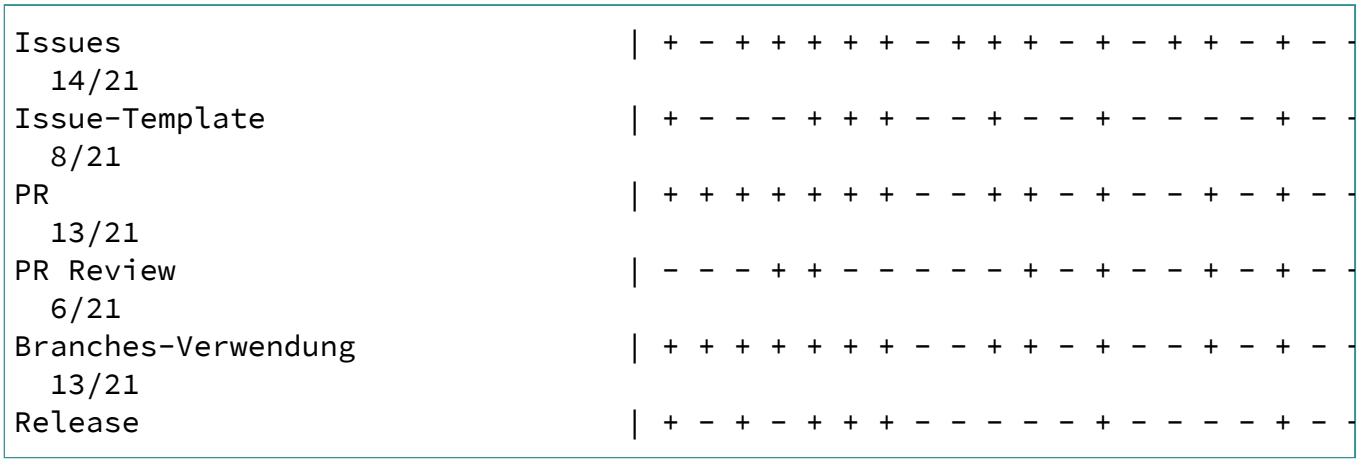
Modellierung von Software

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Teil:	`13/27`
Semester	`Sommersemester 2023`
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/13_UML_Modellierung.md
Autoren	Sebastian Zug, Galina Rudolf & André Dietrich

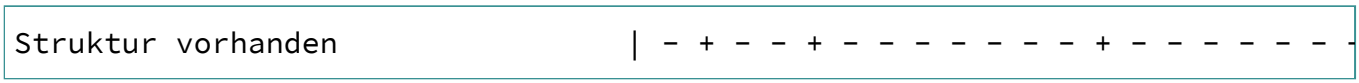
Neues aus Github

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.

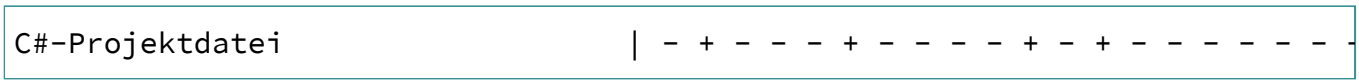
- Verwenden Sie wichtige Github-Features wie



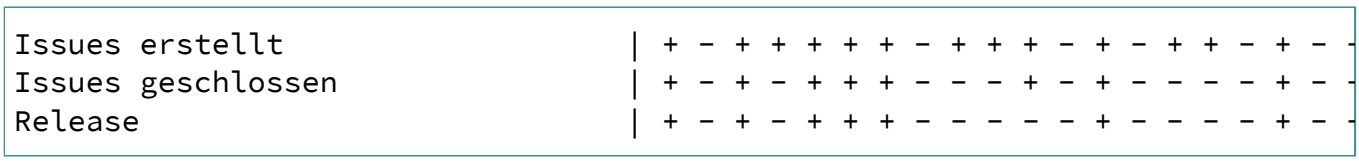
- Strukturieren Sie Ihr Projekt in Ordnern (C# Beispiel, Python Beispiel, Text)



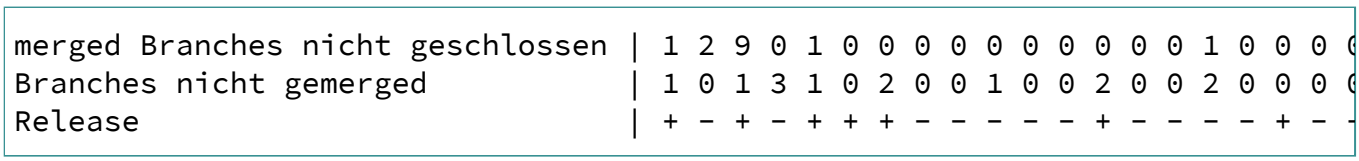
- Fügen Sie immer alle notwendigen Informationen für die Installation bei (Config Files, Pipenv etc., im konkreten Fall Projekt-Datei)



- Umgang mit Issues



- Umgang mit Branches



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

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

SpaceStation

```
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.

```
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.

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 Vererbungshierachie deutlich wartungsfreundlicher realisieren lässt!

Iniitalisation

```
1  using System;
2
3  public class Employee
4  {
5    public string Name {set; get;}
6    public int ID {set; get;}
7
8    public Employee(int id, string name){
9      this.ID = id; this.Name = name;
10   }
11
12   public decimal CalculateBonus(decimal salary){
13     return salary * 0.1M;
14   }
15 }
16
17 public class Program
18 {
19   public static void Main(string[] args){
20     Employee Bernhard = new Employee(1, "Bernhard");
21     Console.WriteLine($"Bonus = {Bernhard.CalculateBonus(11234)}");
22   }
23 }
```

```
Bonus = 1123.4
```

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.

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

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 bein Untertyp von T ist."* [Liskov]

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. Die zugehörigen Membervariablen sind nicht unabhängig von einander.

[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 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()
```

```
    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

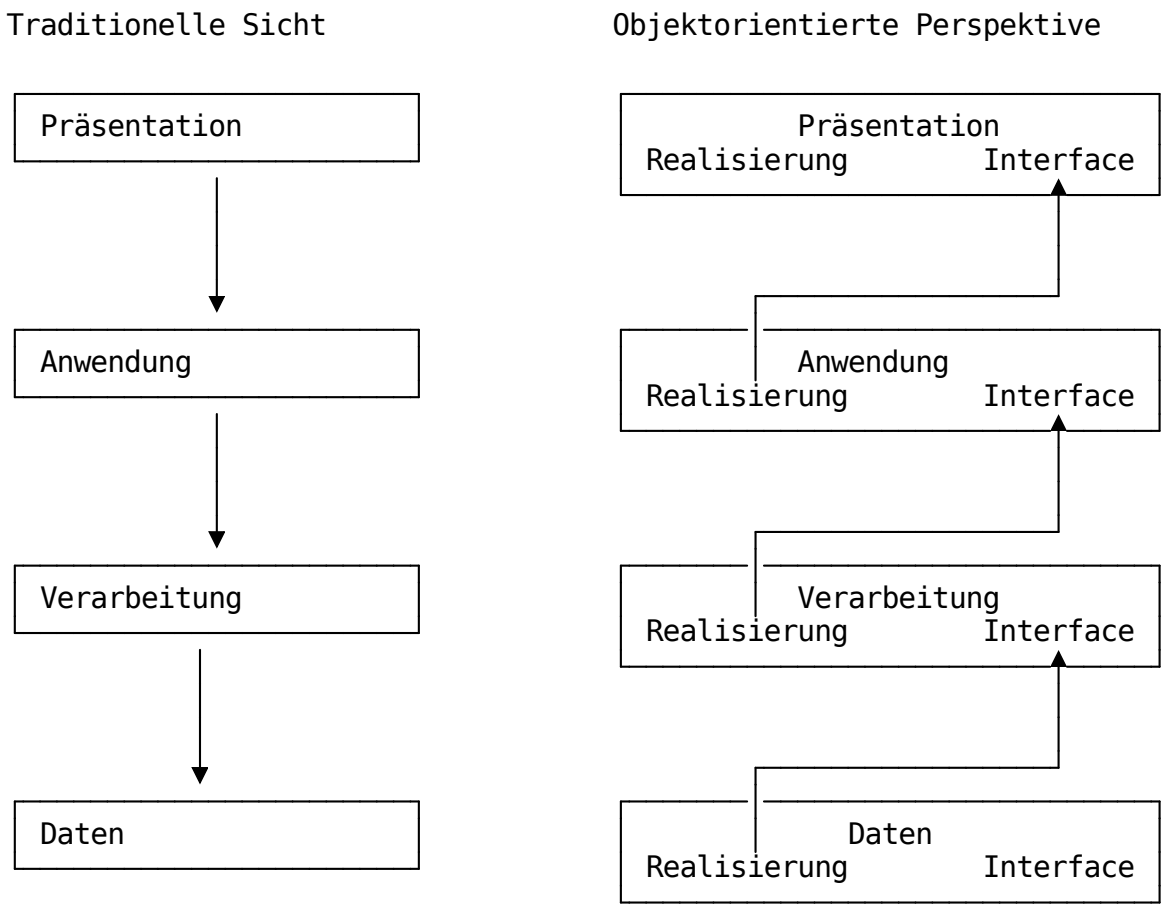
- übersichtlichere kleinere Schnittstellen, die flexibler kombiniert werden können
- Klassen umfassen keine Methoden, die sie nicht benötigen

→ 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.



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 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 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)
//        {
//            messages.SendMessage();
//        }
//    }
//}
```

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

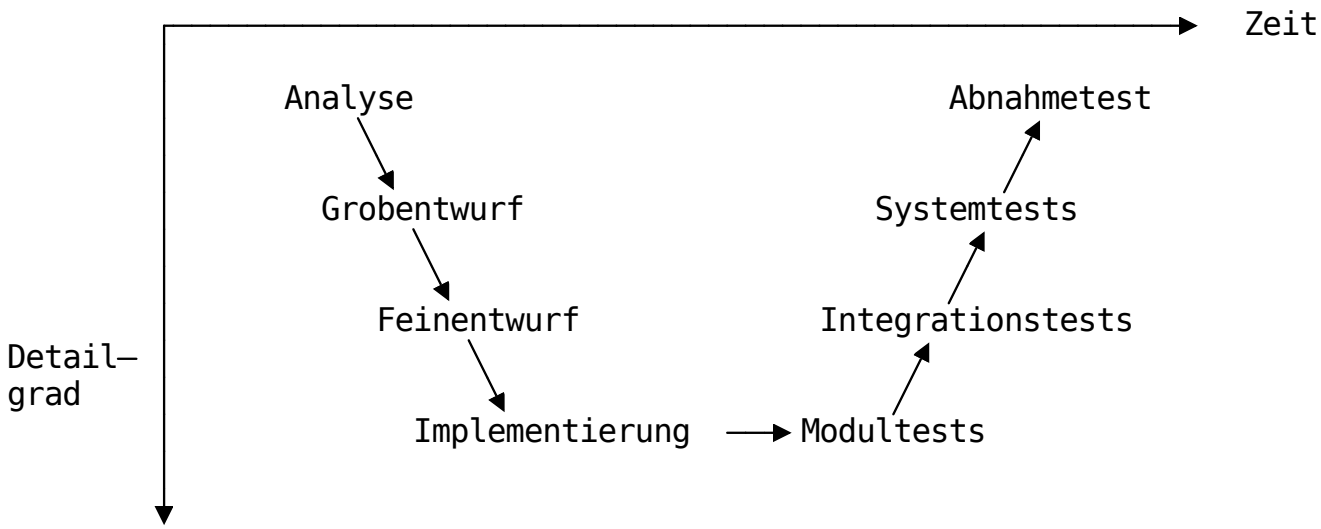
Herausforderungen bei der Umsetzung der Prinzipien

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 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 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

[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]

[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-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.

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

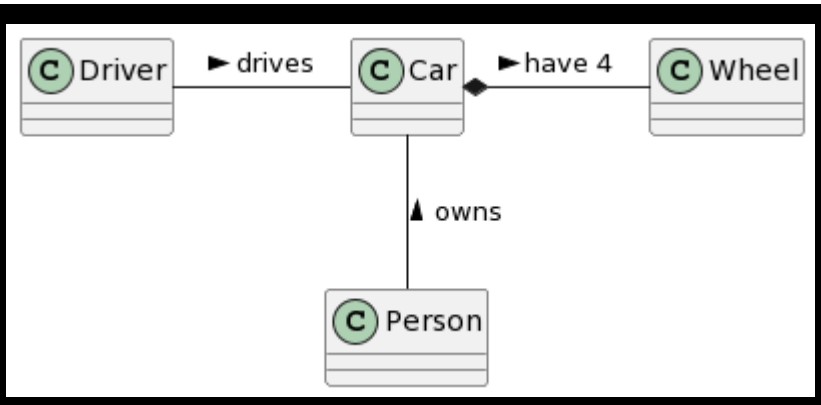
http://plantuml.com/de/

ClassDiagram

```
@startuml
class Car

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

@enduml
```



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

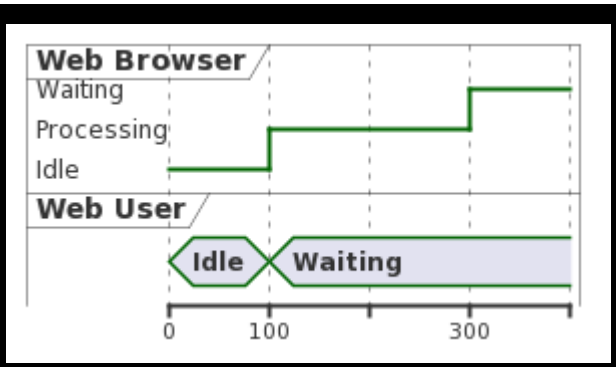
GanttChart

```
@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	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.

Modell vs. Diagramm

[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

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