstleistungen für Innovative Informatik GmbH - derivative work: Chris828 (talk) - File:Objektorientieren methoden historie.png and File:OO-historie.svg, CC BY-SA 3.0

UML Werkzeuge

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

  Herausforderungen: Transformation und Datenaustausch zwischen unterschiedlichen Tools

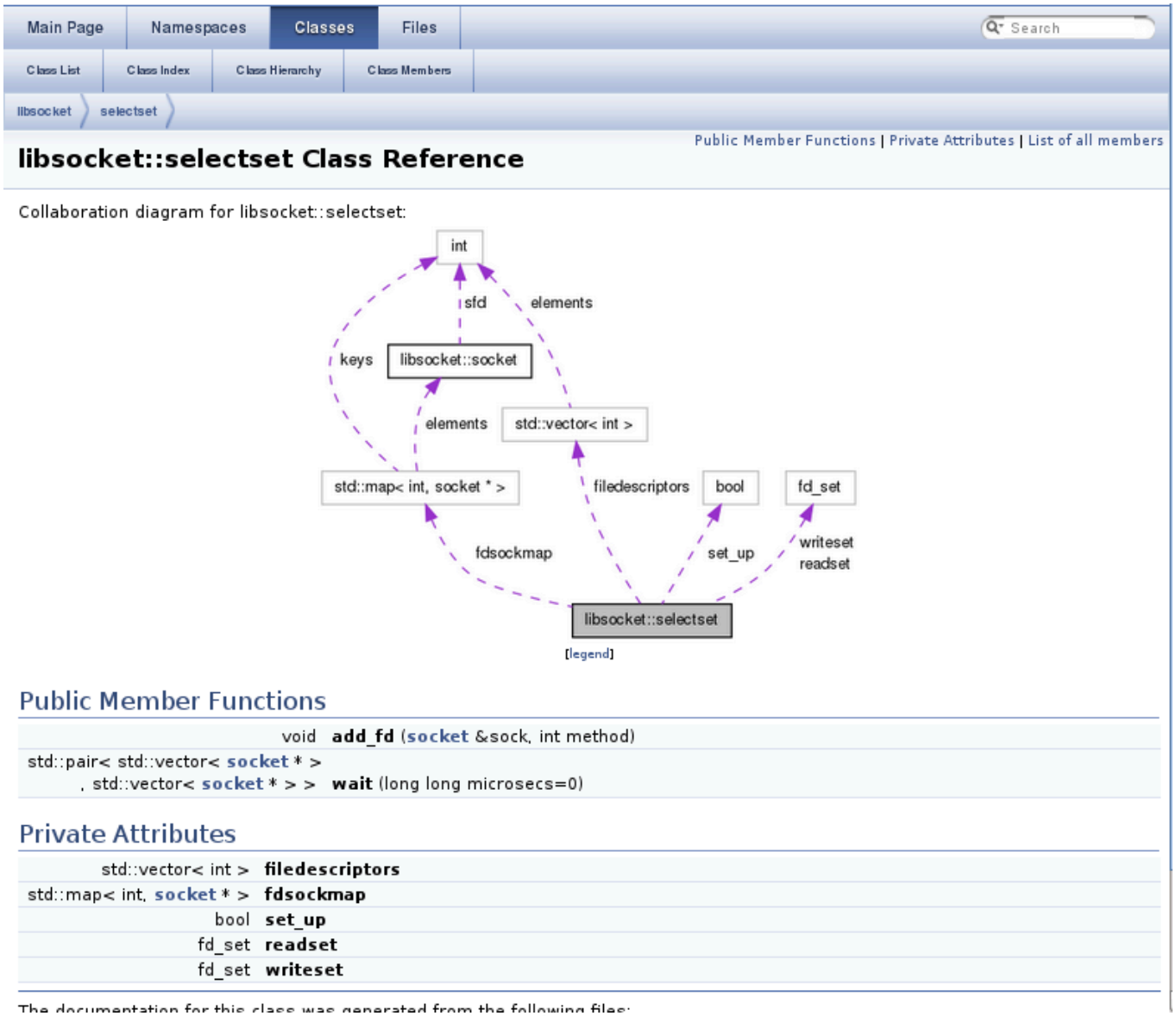
- Quellcoderzeugung - Generierung von Sourcecode aus den Modellen

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

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

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

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



Nutzung einer adaptierten Variante in Doxygen [WikiDoxygen]

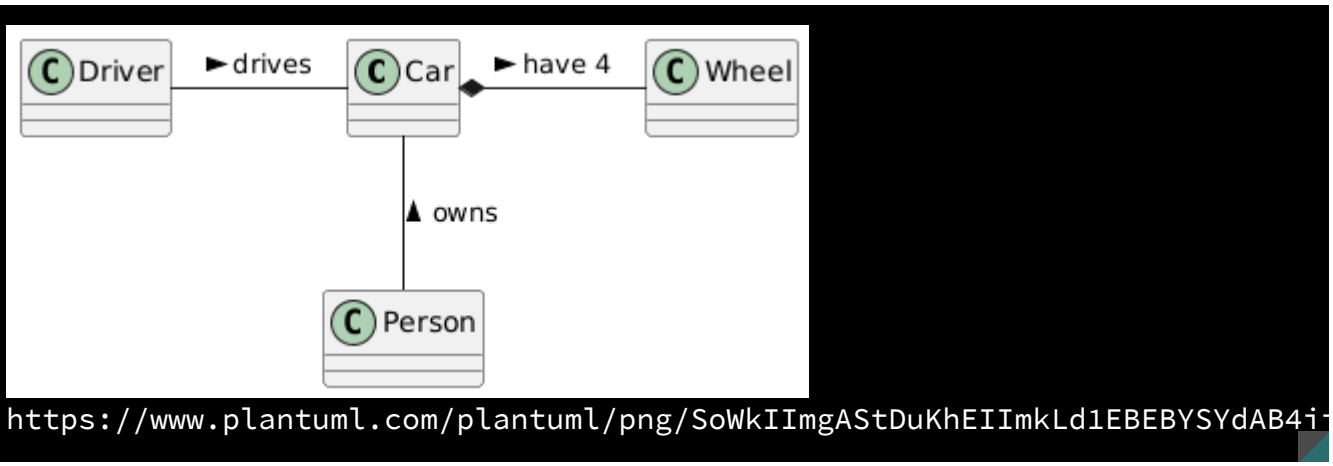
Darstellung von UML im Rahmen dieser Vorlesung

Die Vorlesungsunterlagen der Veranstaltung "Softwareentwicklung" setzen auf die domainspezifische Beschreibungssprache plantUML auf, die verschiedene Aspekte in einer einheitlichen und übersichtlichen Weise darstellt.

http://plantuml.com/de/

ClassDiagram

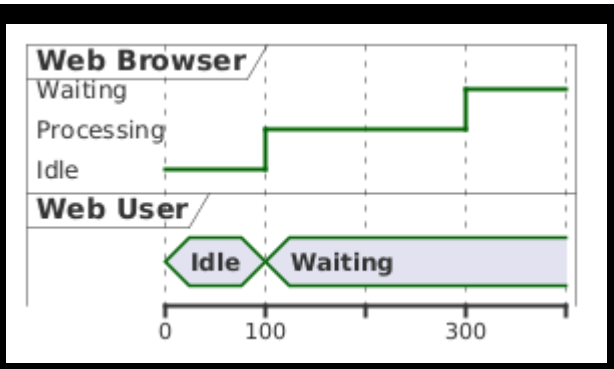
```
@startuml
class Car

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

@enduml
```



https://www.plantuml.com/plantuml/png/SoWkIImgAStDuKhEIImkLd1EBEBYSYdAB4i

GanttChart

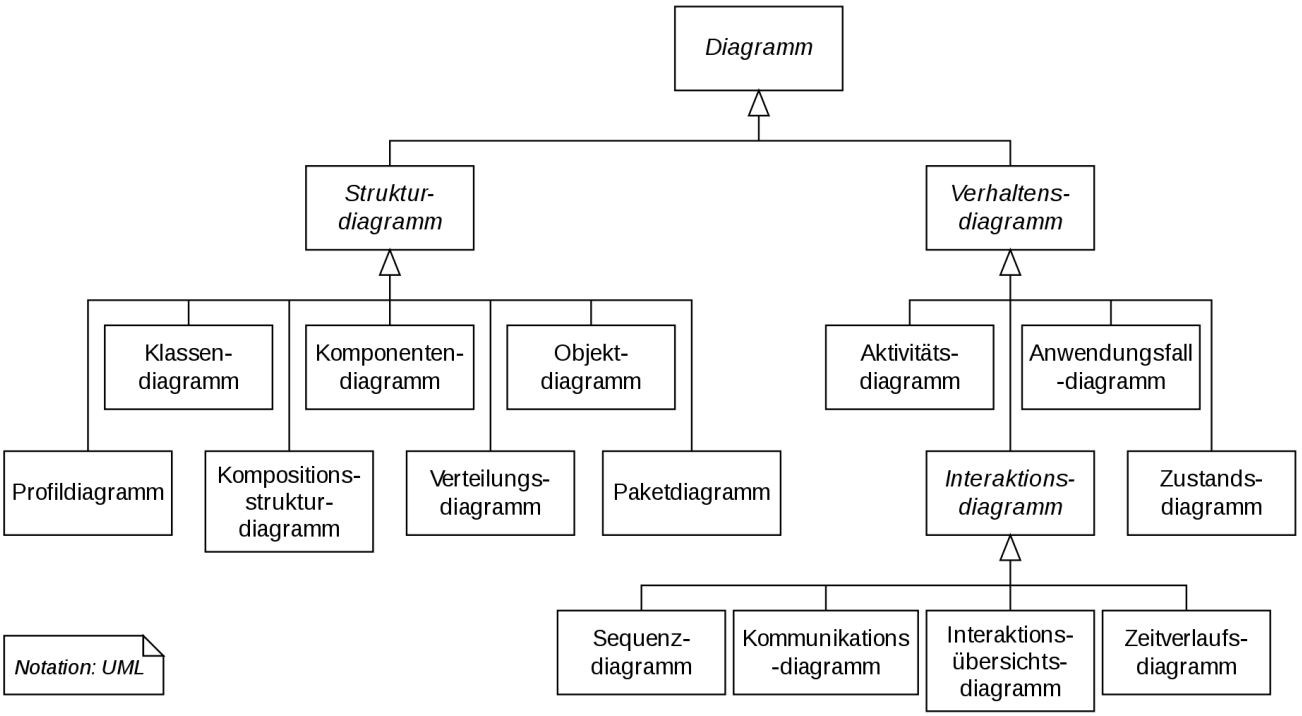
```
@startuml
robust "Web Browser" as WB
concise "Web User" as WU

@0
WU is Idle
WB is Idle

@100
WU is Waiting
WB is Processing

@300
WB is Waiting
@enduml
```




```
https://www.plantuml.com/plantuml/png/JSun2iCm38NXNQSGEK5ABo2yTUjYD3kEA8RK
vhm1
```

[WikiDoxygen] https://commons.wikimedia.org/w/index.php?curid=24966914, Doxygen-Beispielwebseite, Autor Der Messer - Eigenes Werk, CC BY-SA 3.0

Diagramm-Typen



UML Diagramm-Typen [WikiUMLDiagrammTypes]

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?
Kompositionsstrukturdiagramm	Wie sieht die interne Struktur einer Klasse, Komponente, eines Subsystems aus?
Komponentendiagramm	Wie lassen sich die Klassen zu wiederverwendbaren Komponenten (Module, Bibliotheken etc) zusammenfassen und wie werden deren Beziehungen definiert?
Verteilungsdiagramm	Wie werden Softwareanwendungen und -komponenten auf Hardwareknoten (Server, Geräte, Netzwerke) verteilt?

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? Fließen auch Daten in den Prozess?
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? Wie ist die zeitliche Reihenfolge des Nachrichtenaustauschs?
Kommunikationsdiagramm	Wer tauscht mit wem welche Informationen aus?
Timing-Diagramm	Wie hängen die Zustände verschiedener Akteure zeitlich voneinander ab?
Interaktionsübersichtsdiagramm	Über welche Interaktionen verfügt das System, wie laufen sie ab?

[WikiUMLDiagrammTypes] https://upload.wikimedia.org/wikipedia/commons/thumb/d/da/UML-Diagrammhierarchie.svg/1200px-UML-Diagrammhierarchie.svg.png, Autor "Stkl"- derivative work: File: UML-Diagrammhierarchie.png: Sae1962, CC BY-SA 4.0

Aufgaben

- [ ] Schließen Sie das Bearbeiten der Aufgabe 3 im GitHub Classroom
- [ ] Machen Sie sich mit den SOLID Prinzipien weiter vertraut: