

# **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



# Contents

<b>1 Einführung</b>	<b>9</b>
Zielstellung der Veranstaltung . . . . .	10
Qualifikationsziele / Kompetenzen . . . . .	10
Zielstellung der Veranstaltung . . . . .	10
Wozu brauche ich das? . . . . .	11
Organisatorisches . . . . .	12
Dozenten . . . . .	12
Ablauf . . . . .	13
Struktur der Vorlesungen . . . . .	13
Durchführung . . . . .	13
Prüfungen . . . . .	15
Zeitaufwand und Engagement . . . . .	15
Literaturhinweise . . . . .	16
Werkzeuge der Veranstaltung . . . . .	16
Markdown . . . . .	16
GitHub . . . . .	20
Entwicklungsumgebungen . . . . .	20
Aufgaben . . . . .	21
<b>2 Softwareentwicklung als Prozess</b>	<b>23</b>
Softwareentwicklung . . . . .	23
Begriffsdefinition . . . . .	23
Lebenszyklus einer Software . . . . .	23
Methodische Ziele . . . . .	24
Und warum der ganze Aufwand? . . . . .	25
Komplexität von Software . . . . .	25
Fehler in der Softwareentwicklung . . . . .	26
Konsequenzen von Fehlern im Prozess . . . . .	26
Und im Kleinen ... . . . . .	28
Herausforderungen . . . . .	29
Ansätze zur Strukturierung der Aufgaben . . . . .	30
Ok, wir brauchen Unterstützung . . . . .	31
Aufgaben . . . . .	31
<b>3 .NET und Einordnung der Sprache C</b>	<b>33</b>
Einschub: Programmierparadigmen . . . . .	33
Warum also C#? . . . . .	35
Historie der Sprache C . . . . .	35
Konzepte und Einbettung . . . . .	36
Abgrenzung zu Java . . . . .	39
Es wird konkret ... Hello World . . . . .	39
Aufgaben . . . . .	41
<b>4 C# Grundlagen I</b>	<b>43</b>
Symbole . . . . .	43
Schlüsselwörter . . . . .	44
Variablennamen . . . . .	45
Zahlen . . . . .	45
Zeichenketten . . . . .	46

Kommentare . . . . .	46
Datentypen und Operatoren . . . . .	47
Wertdatentypen . . . . .	48
Character Datentypen . . . . .	48
Zahlendatentypen und Operatoren . . . . .	48
Aufgabe . . . . .	53
<b>5 C# Grundlagen II</b>	<b>55</b>
Auf Nachfrage . . . . .	55
Wertdatentypen und Operatoren (Fortsetzung) . . . . .	56
Boolsscher Datentyp und Operatoren . . . . .	56
Enumerations . . . . .	58
Weitere Wertdatentypen . . . . .	59
Referenzdatentypen . . . . .	59
Array Datentyp . . . . .	61
String Datentyp . . . . .	63
Umgang mit Variablen . . . . .	63
Konstante Werte . . . . .	64
Implizit typisierte Variablen . . . . .	64
Nullable - Leere Variablen . . . . .	65
Aufgaben . . . . .	66
<b>6 C# Grundlagen III</b>	<b>67</b>
I/O Schreiboperation . . . . .	67
Einzelner Datentyp . . . . .	67
Kombinierte Formatierung von Strings . . . . .	68
Zeichenfolgeninterpolation . . . . .	70
Ziele der Schreiboperationen . . . . .	71
Beispiel . . . . .	72
I/O Leseoperationen . . . . .	73
Kommandozeilenargumente . . . . .	73
Leseoperationen von der Console . . . . .	74
Transformation der Eingaben . . . . .	75
Ausnahmebehandlungen . . . . .	76
Best Practice . . . . .	77
Beispiel Exception-Handling . . . . .	78
Aufgaben . . . . .	78
<b>7 Programmfluss und Funktionen</b>	<b>81</b>
Anweisungen . . . . .	81
Verzweigungen . . . . .	81
Schleifen . . . . .	86
Sprünge . . . . .	88
Funktionen in C . . . . .	88
Verkürzte Darstellung . . . . .	89
Übergeben von Parametern . . . . .	90
Parameterlisten . . . . .	93
Benannte und optionale Argumente . . . . .	93
Überladen von Funktionen . . . . .	94
Aufgaben . . . . .	94
<b>8 OOP Motivation</b>	<b>97</b>
Structs . . . . .	97
Structs in C . . . . .	98
Herausforderung this . . . . .	99
Konstruktoren . . . . .	100
Veränderliche und nicht veränderliche Felder . . . . .	103
Sichtbarkeitsattribute . . . . .	104
Vergleich mit Klassen . . . . .	105
Beispiel der Woche . . . . .	107
Aufgaben . . . . .	108

<b>9 OOP Konzepte I</b>	<b>109</b>
Auf Nachfrage . . . . .	109
Visionen der Objektorientierung . . . . .	110
Kapselung . . . . .	112
Vererbung . . . . .	113
Polymorphie . . . . .	114
Weitere Beispiele . . . . .	115
Begriffe . . . . .	116
Klassen in C . . . . .	116
Felder . . . . .	117
Konstanten . . . . .	118
Konstruktoren . . . . .	119
Destruktoren / Finalizer . . . . .	122
Eigenschaften . . . . .	122
Indexer . . . . .	123
Operatorenüberladung in C# . . . . .	124
Beispiel der Woche . . . . .	127
Aufgaben . . . . .	128
<b>10 Vererbung</b>	<b>129</b>
Auf Nachfrage . . . . .	129
Vererbung in C . . . . .	129
Zugriffsmechanismen . . . . .	131
Member der Klasse . . . . .	132
Klasse . . . . .	133
Polymorphie in C . . . . .	133
Laufzeitprüfung . . . . .	134
Grundidee der Polymorphie . . . . .	135
Überschreiben von Methoden . . . . .	135
Verdecken von Methoden . . . . .	137
Versiegeln von Klassen oder Membern . . . . .	138
Casts über Klassen . . . . .	138
Beispiel . . . . .	140
Aufgaben . . . . .	140
<b>11 Abstrakte Klassen und Interfaces</b>	<b>141</b>
Auf Nachfrage . . . . .	141
Abstrakte Klassen / Abstrakte Methoden . . . . .	141
Interfaces . . . . .	142
Interfaces vs. Abstrakte Klassen . . . . .	144
Bedeutung von Interfaces . . . . .	144
Auflösung von Namenskonflikten . . . . .	145
Aufgaben . . . . .	145
<b>12 Versionsverwaltung I</b>	<b>147</b>
Motivation . . . . .	147
Lösungsansatz . . . . .	148
Strategien zur Konfliktvermeidung . . . . .	149
Mischen von Dokumenten . . . . .	151
Revisionen . . . . .	153
Formen der Versionsverwaltung . . . . .	155
Git . . . . .	156
Zustandsmodell einer Datei in Git . . . . .	156
Grundlegende Anwendung (lokal!) . . . . .	157
“Kommando zurück” . . . . .	158
Was kann schief gehen? . . . . .	159
Ich sehe was, was Du nicht siehst . . . . .	160
Kommandozeile oder keine Kommandozeile . . . . .	160
Aufgaben . . . . .	162
<b>13 Versionsverwaltung II</b>	<b>163</b>

Verteiltes Versionsmanagement . . . . .	163
Arbeiten mit Branches . . . . .	164
Generieren und Navigation über Branches . . . . .	165
Mergoperationen über Branches . . . . .	165
Rebase mit einem Branch . . . . .	165
Arbeit mit GitHub . . . . .	165
Issues . . . . .	165
Pull requests und Reviews . . . . .	165
Automatisierung der Arbeit . . . . .	166
Ein Wort zur Zusammenarbeit . . . . .	167
Commit Messages . . . . .	167
Generelles Vorgehen . . . . .	168
Aufgaben . . . . .	169
<b>14 Modellierung von Software</b>	<b>171</b>
Neues aus Github . . . . .	171
Motivation des Modellierungsgedankens . . . . .	171
Prinzipien des (objektorientierten) Softwareentwurfs . . . . .	171
Prinzip einer einzigen Verantwortung (Single-Responsibility-Prinzip SRP) . . . . .	172
Open-Closed Prinzip . . . . .	173
Liskovsche Substitutionsprinzip (LSP) . . . . .	173
Interface Segregation Prinzip . . . . .	174
Dependency Inversion Prinzip . . . . .	176
Herausforderungen bei der Umsetzung der Prinzipien . . . . .	178
Unified Modeling Language . . . . .	179
Geschichte . . . . .	179
UML Werkzeuge . . . . .	179
Diagramm-Typen . . . . .	182
Aufgaben . . . . .	183
<b>15 Modellierung von Software</b>	<b>185</b>
Neues aus Github . . . . .	185
UML Diagrammtypen . . . . .	186
Anwendungsfall Diagramm . . . . .	186
Aktivitätsdiagramm . . . . .	187
Sequenzdiagramm . . . . .	189
Klassendiagramme . . . . .	191
Verwendung von UML Tools . . . . .	199
Aufgaben . . . . .	199
<b>16 Modellierung von Software</b>	<b>201</b>
Beispielszenario UML-Modellierung . . . . .	201
Use-Case Diagramm . . . . .	202
Aktivitätsdiagramme . . . . .	202
Klassendiagramme . . . . .	205
<b>17 Modellierung von Software</b>	<b>209</b>
Organisatorisches . . . . .	209
Neues aus der GitHub Woche . . . . .	209
Softwarefehler . . . . .	210
Testen als Teil der Qualitätssicherung . . . . .	211
Definition . . . . .	212
Ablauf beim Testen . . . . .	212
Klassifikation Testmethoden . . . . .	212
Planung von Tests . . . . .	213
Black-Box-Testing / Spezifikationsorientiert . . . . .	215
White-Box-Testing / Strukturorientiert . . . . .	215
Und jetzt konkret! . . . . .	217
Exkurs: Attribute in C . . . . .	217
Idee 1: Eigenen Testmethoden . . . . .	219
Idee 2: Test-Frameworks . . . . .	220

<b>18 Dokumentation und Build-Tools</b>	<b>223</b>
Dokumentation . . . . .	223
Programmiererdokumentation . . . . .	224
Benutzerdokumentation . . . . .	225
Realisierung der Dokumentation in Csharp	225
Paketmanagement . . . . .	230
Build Tools . . . . .	231
dotnet . . . . .	232
MSBuild . . . . .	232
Make . . . . .	233
<b>19 Continuous Integration</b>	<b>235</b>
Exkurs: Alternative Konzepte der Programmentwicklung . . . . .	235
Continuous integration (CI) . . . . .	236
CI Umsetzung mit GitHub . . . . .	237
Anwendungsbeispiel 1 . . . . .	239
Anwendungsbeispiel 2 . . . . .	240
Realisierung der Projektstruktur . . . . .	241
Automatischer Build Prozess . . . . .	241
Generierung der Dokumentation . . . . .	242
Erweiterung der Tests . . . . .	243
Aufgaben der Woche . . . . .	243
<b>20 Generics</b>	<b>245</b>
Motivation . . . . .	245
Generische Typen . . . . .	247
Generische Methoden . . . . .	249
Beschränkungen . . . . .	252
Vererbung bei generischen Typen . . . . .	253
Anwendung . . . . .	254
Aufgaben der Woche . . . . .	256
<b>21 Collections</b>	<b>257</b>
Collections . . . . .	257
Containerimplementierung in Csharp . . . . .	260
Anwendung der Generic Collections . . . . .	263
Achtung! . . . . .	265
Aufgaben der Woche . . . . .	265
<b>22 Delegaten</b>	<b>267</b>
Motivation und Konzept der Delegaten . . . . .	267
Grundidee . . . . .	269
Was passiert hinter den Kulissen? . . . . .	270
Multicast Delegaten . . . . .	270
Schnittstellen vs. Delegaten . . . . .	271
Praktische Implementierung . . . . .	272
Anonyme / Lambda Funktionen . . . . .	272
Generische Delegaten . . . . .	274
Action / Func . . . . .	275
Aufgaben der Woche . . . . .	276
<b>23 Events</b>	<b>277</b>
Organisatorisches . . . . .	277
Nachgefragt . . . . .	278
Wiederholung . . . . .	278
Motivation und Idee der Events . . . . .	279
Publish-Subscribe Prinzip . . . . .	279
Events in C . . . . .	279
Events - Praktische Implementierung . . . . .	282
Parameter . . . . .	282
Generic Events . . . . .	283

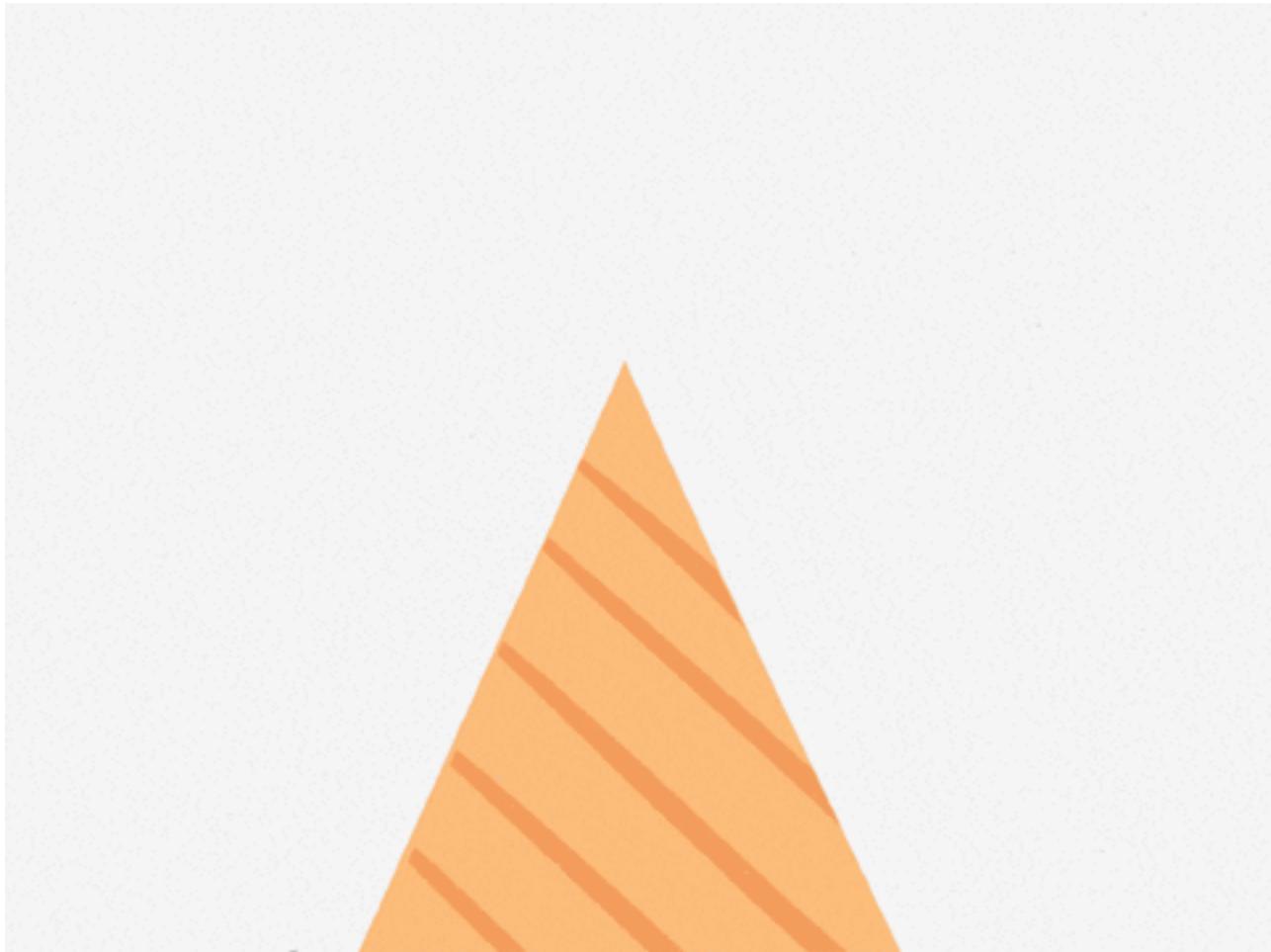
<b>24 Threads</b>	<b>285</b>
Neues aus GitHub . . . . .	285
Motivation - Threads . . . . .	285
Implmementierung unter C . . . . .	286
Thread-Interaktion . . . . .	288
Thread-Initialisierung . . . . .	289
Datenaustausch zwischen Threads . . . . .	291
Locking . . . . .	292
Hintergrund und Vordergrund-Threads . . . . .	292
Ausnahmebehandlung mit Threads . . . . .	293
Thread-Pool . . . . .	294
Aufgaben der Woche . . . . .	295
<b>25 Tasks</b>	<b>297</b>
Logging . . . . .	297
Tasks . . . . .	297
Task Modell in C . . . . .	299
Asynchrone Methoden . . . . .	301
Aufgaben der Woche . . . . .	303
<b>26 Language-Integrated Query</b>	<b>305</b>
Motivation . . . . .	305
Exkurs SQL . . . . .	308
LINQ Umsetzung . . . . .	310
Exkurs “Erweiterungsmethoden” . . . . .	310
Exkurs “Anonyme Typen” . . . . .	313
Exkurs “Enumarables” . . . . .	314
LINQ - Grundlagen . . . . .	315
Hinter den Kulissen . . . . .	319
Basisfunktionen von LINQ . . . . .	320
Filtern . . . . .	320
Gruppieren . . . . .	321
Sortieren . . . . .	322
Ausbaben . . . . .	323
Aufgabe der Woche . . . . .	324
<b>27 Entwurfsmuster</b>	<b>325</b>
Wiederholung - Polymorphie . . . . .	325
Design Pattern . . . . .	328
Kategorien . . . . .	328
Erzeugungsmuster - Singleton Pattern . . . . .	329
Strukturmuster Adapter Pattern . . . . .	332
Erzeugungsmuster (Abstract) Factory Pattern . . . . .	334
Verhaltensmuster State Pattern . . . . .	335
Aufgabe der Woche . . . . .	340
Ausblick . . . . .	340

# Chapter 1

## Einführung

Parameter	Kursinformationen
<b>Veranstaltung:</b>	Vorlesung Softwareentwicklung
<b>Semester</b>	Sommersemester 2022
<b>Hochschule:</b>	Technische Universität Freiberg
<b>Inhalte:</b>	Motivation der Vorlesung "Softwareentwicklung" und Beschreibung der Organisation der Veranstaltung
<b>Link auf den GitHub:</b>	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/00_Einfuehrung.md">https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/00_Einfuehrung.md</a>
<b>Autoren</b>	@author

## Zielstellung der Veranstaltung



### Qualifikationsziele / Kompetenzen

Studierende sollen ...

- die Konzepte objektorientierten und interaktiven Programmierung verstehen,
- die Syntax und Semantik einer objektorientierten Programmiersprache beherrschen um Probleme kollaborativ bei verteilter Verantwortlichkeit von Klassen von einem Computer lösen lassen,
- in der Lage sein, interaktive Programme unter Verwendung einer objektorientierten Klassenbibliothek zu erstellen.

[Auszug aus dem Modulhandbuch 2020]

## Zielstellung der Veranstaltung

*Wir lernen effizient guten Code in einem kleinen Team zu schreiben.*

Genereller Anspruch	Spezifischer Anspruch
Verstehen verschiedener Programmierparadigmen UNABHÄNGIG von der konkreten Programmiersprache	Objektorientierte (und funktionale) Programmierung am Beispiel von C#
praktische Einführung in die methodische Softwareentwicklung	Arbeit mit ausgewählten UML Diagrammen und Entwurfsmustern
Grundlagen der kooperativ/kollaborative Programmierung und Projektentwicklung	Verwendung von Projektmanagementtools und einer Versionsverwaltung für den Softwareentwicklungsprozess

Obwohl Einstimmigkeit darüber besteht, dass kooperative Arbeit für Ingenieure Grundlage der täglichen Arbeitswelt ist, bleibt die Wissensvermittlung im Rahmen der Ausbildung nahezu aus.

**Frage:** Welche Probleme sehen Sie bei der Teamarbeit?

**Spezifisches Ziel:** Wir wollen Sie für die Konzepte und Werkzeuge der kollaborativen Arbeit bei der Softwareentwicklung „sensibilisieren“.

- Wer definiert die Feature, die unsere Lösung ausmachen?
- Wie behalten wir bei synchronen Codeänderungen der Überblick?
- Welchen Status hat die Erfüllung der Aufgabe X erreicht?
- Wie können wir sicherstellen, dass Code in jedem Fall kompiliert und Grundfunktionalitäten korrekt ausführt?
- ...

## Wozu brauche ich das?

Anhand der Veranstaltung entwickeln Sie ein „Gefühl“ für guten und schlechten Code und hinterfragen den Softwareentwicklungsprozess.

### Beispiel 1: Mariner 1 Steuerprogramm-Bug (1962)



Mariner 1 ging beim Start am 22. Juli 1962 durch ein fehlerhaftes Steuerprogramm verloren, als die Trägerrakete vom Kurs abkam und 293 Sekunden nach dem Start gesprengt werden musste. Ein Entwickler hatte einen Überstrich in der handgeschriebenen Spezifikation eines Programms zur Steuerung des Antriebs übersehen und dadurch statt geglätteter Messwerte Rohdaten verwendet, was zu einer fehlerhaften und potenziell gefährlichen Fehlsteuerung des Antriebs führte.

[Link auf Beschreibung des Bugs](#)

**Potentieller Lösungsansatz:** Testen & Dokumentation

### Beispiel 2: Toll-Collect On-Board-Units (2003)

Das Erfassungssystem für die Autobahngebühren für Lastkraftwagen sollte ursprünglich zum 31. August 2003 gestartet werden. Nachdem die organisatorischen und technischen Mängel offensichtlich geworden waren, erfolgte eine mehrfache Restrukturierung. Seit 1. Januar 2006 läuft das System, mit einer Verzögerung von über zwei Jahren, mit der vollen Funktionalität. Eine Baustelle war die On-Board-Units (OBU), diese konnte zunächst nicht in ausreichender Stückzahl geliefert und eingebaut werden, da Schwierigkeiten mit der komplexen Software der Geräte bestanden.

Die On-Board-Units des Systems

---

<sup>1</sup>wikimedia, Autor: NASA, [Link](#)

- reagierten nicht auf Eingaben
- ließen sich nicht ausschalten
- schalteten sich grundlos aus
- zeigten unterschiedliche Mauthöhen auf identischen Strecken an
- wiesen Autobahnstrecken fehlerhaft als mautfrei/mautpflichtig aus

**Potentieller Lösungsansatz:** Testen auf Integrationsebene, Projektkoordination

## Organisatorisches



## Dozenten

Name	Email
Sebastian Zug	sebastian.zug@informatik.tu-freiberg.de
Galina Rudolf	galina.rudolf@informatik.tu-freiberg.de
Nico Sonack	nico.sonack@student.tu-freiberg.de
Felix Busch	Felix.Busch@student.tu-freiberg.de
Anne Gierig	anne.gierich@student.tu-freiberg.de

## Ablauf

Jetzt wird es etwas komplizierter ... die Veranstaltung kombiniert nämlich zwei Vorlesungen:

	<i>Softwareentwicklung (SWE)</i>	<i>Einführung in die Softwareentwicklung (EiS)</i>
Hörerkreis	Fakultät 1 + interessierte Hörer	Fakultät 4 - Studiengang Engineering
Leistungspunkte	6	
Vorlesungen	28 (2 Feiertage)	15 (bis 31. Mai 2021)
Übungen	ab Mai 2 x wöchentlich	ab 25. April 8 Übungen
Prüfungsform	Klausur oder Projekt	zusätzliches Python Tutorial ab Juni maschinenbauspezifisches Software-Projekt (im Wintersemester 2021/22) Prüfungsvoraussetzung: Erfolgreiche Bearbeitung der finalen Aufgabe im Sommersemester

**Ermunterung an unsere EiS-Hörer:** Nehmen Sie an der ganzen Vorlesungsreihe teil. Den Einstieg haben Sie ja schon gelegt ...

## Struktur der Vorlesungen

Woche	Tag	Inhalt der Vorlesung	Bemerkung
1	4. April	Organisation, Einführung von GitHub und LiaScript	
	8. April	Softwareentwicklung als Prozess	
2	11. April	Konzepte von Dotnet und C#	
	15. April	<i>Karfreitag</i>	
3	18. April	<i>Ostermontag</i>	
	22. April	Elemente der Sprache C# (Datentypen)	
4	25. April	Elemente der Sprache C# (Forts. Datentypen)	
	29. April	Elemente der Sprache C# (Ein-/Ausgaben)	
5	2. Mai	Programmfluss und Funktionen	
	6. Mai	Strukturen / Konzepte der OOP	
6	9. Mai	Säulen Objektorientierter Programmierung	
	13. Mai	Klassenelemente in C# / Vererbung	
7	16. Mai	Klassenelemente in C# / Vererbung	
	20. Mai	Versionsmanagement im Softwareentwicklungsprozess	
8	23. Mai	Git und Continuous integration in GitHub	
	27. Mai	UML Konzepte	
9	30. Mai	UML Diagrammtypen	Ende EiS Vorlesungsinhalte
	3. Juni	UML Anwendungsbeispiel	
10	6. Juni	<i>Pfingstmontag</i>	
	10. Juni	Testen	
11	13. Juni	Dokumentation	
	17. Juni	Build Toolchains und ihr Einsatz	
12	20. Juni	Generics	
	24. Juni	Container	
13	27. Juni	Delegaten	
	1. Juli	Events	
14	4. Juli	Threadkonzepte in C#	
	8. Juli	Taskmodell	
15	11. Juli	Language Integrated Query	
	15. Juli	Design Pattern	

## Durchführung

Die Vorlesung wurden im vergangenen Semester aufgezeichnet. Die Inhalte finden sich unter  
<https://teach.informatik.tu-freiberg.de/b/seb-blv-unz-kxu>

Diese Materialien können der Nachbereitung der Veranstaltung dienen, ersetzen aber nicht den Besuch der Vorlesung.

Die Vorlesung findet

- Montags, 11:00 - 12:30
- Freitags, 9:15 - 10:45

im Audimax 1001 statt.

Die Materialien der Vorlesung sind als Open-Educational-Ressources konzipiert und stehen unter Github bereit.

Wie können Sie sich einbringen?

- **Allgemeine theoretische Fragen/Antworten** ... Dabei können Sie sich über github/ das Opal-Forum in die Diskussion einbringen.
- **Rückmeldungen/Verbesserungsvorschläge zu den Vorlesungsmaterialien** ... „*Das versteht doch keine Mensch! Ich würde vorschlagen ...*“ ... dann korrigieren Sie uns. Alle Materialien sind Open-Source. Senden Sie mir einen Pull-Request und werden Sie Mitautor.

Die Übungen bestehen aus selbständig zu bearbeitenden Aufgaben, wobei einzelne Lösungen im Detail besprochen werden. Wir werden die Realisierung der Übungsaufgaben über die Plattform GitHub abwickeln.

Wie können Sie sich einbringen?

- **Allgemeine praktische Fragen/Antworten** ... in den genannten Foren bzw. in den Übungsveranstaltungen
- **Eigene Lösungen** ... Präsentation der Implementierungen in den Übungen
- **Individuelle Fragen** ... an die Übungsleiter per Mail oder in einer individuellen Session

Für die Übungen werden wir Aufgaben vorbereiten, mit denen die Inhalte der Vorlesung vertieft werden. Wir motivieren Sie sich dafür ein Gruppen von 2 Studierenden zu organisieren.



Index	C#	GitHub	Teamarbeit	Inhalte / Teilaufgaben	Woche
0	Basics	nein	nein	Toolchain, Datentypen, Fehler, Ausdrücke, Kontrollfluss, Arrays	5
1				static Funktionen, Klasse und Struktur, Nullables	6
2					7
3	-	ja	ja	Github am Beispiel von Markdown	8
4	OOP	ja	ja	Einführungsbeispiel OOP,	9

Index	C#	GitHub Teamarbeit	Inhalte / Teilaufgaben	Woche
5	OOP	ja	Anwendungsbeispiel: Computersimulation Vererbung, virtuelle Methoden, Indexer, Überladene Operatoren,	10-11
6	OOP	ja	Anwendungsbeispiel: Smartphone (Entwurf mit UML) Vererbung, abstract, virtuell, Generics	12-13
7	OOP	ja	Anwendungsbeispiel: Zoo Generische Collections, Delegaten, Events Anwendungsbeispiel: ????????	14-15

## Prüfungen

In der Klausur werden neben den Programmierfähigkeiten und dem konzeptionellen Verständnis auch die Werkzeuge der Softwareentwicklung adressiert!

- **Softwareentwicklung:** Konventionelle Klausur ODER Programmieraufgabe in Zweier-Team anhand einer selbstgewählten Aufgabe
- **Einführung in die Softwareentwicklung:** Teamprojekt und Projektpräsentationen (im Wintersemester 2022/23) bei bestandener Prüfungsvorleistung in Form einer Teamaufgabe im Sommersemester

Ergebnisse der Klausur Softwareentwicklung 2020

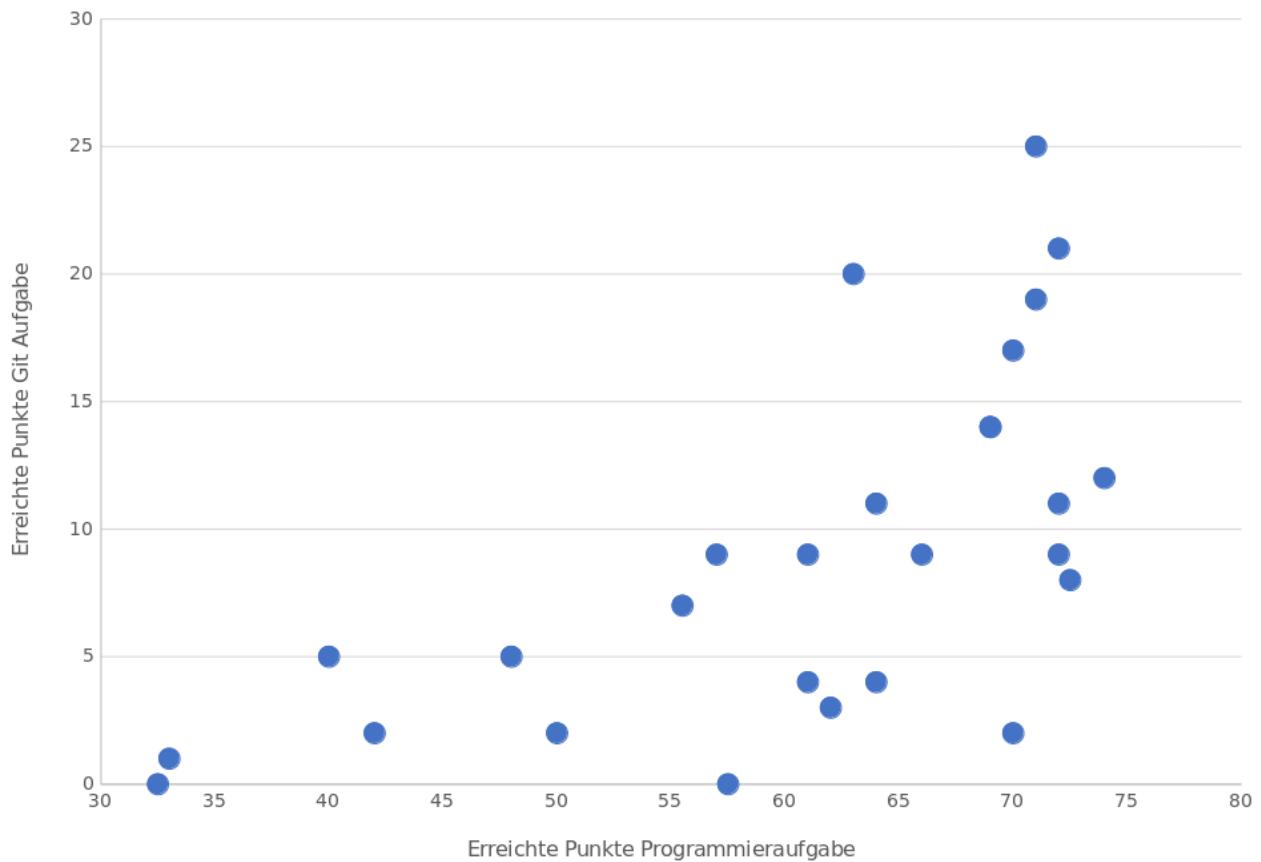


Figure 1.1: Klausurergebnisse Softwareentwicklung 2020

## Zeitaufwand und Engagement

Mit der Veranstaltung Softwareentwicklung verdienen Sie sich 9 CP/6 CP. Eine Hochrechnung mit der von der Kultusministerkonferenz vorgegebenen Formel  $1 \text{ CP} = 30 \text{ Zeitstunden}$  bedeutet, dass Sie dem Fach im Mittel über dem Semester 270 Stunden widmen sollten ... entsprechend bleibt neben den Vorlesungen und Übungen genügend Zeit für die Vor- und Nachbereitung der Lehrveranstaltungen, die eigenständige Lösung von Übungsaufgaben sowie die Prüfungsvorbereitung.

“Erzähle mir und ich vergesse. Zeige mir und ich erinnere. Lass es mich tun und ich verstehe.”

– (*Konfuzius, chin. Philosopher 551-479 v. Chr.*)

### Wie können Sie zum Gelingen der Veranstaltung beitragen?

- Stellen Sie Fragen, seien Sie kommunikativ!
- Geben Sie uns Rückmeldungen in Bezug auf die Geschwindigkeit, Erklärmuster, etc.
- Organisieren Sie sich in *interdisziplinären* Arbeitsgruppen!
- Lösen Sie sich von vermeindlichen Grundwahrheiten:
  - “*in Python wäre ich drei mal schneller gewesen*”
  - “*VIM ... mehr Editor braucht kein Mensch!*”

### Literaturhinweise

Literaturhinweise werden zu verschiedenen Themen als Links oder Referenzen in die Unterlagen integriert.

Es existiert eine Vielzahl kommerzielle Angebote, die aber einzelne Aspekte in freien Tutorial vorstellen. In der Regel gibt es keinen geschlossenen Kurs sondern erfordert eine individuelle Suche nach spezifischen Inhalten.

- **Online-Kurse:**

- Leitfaden von Microsoft für C# aber auch die Werkzeuge
- C# Tutorial for Beginners: Learn in 7 Days [englisch]
- Einsteiger Tutorials [deutsch]
- Programmierkonzepte von C#

- **Video-Tutorials:**

- !?Umfangreicher C# Kurs mit guten konzeptionellen Anmerkungen
- !?Einsteigerkurs als Ausgangspunkt für eine Tutorienreihe
- !?Absoluter Einsteigerkurs

### Algorithmen

- Codebeispiele

- **Bücher:**

- C# 7.0 in a Nutshell - J. Albahari, B. Albahari, O'Reilly 2017
- Kompaktkurs C# 7 - H. Mössenböck, dpunkt.verlag

## Werkzeuge der Veranstaltung

Was sind die zentralen Tools unserer Veranstaltung?

- *Vorlesungstool* -> BigBlueButton für die Aufzeichnungen aus dem vergangenen Semester [Introduction](#)
- *Entwicklungsplattform* -> [GitHub](#)
- *Beschreibungssprache für Lerninhalte* -> [LiaScript](#)

### Markdown

Markdown wurde von John Gruber und Aaron Swartz mit dem Ziel entworfen, die Komplexität der Darstellung so weit zu reduzieren, dass schon der Code sehr einfach lesbar ist. Als Auszeichnungselemente werden entsprechend möglichst kompakte Darstellungen genutzt.

Markdown ist eine Auszeichnungssprache für die Gliederung und Formatierung von Texten und anderen Daten. Analog zu HTML oder LaTex werden die Eigenschaften und Organisation von Textelementen (Zeichen, Wörtern, Absätzen) beschrieben. Dazu werden entsprechende “Schlüsselwörter” verwendet um den Text zu strukturieren.

#### # Überschrift

\_eine \*\*Hervorhebung\*\* in kursiver Umgebung\_

- \* Punkt 1
- \* Punkt 2

Und noch eine Zeile mit einer mathematischen Notation  $a=\cos(b)$ !

---

## Überschrift

eine Hervorhebung in kursiver Umgebung

Punkt 1

Punkt 2

Und noch eine Zeile mit einer mathematischen Notation  $a = \cos(b)$ !

Eine gute Einführung zu Markdown finden Sie zum Beispiel unter:

- [MarkdownGuide](#)
- [GitHubMarkdownIntro](#)

Mit einem entsprechenden Editor und einigen Paketen macht das Ganze dann auch Spaß

- Wichtigstes Element ist ein Previewer, der es Ihnen erlaubt “online” die Korrektheit der Eingaben zu prüfen
- Tools zur Unterstützung komplexerer Eingaben wie zum Beispiel der Tabellen (zum Beispiel für Atom mit [markdown-table-editor](#))
- Visualisierungsmethoden, die schon bei der Eingabe unterstützen
- Rechtschreibprüfung (!)

## Vergleich mit HTML

Im Grunde wurde Markdown erfunden um nicht umständlich HTML schreiben zu müssen und wird zumeist in HTML übersetzt. Das dargestellte Beispiel zeigt den gleichen Inhalt wie das Beispiel zuvor, es ist jedoch direkt viel schwerer zu editieren, dafür bietet es weit mehr Möglichkeiten als Markdown. Aus diesem Grund erlauben die meisten Markdown-Dialekte auch die Nutzung von HTML.

```
<h1>Überschrift</h1>

<i>eine <b>Hervorhebung</b> in kursiver Umgebung</i>

<ul>
  <li>Punkt 1</li>
  <li>Punkt 2</li>
</ul>
```

Und noch eine Zeile mit einer mathematischen Notation

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <semantics>
    <mrow>
      <mi>a</mi>
      <mo>=</mo>
      <mi>c</mi>
      <mi>o</mi>
      <mi>s</mi>
      <mo stretchy="false">(</mo>
      <mi>b</mi>
      <mo stretchy="false">)</mo>
    </mrow>
    <annotation encoding="application/x-tex">
      a=\cos(b)
    </annotation>
  </semantics>
</math>!
```

## Vergleich mit LaTex

Eine vergleichbare Ausgabe unter LaTex hätte einen deutlich größeren Overhead, gleichzeitig eröffnet das Textsatzsystem (über einzubindende Pakete) aber auch ein wesentlich größeres Spektrum an Möglichkeiten und Features (automatisch erzeugte Numerierungen, komplexe Tabellen, Diagramme), die Markdown nicht umsetzen kann.

```
\documentclass[12pt]{article}
\usepackage[latin1]{inputenc}
\begin{document}
\section{Überschrift}
\textit{eine \emph{Betonung} in kursiver Umgebung}
\begin{itemize}
\item Punkt 1
\item Punkt 2
\end{itemize}
Und noch eine Zeile mit einer mathematischen Notation $a=\cos(b)$!
\end{document}
```

Das Ergebnis sieht dann wie folgt aus:

# 1 Überschrift

*eine Betonung in kursiver Umgebung*

- Punkt 1
- Punkt 2

Und noch eine Zeile mit einer mathematischen Notation  $a = \cos(b)$ !

Figure 1.2: pdflatexScreenshot

## LiaScript

Das Problem der meisten Markup-Sprachen und vor allem von Markdown ist, dass die Inhalte nicht mehr nur für ein statisches Medium (Papier/PDF) geschrieben werden. Warum sollte ein Lehrinhalt (vor allem in der Informatik), der voran am Tablet/Smartphone/Notebook konsumiert wird nicht interaktiv sein?

LiaScript erweitert Markdown um interaktive Elemente wie:

- Ausführbarer Code
- Animationen & Sprachausgaben
- Visualisierung
- Quizze & Umfragen
- Erweiterbarkeit durch JavaScript und Macros
- ...

## Einbindung von PlantUML zur Generierung von UML-Diagrammen

@startuml

```
abstract class AbstractList
abstract AbstractCollection
interface List
interface Collection
```

```

List <|-- AbstractList
Collection <|-- AbstractCollection

Collection <|- List
AbstractCollection <|- AbstractList
AbstractList <|-- ArrayList

class ArrayList {
    Object[] elementData
    size()
}

enum TimeUnit {
    DAYS
    HOURS
    MINUTES
}

annotation SuppressWarnings

@enduml
@plantUML.eval(png)

```

**Ausführbarer C# Code**

Wichtig für uns sind die ausführbaren Code-Blöcke, die ich in der Vorlesung nutze, um Beispielimplementierungen zu evaluieren. Dabei werden zwei Formen unterschieden:

**C# 10 mit dotnet Unterstützung**

```

using System;
using System.Collections.Generic;
using System.Collections;
using System.Linq;
using System.Text;

int n;
Console.Write("Number of primes: ");
n = int.Parse(Console.ReadLine());

ArrayList primes = new ArrayList();
primes.Add(2);

for(int i = 3; primes.Count < n; i++) {
    bool isPrime = true;
    foreach(int num in primes) isPrime &= i % num != 0;
    if(isPrime) primes.Add(i);
}

Console.Write("Primes: ");
foreach(int prime in primes) Console.WriteLine($"{prime}");

<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>
</Project>

```

**C# 8 mit mono**

```
using System;
```

```

namespace HelloWorld
{
    public class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Glück auf!");
        }
    }
}

```

**Frage:** Welche Unterschiede sehen Sie zwischen C#8 und C#10 Code schon jetzt?

\*\* Quellen & Tools\*\*

- Das Projekt: <https://github.com/liascript/liascript>
- Die Webseite: <https://liascript.github.io>
- Nützliches
  - Dokumentation zu LiaScript
  - YouTube Kanal zu LiaScript
- Editoren
  - Für den Atom-Editor von GitHub existieren derzeit zwei Plugins:
    1. [liascript-preview](#)
    2. [liascript-snippets](#)
  - Für den Einsatz anderer Editoren eignet sich auch der LiaScript-DevServer

## GitHub

Der Kurs selbst wird als “Projekt” entwickelt. Neben den einzelnen Vorlesungen finden Sie dort auch ein Wiki, Issues und die aggregierten Inhalte als automatisch generiertes Skript.

Link zum GitHub des Kurses: [https://github.com/TUBAF-Ifl-LiaScript/VL\\_Softwareentwicklung](https://github.com/TUBAF-Ifl-LiaScript/VL_Softwareentwicklung)

### Hinweise

1. Mit den Features von GitHub machen wir uns nach und nach vertraut.
2. Natürlich bestehen neben Github auch alternative Umsetzungen für das Projektmanagement wie das Open Source Projekt GitLab oder weitere kommerzielle Tools BitBucket, Google Cloud Source Repositories etc.

## Entwicklungsumgebungen

Seien Sie neugierig und probieren Sie verschiedene Tools und Editoren aus!

- Atom  
!?[Tutorial](#)
- Visual Studio Code  
!?[Tutorial](#)
- VIM/gVIM / neoVIM  
!?[C# Vim Development Setup](#)
- weitere ...

## Aufgaben

- [ ] Legen Sie sich einen GitHub Account an (sofern dies noch nicht geschehen ist).
- [ ] Installieren Sie einen Editor Ihrer Wahl auf Ihrem Rechner, mit dem Sie Markdown-Dateien komfortabel bearbeiten können.
- [ ] Nutzen Sie das Wiki der Vorlesung um Ihre neuen Markdown-Kenntnisse zu erproben und versuchen Sie sich an folgenden Problemen:
- Recherchieren Sie weitere Softwarebugs. Dabei interessieren uns insbesondere solche, wo der konkrete Fehler direkt am Code nachvollzogen werden konnte.
- Fügen Sie eine kurze Referenz auf Ihren Lieblingseditor ein und erklären Sie, warum Sie diesen anderen Systemen vorziehen. Ergänzen Sie Links auf Tutorials und Videos, die anderen nützlich sein können.



# Chapter 2

## Softwareentwicklung als Prozess

Parameter	Kursinformationen
<b>Veranstaltung</b>	Vorlesung Softwareentwicklung
Semester	Sommersemester 2022
<b>Hochschule:</b>	Technische Universität Freiberg
Inhalte:	Definition des Softwarebegriffes und der Herausforderungen
Link auf den GitHub:	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/01_Software.md">@author</a>
Autoren	@author

## Softwareentwicklung

### Begriffsdefinition

Begriff	Definitionsansatz
Software als "Medium"	<i>Software [ist] all das, was zum Funktionieren eines Computers notwendig, aber nicht Hardware ist.</i> <i>Software ist sinnlich nicht wahrnehmbar ... . Sie ist komplex und besteht aus umfangreichen Texten</i>
Software als Ziel	<i>Software macht den Computer nutzbar</i>
Software als Prozess	<i>Software ermöglicht die Abbildung von Prozessen auf einem Rechner</i> <i>Software ist die Idee, die Lösung, die man sich für ein Problem ausgedacht hat, das Verfahren, das helfen soll ...</i> <i>Dabei sind Computerprogramme nicht nur als Beschreibung der auszuführenden Funktionen ... Vereinbarung zur Nutzung, ... Dokumentationsinhalte.</i>

### Lebenszyklus einer Software

Ein Software-Lebenszyklus beschreibt den gesamten Prozess der Herstellung und des Betriebs Implementierung ausgehend von der kundenseitigen Problemstellung über die Realisierung und den Betrieb bis hin zur Ablösung der Software durch einen Nachfolger.

1. Problemstellung
2. Analyse Entwurf
3. Implementierung
4. Test
5. Markteinführung
6. Pflege/Wartung

Welche Querbeziehungen (“Während der Tests wird erkannt, dass die Implementierung Mängel aufweist.”) zwischen den einzelnen Stufen sehen Sie?

**Welche Definitionen ergeben sich daraus für den Entwicklungsprozess?**

“Unter dem Begriff Softwareentwicklung versteht man die Konzeption und standardisierte Umsetzung von Softwareprojekten und die damit verbundenen Prozesse.” [^Freund, S. 25]

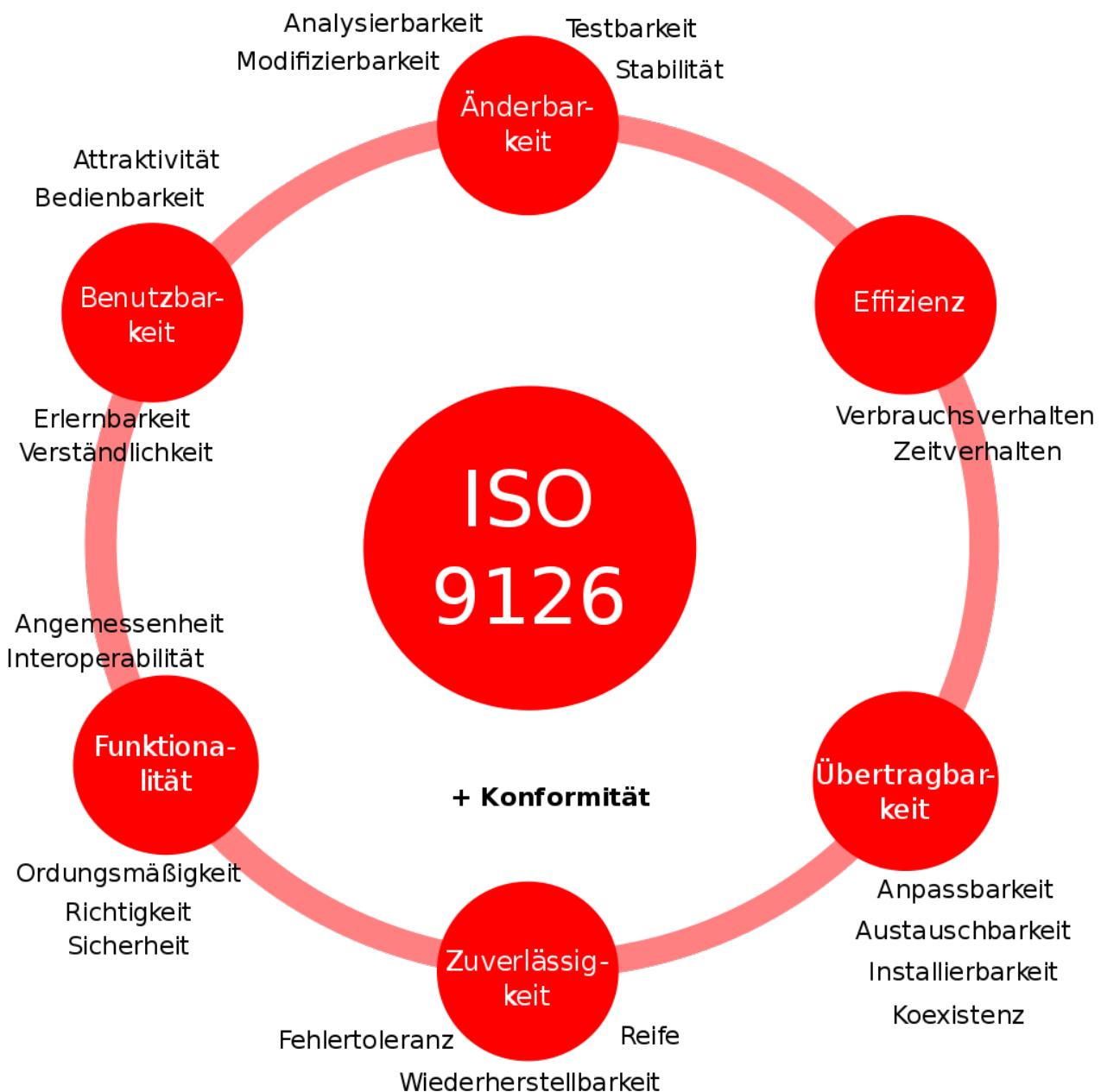
... oder insbesondere auf große Projekte zielend

„Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Softwaresystemen.“ [^Balzert, S. 36]

## Methodische Ziele

Was heißt das, “ingenieurmäßig” oder “standardisiert”?

Gemäß ISO 9126 gibt es die sechs folgenden Qualitätsmerkmale für Softwareprodukte:



<sup>1</sup>

<sup>1</sup>Von Sae1962 - Eigenes Werk, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=52216179>

Nachfolger ISO 25010: Zusätzlich

- Kompatibilität
- Sicherheit

Die Norm kann als eine Art Checkliste verstanden werden.

## Und warum der ganze Aufwand?

### Komplexität von Software

Steigende Komplexität der Softwareprodukte ...

Projekt/Produkt	Lines of Code	Jahr
Unix v 1.0	10.000	1971
Win32/Smile Virus	10.000	2002
Space shuttle	400.000	
Windows 3.1	2.300.000	1992
HD DVD Player (XBox)	4.500.000	2001
Firefox	9.900.000	2010
Android	12.000.000	
F-35 Flugzeug	24.000.000	2013
Facebook	62.000.000	
Autonomes Fahrzeug	100.000.000	

Quelle unter anderem <sup>2</sup> und [McCandless]

Und wann entsteht der Aufwand? Wann muss ein Team Kosten in die Entwicklung investieren?

Projektpphase		Relativer Kosteanteil
Spezifikation und Architekturentwurf	Entwicklung	16%
Detailentwurf und Kodierung		9%
Test		16%
Anpassung	Wartung	12%
Erweiterung und Verbesserung		36%
Fehlerbehebung		12%

Zahlwerte aus einem Diagramm in <sup>3</sup>

**Merke:** Die Entwicklung kleiner Programme unterscheidet sich von der Entwicklung großer Programme!

Kriterium	Kleine Programme	Große Programme
Zeilenzahl	bis zu ein paar 1000 Zeilen	Millionen von LOC
Einsatz	“Eigengebrauch”	kommerzieller Einsatz von Dritten
Anforderungsanalyse	Idee	präzise Spezifikation
Vorgehensmodell	unstrukturiert	strukturierter Entwicklungsprozesse
Test und Validierung	unter Realbedingungen am Endprodukt	Systematische Prüfstrategie
Komplexität	Überschaubare Zahl von Komponenten, Abhängigkeiten usw.	Hohe Komplexität, explizite Organisation in Struktureinheiten und Modulen
Dokumentation	Fehlt in der Regel	zwingend erforderlich, permanente Pflege
Planung und Organisation	Kaum Planung und Projektorganisation	zwingend erforderlich

<sup>2</sup>Dragan Radovanovic, Kif Leswing, “Google runs on 5000 times more code than the original space shuttle”, 2016, [Link](#)

<sup>3</sup>Prof. Dr. Thorsten Lemburg, Einführung in die Softwareentwicklung, [Link](#)

Darstellung entsprechen motiviert aus <sup>4</sup>.

[^McCandless] David McCandless [https://informationisbeautiful.net/visualizations/million-lines-of-code/] (https://informationisbeautiful.net/visualizations/million-lines-of-code/)

## Fehler in der Softwareentwicklung

Die Zusammenfassung wurde durch die Ausführungen von <sup>5</sup> motiviert

### 1. Management

- Es wird mit der Codierung sofort angefangen.
- Eine Festlegung der Anforderungen/Qualitätsmerkmale fehlt
- Eine Abnahme der Phasenergebnisse erfolgt nicht
- Ein Vorgehensmodell fehlt, bzw. wird nicht verfolgt
- Die Schulung für die Software-Erststeller und -Anwender wird vernachlässigt oder als nicht notwendig angesehen
- Die Terminvorgaben sind unrealistisch und nicht koordiniert.
- Begriffe werden nicht definiert

### 2. Architektur

- Die Systemarchitektur ist nicht oder nur sehr umständlich erweiterbar (fehlende Datenkapselung, fehlende Modularität)
- Es wird ein unnützes Maß an Komplexität in den Entwurf integriert: “*Es könnte doch sein, dass ...*”

### 3. Entwicklungfluss

- Die Auswahl der Werkzeuge/Methoden ist unzureichend vorbereitet
- Es wird nicht systematisch bzw. unzureichend getestet.
- Standards und Richtlinien werden nicht beachtet.

### 4. Dokumentation

- Schlechte Namensvergabe wie z.B. File-, Klassen-, Methoden- und Variablennamen
- Die Dokumentation fehlt bzw. ist veraltet, unzureichend oder nicht adäquat

## Konsequenzen von Fehlern im Prozess

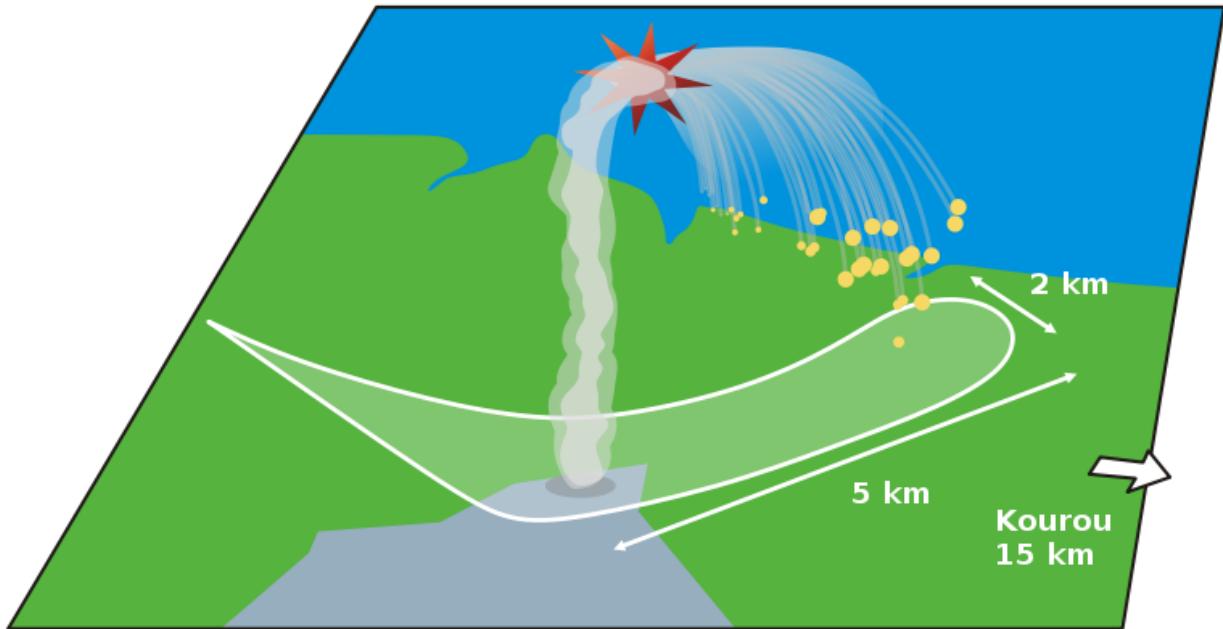
### Ariane Jungfernflug

V88 war die Startnummer des Erstflugs der europäischen Schwerlast-Trägerrakete Ariane 5 am 4. Juni 1996. Die Rakete trug die Seriennummer 501. Der Flug endete etwa 40 Sekunden nach dem Start, als die Rakete nach einer Ausnahmesituation in der Software der Steuereinheit plötzlich vom Kurs abkam und sich kurz darauf selbst zerstörte. Vier Cluster-Forschungssatelliten zur Untersuchung des Erdmagnetfelds gingen dabei verloren (Schaden 290 Millionen Euro).

---

<sup>4</sup>Prof. Dr. Thorsten Lemburg, Einführung in die Softwareentwicklung - Seminar: Softwareentwicklung in der Wissenschaft, [Link](#)

<sup>5</sup>Prof. Dr. Thorsten Lemburg, Einführung in die Softwareentwicklung, [Link](#)



6

```
-- Overflow is correctly handled for the vertical component
L_M_BV_32 := TBD.T_ENTIER_16S((1.0 / C_M_LSB_BH) *
                                G_M_INFO_DERIVE(T_ALG.E_BH));

if L_M_BV_32 > 32767 then
    P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;           -- largest 16Bit number (Two's complement)
elseif L_M_BV_32 < -32768 then
    P_M_DERIVE(T_ALG.E_BV) := 16#8000#;           -- smallest negative 16Bit number
else
    P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TBD.T_ENTIER_16S(L_M_BV_32));
end if;

-- But not for the horizontal one
P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS(TBD.T_ENTIER_16S
                                         ((1.0 / C_M_LSB_BH) *
                                          G_M_INFO_DERIVE(T_ALG.E_BH));
```

### Mars Rover

Mars Pathfinder war ein US-amerikanischer Mars-Lander, der 1996 von der NASA eingesetzt wurde. Er brachte 1997 den ersten erfolgreichen Mars-Rover Sojourner auf die Marsoberfläche.



Figure 2.1: Mars Rover

Nach dem Beginn der Aufzeichnung von meteorologischen Daten mit dem Sojourner traten plötzlich scheinbar zufällige System-Zurücksetzungen auf. Das Betriebssystem bootete neu, was mit einem Datenverlust einherging. Diese Fehler waren aber auch schon auf der Erde aufgetreten ...

Durch Analyse des Logbuches zu diesem Zeitpunkt konnte festgestellt werden, dass es bei der Programmierung

---

<sup>6</sup>Wikimedia, Autor Phrd, Fragment fallout zone of failed Ariane 501 launch. [Link](#)

von "Sojourner" ein Problem gab. Dabei schlug die sogenannte **Prioritäten-Inversion** zu, die sich zeigt, wenn mehrere Prozesse ein und die selbe Ressource nutzen.

Weitere Informationen unter [What the media coudn't tell you about Mars Pathfinder](#)

### Agressiver Gandhi

Der als friedlich bekannte Inder Mahatma Gandhi ist im Spiele Civilization 5 dafür bekannt, besonders gerne nukleare Waffen zu nutzen. Diese Affinität ist einen Programmierfehler im ersten Teil zuzuordnen, in welchem der Aggressionswert bestimmt, wie wahrscheinlich es ist, dass der Herrscher eine atomare Waffe benutzt. Gandhi startet dort mit einem Aggressionswert von 1, jedoch bekommt jede Demokratie bei Spielstart -2 Aggressionspunkte, was zu einem Wert von -1 führt. Binär betrachtet entspricht das folgender 8 Bit Zahl:

1111 1111

Dieser Wert wird intern aber als ein unsigned char, also wie eine 8 Bit vorzeichenlosen Ganzzahl behandelt. Dies führt dazu, dass nicht eine -1 gelesen wird, sondern der Maximalwert dieses Datentyps 255.

### Und im Kleinen ...

Das folgende anschauliche Beispiel und die zugehörige Analyse ist durch ein Beispiel der Vorlesung "Software Engineering" von Prof. Dr. Schürr, (TU Darmstadt) motiviert.

**Achtung:** Das folgende Codebeispiel enthält eine Fülle von Fehlern!

```
#include <stdio.h>

FILE *fp;

void main()
{
    fp = fopen("numbers.txt", "r");
    int a[10];
    int num = 0, l = 0;

    while(1){
        if (fscanf(fp, "%d", &num) == 1) {
            a[l] = num;
            l++;
        } else {
            break;
        }
    }
    for(int i=0; i<l; i++)
        printf("%5i ",a[i]);
    printf("\n");

    int aux;
    for(int i=2; i<l; i++){
        for(int j=1; j>i; j--){
            if (a[j-1] > a[j]){
                aux = a[j-1];
                a[j-1] = a[j];
                a[j] = aux;
            }
        }
    }
    for(int i=0; i<l; i++)
        printf("%5i ",a[i]);

    printf("\nAus Maus!\n");
}
```

**Aufgabe:** Lesen Sie den Code, erklären Sie die Aufgabe, die er löst und identifizieren Sie Fehlerquellen.

Welche Probleme sehen Sie im Hinblick auf die zuvor genannten Qualitätsmerkmale

Aspekt	Bewertung
Funktionalität	?
Zuverlässigkeit	
Benutzbarkeit	
Effizienz	
Wartungsfreundlichkeit	
Übertragbarkeit	

Aspekt	Bewertung
Funktionalität	feste Feldlänge, das Programm stürzt bei mehr als 10 Einträgen ab
Zuverlässigkeit	keine Überprüfung der Existenz der Datei, kein Schließen der Datei
Benutzbarkeit	im Programmcode enthaltene Dateinamen, feste Feldlänge
Effizienz	quadratischer Aufwand der Sortierung
Wartungsfreundlichkeit	fehlende Dokumentation, unverständliche Variablenbezeichner, redundante Codeelemente
Übertragbarkeit	

## Herausforderungen

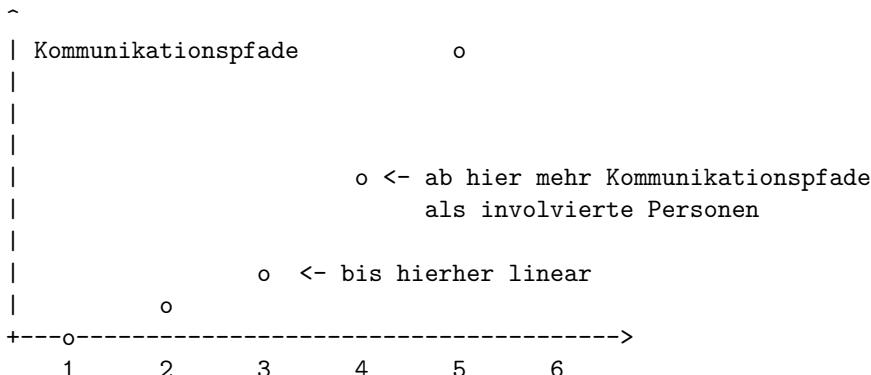
### Warum ist Softwareentwicklung so herausfordernd

- Die Größe der zu lösenden Probleme. Software ist nicht einfacher, als die Probleme, die sie löst. Je größer und schwieriger die Software, desto aufwendiger und schwieriger ist die Entwicklung.
- Die Tatsache, dass Software ein immaterielles Produkt ist. Die Immaterialität macht das Arbeiten mit Software schwieriger als dasjenige mit materiellen Produkten vergleichbarer Komplexität, da die Risiken auch schwerer zu erkennen sind.
- Sich permanent verändernde Ziele aufgrund der Evolution. Schon das Bestimmen und Erreichen fixierter Ziele bei der Entwicklung ist keine leichte Aufgabe. Sich verändernde Ziele machen das ganze nochmal um eine Größenordnung schwieriger.
- Fehler infolge von Fehleinschätzungen zur Skalierung ("was im Kleinen geht, geht genauso im Großen"). Software-Entwicklung wird daher unbewusst meist als viel einfacher eingeschätzt, als sie tatsächlich ist. Dies führt zu unrealistischen Erwartungen und zu von Beginn an zu tiefen Kosten- und Terminschätzungen.
- Funktionierende Einzelkomponenten stellen noch lange kein funktionierendes Gesamtsystem sicher.

### Der Faktor Mensch

In einem Projekt stellt sich die Frage wie viele Personen involviert sind und damit welche Komplexität der Entwicklungsfluss hat. In einem Team von 3 Personen wird muss sichergestellt sein, dass eine Information bei 2 Partnern ankommt. Die maximale Anzahl ergibt sich zu

$$N = \frac{n \cdot (n - 1)}{2}$$

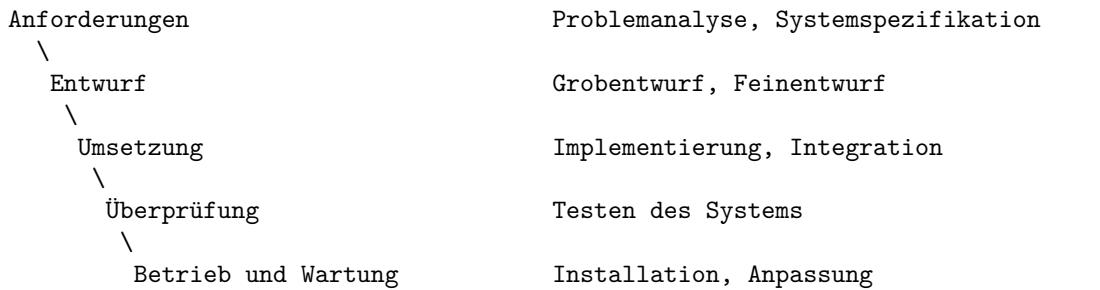


Grafik motiviert aus <sup>7</sup>

Hier ist eine koordinierte Interaktion und Kommunikation notwendig!

## Ansätze zur Strukturierung der Aufgaben

*Erweitertes Wasserfallmodell*



Eigenschaften des Wasserfallmodells:

- Aktivitäten sind in der vorgegebenen Reihenfolge und in der vollen Breite vollständig durchzuführen.
- Am Ende jeder Aktivität steht ein fertiggestelltes Dokument, d.h. das Wasserfallmodell ist ein „dokumentgetriebenes“ Modell.
- Der Entwicklungsablauf ist sequentiell und als Top-down-Verfahren realisiert.
- Es ist einfach, verständlich und benötigt nur wenig Managementaufwand.

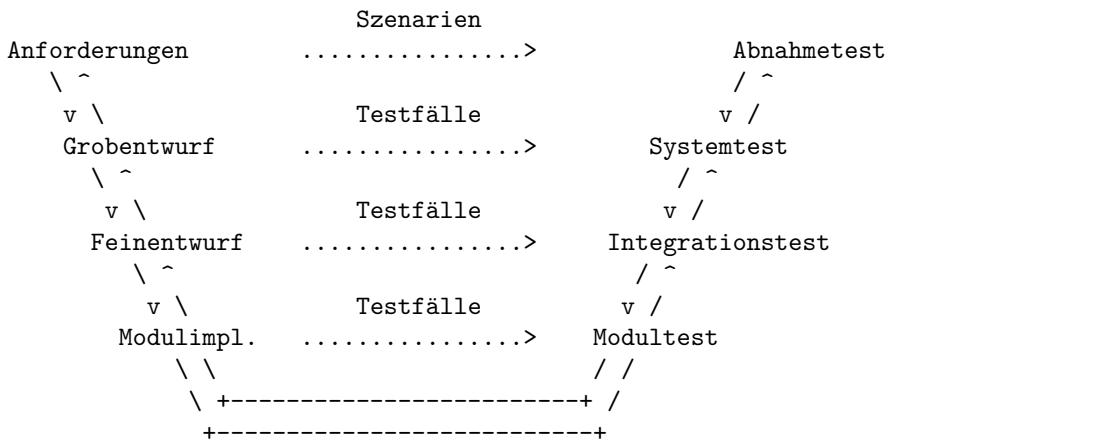
Vorteile:

- klare Abgrenzung der Phasen – einfache Möglichkeiten der Planung und Kontrolle
- bei stabilen Anforderungen und klarer Abschätzung von Kosten und Umfang ein sehr effektives Modell

Nachteile:

- Das Modell ist nur bei einfachen Projekten anwendbar – Unflexibel gegenüber Änderungen und im Vorgehen
- Frühes festschreiben der Anforderungen ist sehr problematisch und kann zu teuren Änderungen führen
- Fehler werden eventuell erst sehr spät erkannt und müssen mit erheblichen Aufwand entfernt werden

*V-Modell*



Tätigkeitsbereiche des V-Modell: Softwareerstellung, Qualitätssicherung, Konfigurationsmanagement, Projektmanagement

Vorteile:

- Integrierte und detaillierte Darstellung von den Tätigkeitsbereichen
- Generisches Modell mit definierten Möglichkeiten zur Anpassung an projektspezifische Anforderungen
- Gut geeignet für große Projekte

Nachteile:

<sup>7</sup>Prof. Dr. Thorsten Lemburg, Einführung in die Softwareentwicklung - Seminar: Softwareentwicklung in der Wissenschaft, [Link](#)

- Für kleine und mittlere Softwareentwicklungen führt das V-Modell zu einem unnötigen Overhead
- Die im V-Modell definierten Rollen (bis zu 25) sind für gängige Softwareentwicklungen nicht realistisch
- explizite Werkzeuge notwendig

**Achtung:** Die Nachteile aus den sehr statisch ausgerichteten Modellen lassen sich für größere Projekte kaum umsetzen. Modernere Ansätze lösen den stringenten Ablauf auf und fokussieren einem schnelleren Einsatz: *Agile Softwareentwicklung, Rational Unified Process, Spiralmodell* - [Link](#)

## Ok, wir brauchen Unterstützung

*CASE is the use of computer-based support in the software development process*

### Klassifikation nach dem Einsatzzweck

- Anforderungsanalyse
  - Spezifikation
  - Modellierung
- Code-Erstellung (Editoren)
  - Editor, IDE
  - Dokumentation
- Ausführung und Testen
  - Compiler, Interpreter
  - IDE, Build-System
  - Debugger
- Koordination Entwicklungsprozess
  - Projektverwaltung
  - Code-Base Management und Versionierung
  - Deployment
  - Support

### Grad der Integration verschiedener Funktionalitäten

- Tools - einzelne Aktivitäten im Software Life-cycle
  - Workbenches - mehrere Werkzeuge
  - Integrated Development Environments - Kombination mehrerer Workbenches und Werkzeuge zur Unterstützung des kompletten Software Life-cycle
- Texteditor vs. Integrated Development Environment (IDE) ... worfür soll ich mich entscheiden?
- Analyse des Workflows und der Formen der Zusammenarbeit
  - Analyse der verwendeten Spezifikations und Modellierungstechniken, Programmiersprachen, etc.
  - Analyse der Komplexität des Vorhabens, der erforderlichen Unterstützung
  - Analyse der Rahmenbedingungen (Betriebssysteme, Kosten)

## Aufgaben

- [ ] Betrachten Sie die Darstellung unter [Webseite Programmwechsel](#) und versuchen Sie die überspitzten Missverständnisse der einzelnen Protagonisten im Kontext eines Softwareprojektes zu stellen.
- [ ] Korrigieren Sie das `allesFalsch.c` Beispiel, verbessern Sie die Lesbarkeit des Codes.



# Chapter 3

## .NET und Einordnung der Sprache C

---

Parameter Kursinformationen

---

**Veranstaltung:** Vorlesung Softwareentwicklung

Semester Sommersemester 2022

**Hochschule:** Technische Universität Freiberg

**Inhalte:** Basiskonzepte von C# und dotnet

**Link auf den** [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/02\\_DotNet.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/02_DotNet.md)

**GitHub:**

**Autoren** @author

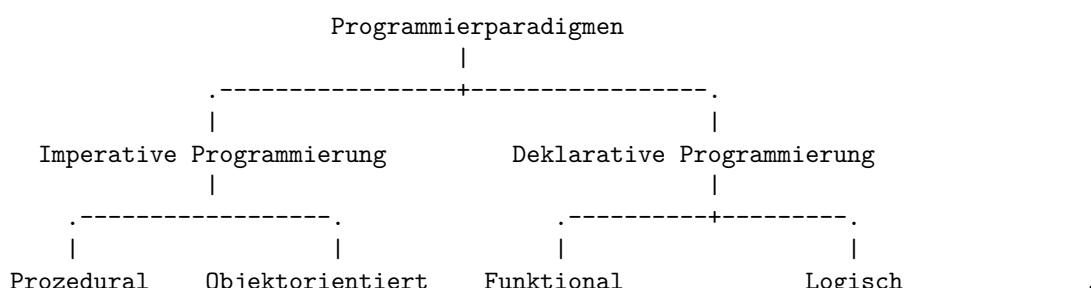
---

### Einschub: Programmierparadigmen

Ein Programmierparadigma bezeichnet die gedankliche, konzeptionelle Grundstruktur die der Darstellung des Problems in Code zugrunde liegt.

Das Programmierparadigma:

- beschreibt den fundamentalen Programmierstil bzw. Eigenschaften von Programmiersprachen
- unterscheidet sich durch die Repräsentation der statischen und dynamischen Programmelemente
- beruht auf Sprache, aber auch auf individuellem Stil.



- **Imperative Programmierung** - Quellcode besteht aus einer Folge von Befehlen die in einer festen Reihenfolge abgearbeitet werden.
  - **Prozedurale Programmierung** - Zerlegung von Programmen in überschaubare Teile, die durch eine definierte Schnittstelle aufrufbar sind (Kernkonzepte: Funktion, Prozedur, Routine, Unterprogramm)
  - **Objektorientierte Programmierung** - Kapselung von Daten und Funktionen in einem Konzept
- **Deklarative Programmierung** - es wird kein Lösungsweg implementiert, sondern nur angegeben, was gesucht ist.

- **Funktionale Sprachen** - Abbildung der Algorithmen auf funktionale Darstellungen
- **Logische Sprachen** - Ableitung einer Lösung aus einer Menge von Fakten, Generierung einer Auswahl von Daten

```
% Prolog Text mit Fakten
mann(adam).
mann(tobias).
mann(frank).
frau(eva).
frau(daniela).
frau(ulrike).
vater(adam,tobias).
vater(tobias,frank).
vater(tobias,ulrike).
mutter(eva,tobias).
mutter(daniela,frank).
mutter(daniela,ulrike).

grossvater(X,Y) :-
    vater(X,Z),
    vater(Z,Y).

grossvater(adam,frank).
```

- **Weiter Konzepte** ... keine spezifische Zuordnbarkeit

- **Strukturierte Programmierung** ... Verzicht bzw. Einschränkung des Goto Statements zugunsten von Kontrollstrukturen (Kernkonzepte: Verzweigungen, Schleifen)
- **Nebenläufig, Reflektiv, Generisch, ...**
- **Aspektorientierte Programmierung,**

Viele Sprachen unterstützen verschiedene Elemente der Paradigmen, bzw. entwickeln sich in dieser Richtung weiter.

Sprache	imperativ	deklarativ
Pascal	prozedural	
C	prozedural	
Ada	objektorientiert	
Java	objektorientiert	
Python	objektorientiert,	funktional
C#	prozedural, objektorientiert	funktional
C++	prozedural, objektorientiert	funktional
Haskell		funktional
Prolog		Logisch
SQL		Logisch

Viele Paradigmen in einer Sprache am Beispiel eines Python Programmes ... Berechnen Sie die Summe der Ziffern eines Arrays.

```
my_list = range(0,10)

# imperative
result = 0
for x in my_list:
    result += x
print("Result in imperative style      :" + str(result))

# procedural
result = 0
def do_add(list_of_numbers):
    result = 0
    for x in my_list:
```

```

        result += x
    return result
print("Result in procedural style      :" + str(do_add(my_list)))

# object oriented
class MyClass(object):
    def __init__(self, any_list):
        self.any_list = any_list
        self.sum = 0
    def do_add(self):
        self.sum = sum(self.any_list)
create_sum = MyClass(my_list)
create_sum.do_add()
print("Result in object oriented style :" + str(create_sum.sum))

# functional
import functools
result = functools.reduce(lambda x, y: x + y, my_list)
print("Result in functional style     :" + str(result))

```

@Pyodide.eval

“Das ist ja alles gut und schön, aber ich ich bin C Programmierer!”

**Anti-Pattern “Golden Hammer”:** *If all you have is a hammer, everything looks like a nail.*

Lösungsansätze: \* Individuell - Hinterfragen des Vorgehens und der Intuition, bewusste Weiterentwicklung des eigenen Horizontes (ohne auf jeden Zug aufzuspringen) \* im Team - Teilen Sie Ihre Erfahrungen im Team / der Community, besetzen Sie Teams mit Mitarbeitern unterschiedlichen Backgrounds (Technical Diversity)

Weitere Diskussion unter: <https://sourcemaking.com/antipatterns/golden-hammer>

## Warum also C#?

C# wurde unter dem Codenamen *Cool* entwickelt, vor der Veröffentlichung aber umbenannt. Der Name C Sharp leitet sich vom Zeichen Kreuz (#, englisch sharp) der Notenschrift ab, was dort für eine Erhöhung des Grundtons um einen Halbton steht. C sharp ist also der englische Begriff für den Ton *cis* (siehe Anspielung auf C++)

### C#

- ist eine moderne und nur in überschaubarem Maße durch die eigene Entwicklung “verschandelte” Sprache
- enthält Elemente vieler verschiedener Paradigmen
- ist plattformunabhängig
- bietet eine breite Sammlung von Bibliotheken
- integriert Bibliotheken und Konzepte für die GUI-Programmierung
- kann mit anderen Sprachen über .NET interagieren
- unterstützt Multi-Processing problemlos
- ist typsicher
- ...

## Historie der Sprache C

Version Jahr .NET	Version C#	Ergänzungen
2002 1.0	1.0	
2006 3.0	2.0	Generics, Anonyme Methoden, Iteratoren, Private setters, Delegates
2007 3.5	3.0	Implizit typisierte Variablen, Objekt- und Collection-Initialisierer, Automatisch implementierte Properties, LINQ, Lambda Expressions
2010 4.0	4.0	Dynamisches Binding, Benannte und optionale Argumente, Generische Co- und Kontravarianz
2012 4.5	5.0	Asynchrone Methoden

Version Jahr .NET	Version C#	Ergänzungen
2015 4.6	6.0	Exception Filters, Indizierte Membervariablen und Elementinitialisierer, Mehrzeilige String-Ausdrücke, Implementierung von Methoden mittels Lambda-Ausdruck
2017 4.6.2/.NET Core	7.0	Mustervergleiche (Pattern matching), Binärliterale 0b..., Tupel
2019 .NET Core 3	8.0	Standardimplementierungen in Schnittstellen, Switch Expressions, statische lokale Funktionen, Index-Operatoren für Teilmengen
2020 .NET 5.0	9.0	Datensatztypen (Records), Eigenschafteninitialisierung, Anweisungen außerhalb von Klassen, Verbesserungen beim Pattern Matching
2021 .NET 6.0	10.0	Erforderliche Eigenschaften, Null-Parameter-Prüfung, globale Using-Statements

Die Sprache selbst ist unmittelbar mit der Ausführungsumgebung, dem .NET Konzept verbunden und war ursprünglich stark auf Windows Applikationen zugeschnitten.

## Konzepte und Einbettung

.NET ist ein Sammelbegriff für mehrere von Microsoft/Dritten herausgegebene Software-Plattformen, die der Entwicklung und Ausführung von Anwendungsprogrammen dienen. Dabei erlebt die Plattform einen permanenten Wandel. Die Bedeutung der einzelnen Teile und Technologien, die .NET umfasst, hat sich im Laufe der Zeit gewandelt. Stand November 2020 spielen folgende Frameworks eine herausgehobene Rolle in der Praxis:

- das nur unter Windows unterstützte klassische .NET Framework, das mit der Version 4.8 in der letzten Version vorliegt.
- das als dessen Nachfolger positionierte, auf verschiedenen Plattformen unterstützte Framework .NET 5 (bei dem auch einige Techniken gekündigt wurden). Es wurde mehrere Jahre parallel unter der Bezeichnung .NET Core entwickelt.
- die Plattform Mono und darauf basierende Techniken (von Microsoft meist als Xamarin bezeichnet). Diese unterstützt seit längerem .NET auf verschiedenen Plattformen (in der Vergangenheit jedoch oft unvollständig implementiert).

Mono ist eine alternative, ursprünglich unabhängige Implementierung von Microsofts .NET Standards. Sie ermöglicht die Entwicklung von plattformunabhängiger Software auf den Standards der Common Language Infrastructure und der Programmiersprache C#. Entstanden ist das Mono-Projekt 2001 unter Führung von Miguel de Icaza von der Firma Ximian, die 2003 von Novell aufgekauft wurde. Die Entwickler wurden 2011 in eine neue Firma namens Xamarin übernommen, die im Jahr 2016 eine Microsoft-Tochtergesellschaft wurde. In der Folge wurde Microsoft Hauptsponsor des Projektes.

Mono "hinkt" als Open Source Projekt der eigentlichen Entwicklung etwas nach.

```

.NET Core 3.1
Teile von .NET Framework
Teile von Mono
+ neue Features
-----
.NET 5.0

```

Der Artikel in [heise](#) fasst diesen Status gut zusammen.

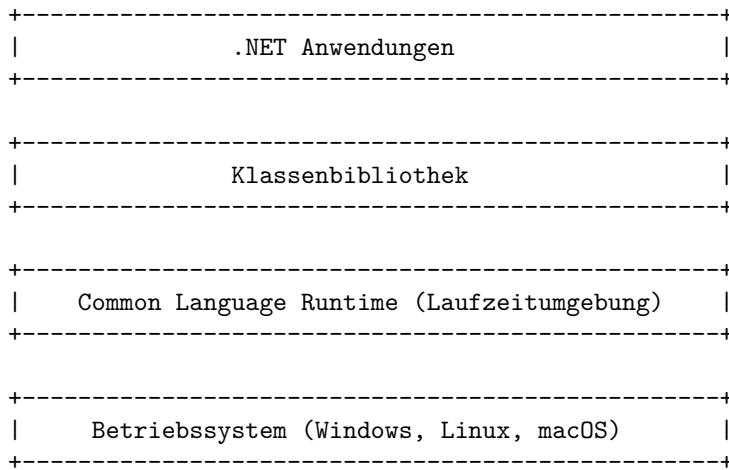
Zum Vergleich sei auf eine Darstellung der **vor dem Erscheinen von .NET 5** verwiesen:

Betriebssystem	.NET Framework	.NET Core	Xamarin
Windows 7/8	X	X	
Windows 10 Desktop	X	X	
Windows 10 Mobile Geräte			X
Linux		X	
macOS		X	
iOS			X

Einen guten Überblick über das historische Nebeneinander gibt das Video von Tim Corey (Achtung, die Darstellung greift den Stand von 2017 auf!) [Link](#)

.NET 6 ist als LTS-Version im November 2021 erscheinen und wird für 3 Jahre unterstützt.

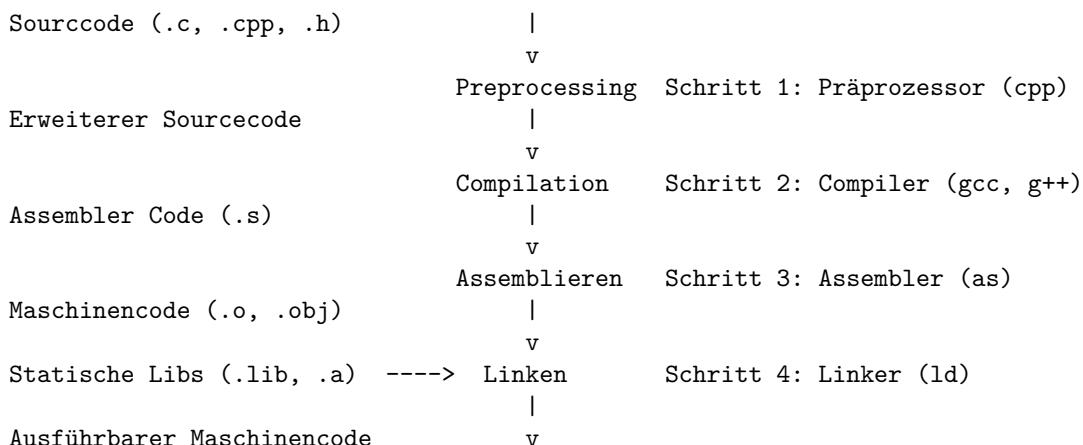
Ziel des .NET-Ökosystems ist die Erhöhung der Anwendungskompatibilität zwischen verschiedenen Systemen und Plattformen. Programme, die das .NET Framework verwenden, werden in der Regel so ausgeliefert, dass benötigte Komponenten des Frameworks automatisch mit installiert werden.



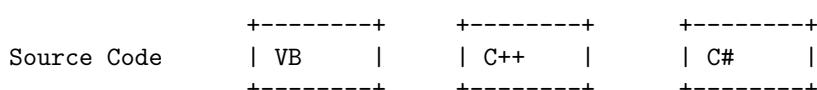
- die Laufzeitumgebung (CLR) implementiert die Ausführungsplattform des .NET Codes. Sie umfasst die Sicherheitsmechanismen, Versionierung, automatische Speicherbereinigung und vor allem die Entkopplung der Programmausführung vom Betriebssystem.
- die Klassenbibliothek gliedern sich intern in Basisklassen und eigenen Bibliotheken für verschiedene Anwendungstypen:
  - ASP.NET ... ist ein Web Application Framework, mit dem sich dynamische Webseiten, Webanwendungen und Webservices entwickeln lassen.
  - Windows Forms/ WPF ... ist ein GUI-Toolkit des Microsoft .NET Frameworks. Es ermöglicht die Erstellung grafischer Benutzeroberflächen (GUIs) für Windows.
  - ...

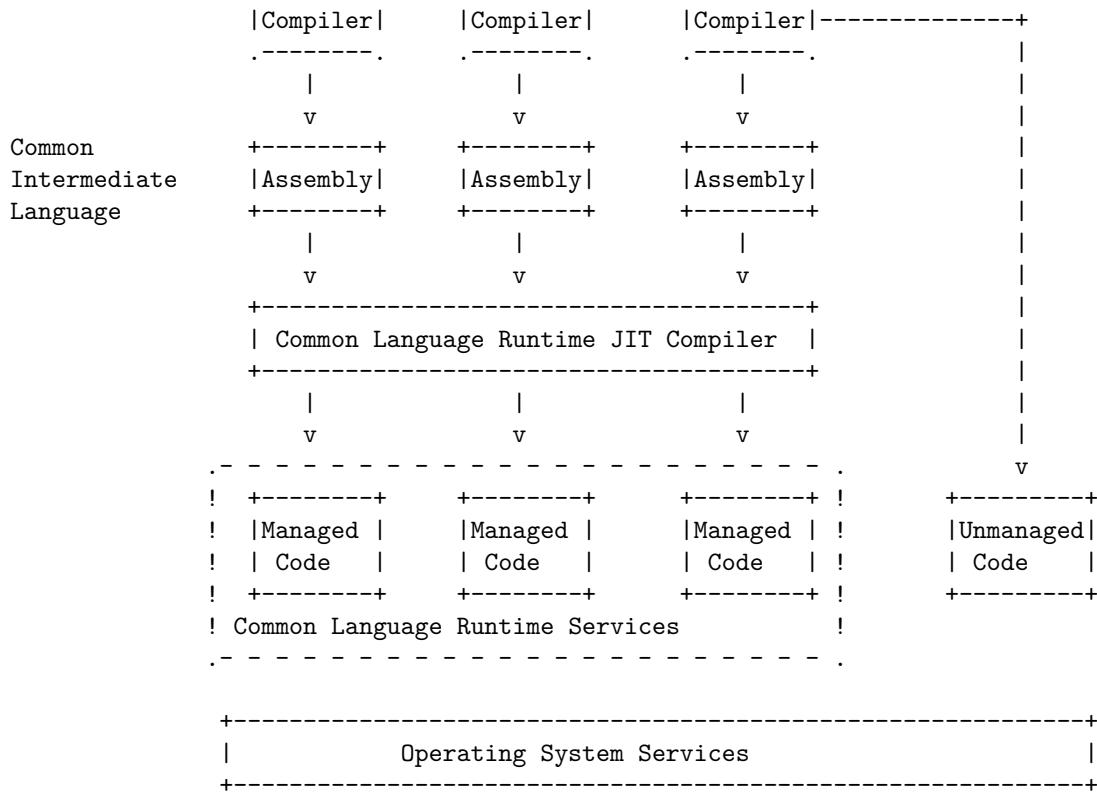
Die Open Source Community stand dem .NET Konzept kritisch gegenüber, da eine “unklare Lage” im Hinblick auf die Lizenzen bestand. Aufgrund der Gefahr durch Patentklagen seitens Microsoft warnte Richard Stallman davor Mono in die Standardkonfiguration von Linuxdistributionen aufzunehme. Ab 2013 änderte Microsoft aber seine Strategie und veröffentlichte den Quellcode von .NET komplett als Open Source unter einer MIT-Lizenz bzw. Apache-2.0-Lizenz.

*Compilierung unter C* (zur Erinnerung und zum Vergleich)



*Build-Prozess mit C#*





Die spezifischen Compiler der einzelnen .NET Sprachen (C#. Visual Basic, F#) bilden den Quellcode auf einen Zwischencode ab. Die Common Language Infrastructure (CLI) ist eine von ISO und ECMA standardisierte offene Spezifikation (technischer Standard), die ausführbaren Code und eine Laufzeitumgebung beschreibt.

*Was passiert unter der Haube der CLR?*

Für die *Managed Code Execution* stellt die CLR ein entsprechendes Set von Komponenten bereit:

- Class Loader ... Einlesen der Assemblies in die CLR Ausführungsumgebung unter Beachtung der Sicherheits-, Versions-, Typinformationen usw.
  - Just-in-Time Compiler ... Abbildung der CIL auf den ausführbaren Maschinencode
  - Code Execution und Debugging
  - Garbage Collection ... der GC ist für die Bereinigung von Referenz-Objekten auf dem Heap verantwortlich und wird von der CLR zu nicht-deterministischen Zeitpunkten gestartet.

```
.assembly HalloWelt { }
.assembly extern mscorelib { }
.method public static void Main() cil managed
{
    .entrypoint
    .maxstack 1
    ldstr "Hallo Welt!"
    call void [mscorelib]System.Console::WriteLine(string)
    ret
}
```

Ein Assembly umfasst: \* das Assemblymanifest, das die Assemblymetadaten enthält. \* die Typmetadaten. \* den CIL-Code \* Links auf mögliche Ressourcen.

Ein Assembly bildet:

- **bildet eine Sicherheitsgrenze** - Eine Assembly ist die Einheit, bei der Berechtigungen angefordert und erteilt werden.
  - **bildet eine Typgrenze** - Die Identität jedes Typs enthält den Namen der Assembly, in der dieser sich befindet. Wenn der Typ MyType in den Gültigkeitsbereich einer Assembly geladen wird, ist dieser nicht dieselbe wie der Typ MyType, der in den Gültigkeitsbereich einer anderen Assembly geladen wurde.
  - **bildet eine Versionsgrenze** - Die Assembly ist die kleinste, in verschiedenen Versionen verwendbare Einheit in der Common Language Runtime. Alle Typen und Ressourcen in derselben Assembly bilden eine Einheit mit derselben Version.

- bildet eine Bereitstellungseinheit** - Beim Starten einer Anwendung müssen nur die von der Anwendung zu Beginn aufgerufenen Assemblies vorhanden sein. Andere Assemblies, z. B. Lokalisierungsressourcen oder Assemblies mit Hilfsklassen, können bei Bedarf abgerufen werden. Dadurch ist die Anwendung beim ersten Herunterladen einfach und schlank.

## Abgrenzung zu Java

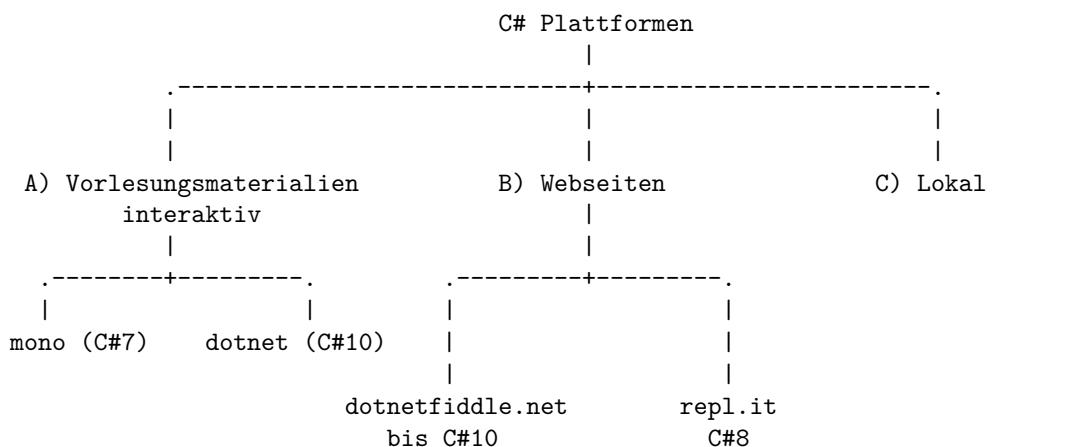
vgl. Vortrag von Mössenböck ([link](#))

	Java	C#
Veröffentlichung	1995	2001
Plattform	(Java) Unix/Linux, Windows, MacOS, Android	(.NET) Windows, Linux, Android, iOS, MacOS
VM	Java-VM	CLR
Zwischencode	Java-Bytecode	CIL
JIT	per Methoden	per Methode / gesamt
Komponenten	Beans	Assemblies
Versionierung	nein	ja
Leitidee	Eine Sprache auf vielen Plattformen	Viele Sprachen auf vielen Plattformen

## Es wird konkret ... Hello World

Die organisatorischen Schlüsselkonzepte in C# sind: **Programme, Namespaces, Typen, Member** und **Assemblies**. C#-Programme bestehen aus mindestens einer Quelldatei, von denen mindestens eine **Main** als einen Methodennamen hat. Programme deklarieren Typen, die Member enthalten, und können in Namespaces organisiert werden.

Wenn C#-Programme kompiliert werden, werden sie physisch in Assemblies verpackt. Assemblies haben unter Windows Betriebssystemen die Erweiterung .exe oder .dll, je nachdem, ob sie Anwendungen oder Bibliotheken implementieren.



### A) Vorlesungsmaterialien - LiaScript Umgebung

Die LiaScript basierte Komplierung und Ausführung kann wie bereits erläutert auf der Basis von mono und dem dotnet Framework umgesetzt werden.

```

using System;

public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, world!");
    }
}
  
```

```
using System;

Console.WriteLine("Hello, world!");

<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

B) Repl.it

<https://replit.com/>

C) .NET Kommandozeile

Das .NET Core Framework kann unter [.NET](#) für verschiedene Betriebssystem heruntergeladen werden. Das SDK umfasst sowohl die Bibliotheken, Laufzeitumgebung und Tools. An dieser Stelle sei nur auf die dotnet Tools verwiesen, die anderen Werkzeuge werden zu einem späteren Zeitpunkt eingeführt.

```
> dotnet new console
> dotnet build
> dotnet run
```

Aus dem Generieren eines neuen Konsolenprojektes ergibt sich ein beeindruckender Baum von Projektdateien.

```
> dotnet new console

> tree
.

|-- bin
   '-- Debug
      '-- net5.0
         '-- ref
            '-- visual_studio_code.dll
         '-- visual_studio_code
         '-- visual_studio_code.deps.json
         '-- visual_studio_code.dll
         '-- visual_studio_code.pdb
         '-- visual_studio_code.runtimeconfig.dev.json
         '-- visual_studio_code.runtimeconfig.json

-- obj
   '-- Debug
      '-- net5.0
         '-- apphost
         '-- ref
            '-- visual_studio_code.dll
         '-- visual_studio_code.AssemblyInfo.cs
         '-- visual_studio_code.AssemblyInfoInputs.cache
         '-- visual_studio_code.assets.cache
         '-- visual_studio_code.csprojAssemblyReference.cache
         '-- visual_studio_code.csproj.CoreCompileInputs.cache
         '-- visual_studio_code.csproj.FileListAbsolute.txt
         '-- visual_studio_code.dll
         '-- visual_studio_code.GeneratedMSBuildEditorConfig.editorconfig
         '-- visual_studio_code.genruntimeconfig.cache
         '-- visual_studio_code.pdb
         '-- project.assets.json
         '-- project.nuget.cache
         '-- visual_studio_code.csproj.nuget.dgspec.json
         '-- visual_studio_code.csproj.nuget.g.props
         '-- visual_studio_code.csproj.nuget.g.targets

-- Program.cs
-- visual_studio_code.csproj
```

*C) .NET Visual Code*

Alternativ können Sie auch die Microsoft Visual Studio oder Visual Code Suite nutzen. Diese kann man zum Beispiel auf unser gerade erstelltes Projekt anwenden

<https://code.visualstudio.com/docs/languages/csharp>

## Aufgaben

- [ ] Installieren Sie das .NET 6 auf Ihrem Rechner und erfreuen Sie sich an einem ersten “Hello World”
- [ ] Testen Sie mit einem Kommilitonen die Features von repl.it! Arbeiten Sie probeweise an einem gemeinsamen Dokument.
- [ ] Legen Sie sich einen GitHub-Account an.



# Chapter 4

## C# Grundlagen I

---

Parameter	Kursinformationen
<b>Veranstaltung:</b>	Vorlesung Softwareentwicklung
<b>Semester:</b>	Sommersemester 2022
<b>Hochschule:</b>	Technische Universität Freiberg
<b>Inhalte:</b>	Einführung in die Basiselemente der Programmiersprache C#
<b>Link auf den GitHub:</b>	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/03_CsharpGrundlagenI.md">https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/03_CsharpGrundlagenI.md</a>
<b>Autoren</b>	@author

---

## Symbolle

Woraus setzt sich ein C# Programm zusammen?

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        // Print Hello World message
        string message = "Glück auf";
        Console.WriteLine(message + " Freiberg");
        Console.WriteLine(message + " Softwareentwickler");
    }
}

using System;

string message = "Glück auf";
Console.WriteLine(message + " Freiberg");
Console.WriteLine(message + " Softwareentwickler");

<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

C# Programme umfassen

- Schlüsselwörter der Sprache,
- Variablennamen,
- Zahlen,
- Zeichen,
- Zeichenketten,
- Kommentare und
- Operatoren.

Leerzeichen, Tabulatorsprünge oder Zeilenenden werden als Trennzeichen zwischen diesen Elementen interpretiert.

```
using System; public class Program {static void Main(string[] args){  
// Print Hello World message  
string message = "Glück auf"; Console.WriteLine(message + " Freiberg");  
Console.WriteLine(message + " Softwareentwickler");}}
```

## Schlüsselwörter

... C# umfasst 77 Schlüsselwörter (C# 7.0), die immer klein geschrieben werden. Schlüsselwörter dürfen nicht als Namen verwendet werden. Ein vorangestelltes @ ermöglicht Ausnahmen.

```
var  
if  
operator  
@class // class als Name !
```

Welche Schlüsselwörter sind das? (C# 7.0)

```
abstract | as | base | bool | break | byte |  
case | catch | char | checked |class | const |  
continue | decimal | default | delegate | do | double |  
else | enum | event | explicit | extern | false |  
finally | fixed | float | for | foreach | goto |  
if | implicit | in | int | interface | internal |  
is | lock | long |namespace | new | null |  
object | operator | out | override | params | private |  
protected | public | readonly | ref | return | sbyte |  
sealed | short | sizeof | stackalloc |static | string |  
struct | switch | this | throw | true | try |  
typeof | uint | ulong | unchecked | unsafe | ushort |  
using | virtual |void | volatile | while | |
```

Auf die Aufführung der 40 kontextabhängigen Schlüsselwörter wie `where` oder `ascending` wurde hier verzichtet.

Ist das viel oder wenig, welche Bedeutung hat die Zahl der Schlüsselwörter?

Sprache	Schlüsselwörter	Bemerkung
F#	98	64 + 8 from ocaml + 26 future
C	42	C89 - 32, C99 - 37,
C++	92	C++11
PHP	49	
Java	51	Java 5.0 (48 without unused keywords const and goto)
JavaScript	38	reserved words + 8 words reserved in strict mode only
Python 3.7	35	
Python 2.7	31	
Smalltalk	6	

Weiterführende Links:

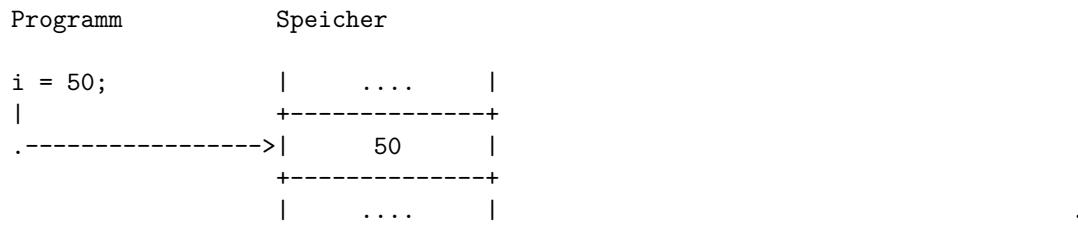
<https://stackoverflow.com/questions/4980766/reserved-keywords-count-by-programming-language>

oder

<https://halyph.com/blog/2016/11/28/prog-lang-reserved-words.html>

## Variablennamen

Variablennamen repräsentieren Speicherbereiche, so dass keine explizite Adressangabe durch den Programmierer zu tätigen ist. Der Compiler „kümmert“ sich um den Rest.



Variablennamen umfassen Buchstaben, Ziffern oder `_`. Das erste Zeichen eines Namens muss ein Buchstabe (des Unicode-Zeichensatzes) oder ein `_` sein. Der C# Compiler ist *case sensitive* (Unterschied zwischen Groß- und Kleinschreibung, z.B. `Test != test`).

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        var Δ = 1;
        Δ++;
        System.Console.WriteLine(Δ);
    }
}
```

Die Vergabe von Namen sollte sich an die Regeln der Klassenbibliothek halten, damit bereits aus dem Namen der Typ ersichtlich wird:

- C#-Community bevorzugt *camel case* `MyNewClass` anstatt *underscoring* `My_new_class`. (Eine engagierte Diskussion zu diesem Thema findet sich unter [Link](#))
- außer bei lokalen Variablen und Parametern oder den Feldern einer Klasse, die nicht von außen sichtbar sind beginnen Namen mit großen Anfangsbuchstaben (diese Konvention wird als *pascal case* bezeichnet)
- Methoden ohne Rückgabewert sollten mit einem Verb beginnen `PrintResult()` alles andere mit einem Substantiv. Boolesche Ausdrücke auch mit einem Adjektiv `valid` oder `empty`.

## Zahlen

Zahlenwerte können als

Format	Variabilität	Beispiel
Ganzzahl	Zahlensystem, Größe, vorzeichenbehaftet/vorzeichenlos	1231, -23423, 0x245
Gleitkommazahl	Größe	234.234234

übergeben werden. Der C# Compiler wertet die Ausdrücke und vergleicht diese mit den vorgesehenen Datentypen. Auf diese wird im Anschluss eingegangen.

Eingabe von Zahlenwerten

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(0xFF);
        Console.WriteLine(0b1111_1111); // ab C#7 unterstützt
        Console.WriteLine(100_000_000);
        Console.WriteLine(1.3454E06);
    }
}
```

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

## Zeichenketten

... analog zu C werden konstante Zeichen mit einfachen Hochkommas 'A', 'b' und Zeichenkettenkonstanten "Bergakademie Freiberg" mit doppelten Hochkommas festgehalten. Es dürfen beliebige Zeichen bis auf die jeweiligen Hochkommas oder das \ als Escape-Zeichen (wenn diese nicht mit dem Escape Zeichen kombiniert sind) eingeschlossen sein.

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Das ist ein ganzer Satz");
        Console.WriteLine('e'); // <- einzelnes Zeichen
        Console.WriteLine("A" == 'A');
    }
}

using System;

public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(@"Das ist ein ganz schön langer
                        Satz, der sich ohne die
                        Zeilenumbrüche blöd lesen
                        würde");
        Console.WriteLine("Das ist ein ganz schön langer \nSatz, der sich ohne die \nZeilenumbrüche blöd
                        Console.WriteLine("Das ist ein ganz schön langer" +
                        "Satz, der sich ohne die" +
                        "Zeilenumbrüche blöd lesen" +
                        "würde");
    }
}
```

## Kommentare

C# unterscheidet zwischen *single-line* und *multi-line* Kommentaren. Diese können mit XML-Tags versehen werden, um die automatische Generierung einer Dokumentation zu unterstützen. Wir werden zu einem späteren Zeitpunkt explizit auf die Kommentierung und Dokumentation von Code eingehen.

Kommentare werden vor der Kompilierung aus dem Quellcode gelöscht.

```
using System;

// <summary> Diese Klasse gibt einen konstanten Wert aus </summary>
public class Program
{
    static void Main(string[] args)
    {
        // Das ist ein Kommentar
        System.Console.WriteLine("Hier passiert irgendwas ...");
        /* Wenn man mal
           etwas mehr Platz
```

```
braucht */
}
}
```

In einer der folgenden Veranstaltungen werden die Möglichkeiten der Dokumentation explizit adressiert.

1. Code gut kommentieren (Zielgruppenorientierte Kommentierung)
2. Header-Kommentare als Einstiegspunkt
3. Gute Namensgebung für Variablen und Methoden
4. Community- und Sprach-Standards beachten
5. Dokumentationen schreiben
6. Dokumentation des Entwicklungsflusses

**Merke:** Machen Sie sich auch in Ihren Programmcodes kurze Notizen, diese sind hilfreich, um bereits gelöste Fragestellungen (in der Prüfungsvorbereitung) nachvollziehen zu können.

## Datentypen und Operatoren

**Frage:** Warum nutzen einige Programmiersprachen eine Typisierung, andere nicht?

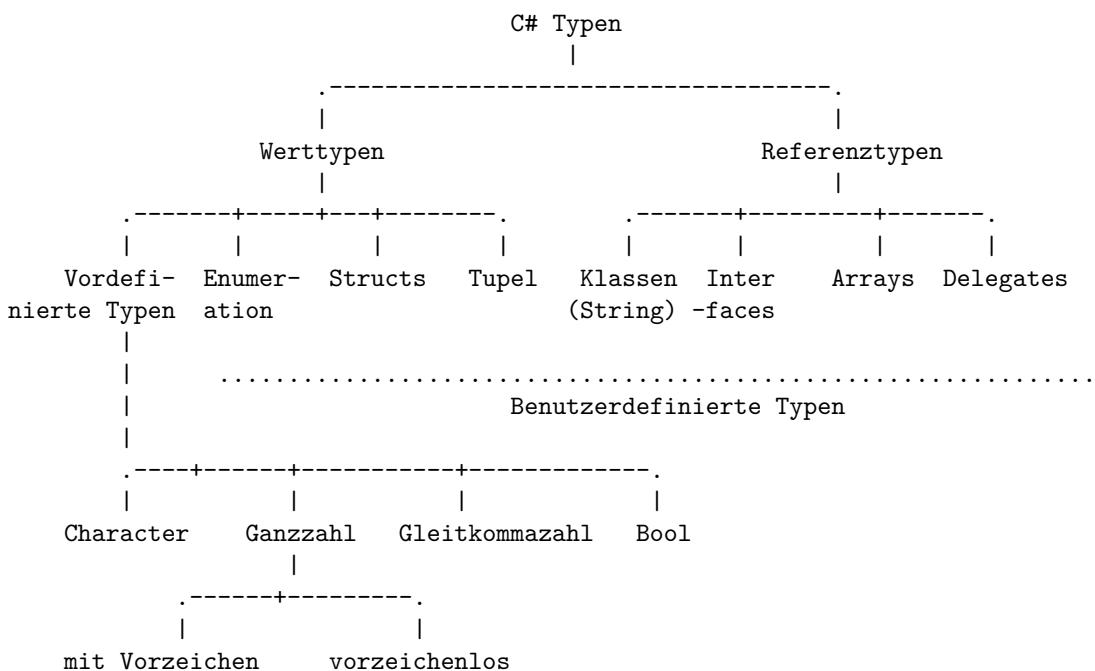
```
number = 5
my_list = list(range(0,10))

print(number)
print(my_list)

#number = "Tralla Trulla"
#print(number)
```

**Merke:** Datentypen definieren unter anderem den möglichen “Inhalt”, Speichermechanismen (Größe, Organisation).

Datentypen können sehr unterschiedlich strukturiert werden. Das nachfolgende Schaubild realisiert dies auf 2 Ebenen (nach Mössenböck, Kompaktkurs C# 7 )



Die Zuordnung zu Wert- und Referenzdatentypen ergibt sich dabei aus den zwei grundlegenden Organisationsformen im Arbeitsspeicher.

	Werttypen	Referenztypen
Variable enthält	einen Wert	eine Referenz
Speicherort	Stack	Heap
Zuweisung	kopiert den Wert	kopiert die Referenz

	Werttypen	Referenztypen
Speicher	Größe der Daten	Größe der Daten, Objekt-Metadata, Referenz

## Wertdatentypen

Im Folgenden werden die Werttypen und deren Operatoren besprochen, bevor in der nächsten Veranstaltung auf die Referenztypen konzeptionell eingegangen wird.

### Character Datentypen

Der `char` Datentyp repräsentiert Unicode Zeichen (vgl. [Link](#)) mit einer Breite von 2 Byte.

```
char oneChar = 'A';
char secondChar = '\n';
char thirdChar = (char) 65; // Referenz auf ASCII Tabelle
```

Die Eingabe erfolgt entsprechend den Konzepten von C mit einfachen Anführungszeichen. Doppelte Anführungsstriche implizieren `String`-Variablen!

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        var myChar = 'A';
        var myString = "A";
        Console.WriteLine(myChar.GetType());
        Console.WriteLine(myString.GetType());
    }
}
```

Neben der unmittelbaren Eingabe über die Buchstaben und Zeichen kann die Eingabe entsprechend

- einer Escapesequenz für Unicodezeichen, d. h. \u gefolgt von der aus vier(!) Symbolen bestehenden Hexadezimaldarstellung eines Zeichencodes.
- einer Escapesequenz für Hexadezimalzahlen, d. h. \x gefolgt von der Hexadezimaldarstellung eines Zeichencodes.

erfolgen.

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine('\u2328' + " Unicodeblock Miscellaneous Technical");
        Console.WriteLine('\u2FOC' + " Unicodeblock Kangxi Radicals");
    }
}
```

Entsprechend der Datenbreite können `char` Variablen implizit in `short` überführt werden. Für andere numerische Typen ist eine explizite Konvertierung notwendig.

### Zahlendatentypen und Operatoren

Type	Suffix	Name	.NET Typ	Bits	Wertebereich
Ganzzahl vorzeichenbehaftet		sbyte	SByte	8	-128 bis 127
		short	Int16	16	-32.768 bis 32.767
		int	Int32	32	-2.147.483.648 bis 2.147.483.647
	L	long	Int64	64	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807

Type	Suffix	Name	.NET Typ	Bits	Wertebereich
Ganzzahl ohne Vorzeichen		byte	Byte	8	0 bis 255
		ushort	UInt16	16	0 bis 65.535
	U	uint	UInt32	32	0 bis 4.294.967.295
	UL	ulong	UInt64	64	0 bis 18.446.744.073.709.551.615
Gleitkommazahl	F	float	Single	32	
	D	double	Double	64	
	M	decimal	Decimal	128	

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        int i = 5;
        Console.WriteLine(i.GetType());
        Console.WriteLine(int.MinValue);
        Console.WriteLine(int.MaxValue);
    }
}
```

### Numerische Suffixe

Suffix	C# Typ	Beispiel	Bemerkung
F	float	float f = 1.0F	
D	double	double d = 1D	
M	decimal	decimal d = 1.0M	Compilerfehler bei Fehlen des Suffix
U	uint	uint i = 1U	

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        float f = 5.1F;
        Console.WriteLine(f.GetType());
    }
}
```

### Exkurs: Gleitkommazahlen

**Frage:** Gleitkommazahlen, wie funktioniert das eigentlich und wie lässt sich das Format auf den Speicher abbilden?

Ein naheliegender und direkt zu Gleitkommazahlen führender Gedanke ist der Ansatz neben dem Zahlenwert auch die Position des Kommas abzuspeichern. In der “ingenieurwissenschaftlichen Schreibweise” ist diese Information aber an zwei Stellen verborgen, zum einen im Zahlenwert und zum anderen im Exponenten.

**Beispiel:** Der Wert der *Lichtgeschwindigkeit* beträgt

$$\begin{aligned}
 c &= 299\,792\,458 \text{ m/s} \\
 &= 299\,792,458 \cdot 10^3 \text{ m/s} \\
 &= 0,299\,792\,458 \cdot 10^9 \text{ m/s} \\
 &= 2,997\,924\,58 \cdot 10^8 \text{ m/s}
 \end{aligned}$$

Um diese zusätzliche Information eindeutig abzulegen, normieren wir die Darstellung - die Mantisse wird in einen festgelegten Wertebereich, zum Beispiel  $1 \leq m < 10$  gebracht.

Die Gleitkommadarstellung besteht dann aus dem Vorzeichen, der Mantisse und dem Exponenten. Für binäre Zahlen ist diese Darstellung in der [IEEE 754](#) genormt.

```
+----+ ~ -----+-----+ ~ -----+
|V|  Mantisse   |  Exponent   |  V=Vorzeichenbit
+----+ ~ -----+-----+ ~ -----+
1      23          8           = 32 Bit (float)
1      52          11          = 64 Bit (double)
```

Welche Probleme treten bei der Verwendung von `float`, `double` und `decimal` ggf. auf?

### Rundungsfehler

Ungenauie Darstellungen bei der Zahlenrepräsentation führen zu:

- algebraisch inkorrektne Ergebnissen
- fehlender Gleichheit bei Konvertierungen in der Verarbeitungskette
- Fehler beim Test auf Gleichheit

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        double fnumber = 123456784649577.0;
        double additional = 0.0000001;
        Console.WriteLine("Experiment 1");
        Console.WriteLine("{0} + {1} = {2:G17}", fnumber, additional,
                           fnumber + additional);
        Console.WriteLine(fnumber == (fnumber + additional));
    }
}

using System;

public class Program
{
    static void Main(string[] args)
    {
        double value = .1;
        double result = 0;
        for (int ctr = 1; ctr <= 10000; ctr++){
            result += value;
        }
        Console.WriteLine("Experiment 2");
        Console.WriteLine(".1 Added 10000 times: {0:G17}", result);
    }
}
```

### Dezimal-Trennzeichen

Im Beispielprogramm wird ein Dezimalpunkt als Trennzeichen verwendet. Diese Darstellung ist jedoch kulturspezifisch. In Deutschland gelten das Komma als Dezimaltrennzeichen und der Punkt als Tauschender-Trennzeichen. Speziell bei Ein- und Ausgaben kann das zu Irritationen führen. Diese können durch die Verwendung der Klasse `System.Globalization.CultureInfo` beseitigt werden.

Zum Beispiel wird mit der folgenden Anweisung die Eingabe eines Dezimalpunkts statt Dezimalkomma erlaubt.

```
double wert = double.Parse(Console.ReadLine(), System.Globalization.CultureInfo.InvariantCulture);
```

### Division durch Null

Die Datentypen `float` und `double` kennen die Werte *NegativeInfinity* (`-1.#INF`) und *PositiveInfinity* (`1.#INF`), die bei Division durch Null entstehen können. Außerdem gibt es den Wert *NaN* (*not a number*, `1.#IND`), der

einen irregulären Zustand repräsentiert. Mit Hilfe der Methoden *IsInfinity()* bzw. *IsNaN()* kann überprüft werden, ob diese Werte vorliegen.

```
Console.WriteLine(Double.NaN(0.0/0.0)); //gibt true aus
```

## Numerische Konvertierungen

Konvertierungen beschreiben den Transformationsvorgang von einem Zahlentyp in einen anderen. Im Beispiel zuvor provoziert die Zeile

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        float f = 5.1D;
    }
}
```

eine Fehlermeldung

```
... error CS0664:
Literal of type double cannot be implicitly converted
to type `float'. Add suffix `f' to create a literal of
this type.
...
```

Das Problem ist offensichtlich. Wir versuchen einen Datentypen, der größere Werte umfassen kann auf einen Typen mit einem kleineren darstellbaren Zahlenbereich abzubilden. Der Compiler unterbindet dies logischerweise.

C# kennt implizite und explizite Konvertierungen.

```
int x = 1234;
long y = x;
short z = (short) x;
```

Da die Konvertierung von Ganzkommazahlen in Gleitkommazahlen in jedem Fall umgesetzt werden kann, sieht C# hier eine implizite Konvertierung vor. Umgekehrt muss diese explizit realisiert werden.

Explizite Konvertierung mit dem Typkonvertierungsoperator (runde Klammern) ist ebenfalls nicht immer möglich. Zusätzliche Möglichkeiten der Typkonvertierung bietet für elementare Datentypen die Klasse **Convert** durch zahlreiche Methoden wie z.B.:

```
int wert=Convert.ToInt32(Console.ReadLine()); //string to int
```

## Arithmetische Operatoren

### Alle Numerischen Datentypen

Die arithmetischen Operatoren +, -, \*, /, % sind für alle numerischen Datentypen die bekannten Operationen Addition, Subtraktion, Multiplikation, Division und Modulo, mit Ausnahme der 8 und 16-Bit breiten Typen (byte und short). Diese werden vorher implizit zu einem **int** konvertiert und dann wird die bekannte Operation durchgeführt (Siehe Folie 2/2).

Die Addition und Subtraktion kann mit Inkrement und Dekrement-Operatoren abgebildet werden.

```
using System;
```

```
public class Program
{
    static void Main(string[] args)
    {
        int result = 101;
        for (int i = 0; i<100; i++){ // Anwendung des Inkrement Operators
            result--; // Anwendung des Dekrement Operators
        }
    }
}
```

```

        Console.WriteLine(result);
    }
}

```

## Integraltypen

Divisionsoperationen generieren einen abgerundeten Wert bei der Anwendung auf Ganzkommazahlen. Fangen sie mögliche Divisionen durch 0 mit entsprechenden Exceptions ab!

- Wechsel zu Floatingpoint Zahlen (über Komma und Suffix),
- Motivation der Format Specifiers von WriteLine
- Division durch 0 ->

```

using System;

public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Division von 2/3 = {0:D}", 2/3);
    }
}

```

Überlausituationen (Vergleiche Ariane 5 Beispiel der zweiten Vorlesung) lassen sich in C# sehr komfortabel handhaben:

- Einführung des checked Operators ->

```

using System;

public class Program
{
    static void Main(string[] args)
    {
        int a = int.MinValue;
        Console.WriteLine("Wert von a = {0}", a);
        a--;
        Console.WriteLine("Wert von a nach Dekrement = {0}", a);
    }
}

```

Die Überprüfung kann auf Blöcke `checked{}` ausgedehnt werden oder per Compiler-Flag den gesamten Code einbeziehen. Der `checked` Operator kann nicht zur Analyse von Operationen mit Gleitkommazahlen herangezogen werden!

## 8 und 16-Bit Integraltypen

Diese Typen haben keine “eigenen” Operatoren. Vielmehr konvertiert der Compiler diese implizit, was bei der Abbildung auf den kleineren Datentyp zu entsprechenden Fehlermeldungen führt.

- \* Generierung Kompilerfehler
- \* Ergänzung cast Operator

->

```

using System;

public class Program
{
    static void Main(string[] args)
    {
        short x = 1, y = 1;
        short z = x + y;
        Console.WriteLine("Die Summe ist gleich {0:D}", z);
    }
}

```

## Bitweise Operatoren

Bitweise Operatoren verknüpfen Zahlen auf der Ebene einzelnen Bits, analog anderen Programmiersprachen stellt C# folgende Operatoren zur Verfügung:

Symbol	Wirkung
<code>~</code>	invertiert jedes Bit
<code> </code>	verknüpft korrespondierende Bits mit ODER
<code>&amp;</code>	verknüpft korrespondierende Bits mit UND
<code>^</code>	verknüpft korrespondierende Bits mit XOR
<code>&lt;&lt;</code>	bitweise Verschiebung nach links
<code>&gt;&gt;</code>	bitweise Verschiebung nach rechts

```
using System;

public class Program
{
    public static string printBinary(int value)
    {
        return Convert.ToString(value, 2).PadLeft(8, '0');
    }

    static void Main(string[] args)
    {
        int x = 21, y = 12;
        Console.WriteLine(printBinary(7));
        Console.WriteLine("dezimal:{0:D}, binär:{1}", x, printBinary(x));
        Console.WriteLine("dezimal:{0:D}, binär:{1}", y, printBinary(y));
        Console.WriteLine("x & y = {0}", printBinary(x & y));
        Console.WriteLine("x | y = {0}", printBinary(x | y));
        Console.WriteLine("x << 1 = {0}", printBinary(x << 1));
        Console.WriteLine("x >> 1 = {0}", printBinary(x >> 1));
    }
}
```

## Aufgabe

- [ ] Machen Sie sich noch mal mit dem Ariane 5 Desaster vertraut. Wie hätte eine C# Lösung ausgesehen, die den Absturz verhindert hätte?
- [ ] Experimentieren Sie mit den Datentypen. Vollziehen Sie dabei die Erläuterungen des nachfolgenden Videos nach:

!?alt-text



# Chapter 5

## C# Grundlagen II

---

Parameter Kursinformationen

Veranstaltung Vorlesung Softwareentwicklung

Semester Sommersemester 2022

Hochschule: Technische Universität Freiberg

Inhalte: Einführung in die Basiselemente der Programmiersprache C#

Link auf [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/04\\_CsharpGrundlagenII.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/04_CsharpGrundlagenII.md)

GitHub:

Autoren @author

---

### Auf Nachfrage ...

Was passiert, wenn man eine größere Zahl in eine kleinere konvertiert, so dass offensichtlich Stellen verloren gehen?

Type	Name	Bits	Wertebereich
Ganzzahl ohne Vorzeichen	byte	8	0 bis 255
	ushort	16	0 bis 65.535
	uint	32	0 bis 4.294.967.295
	ulong	64	0 bis 18.446.744.073.709.551.615

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        byte x = 0;
        ushort y = 65535;
        Console.WriteLine(x);
        Console.WriteLine(y);

        x = y; // Fehler! Die Konvertierung muss explizit erfolgen!
    }
}
```

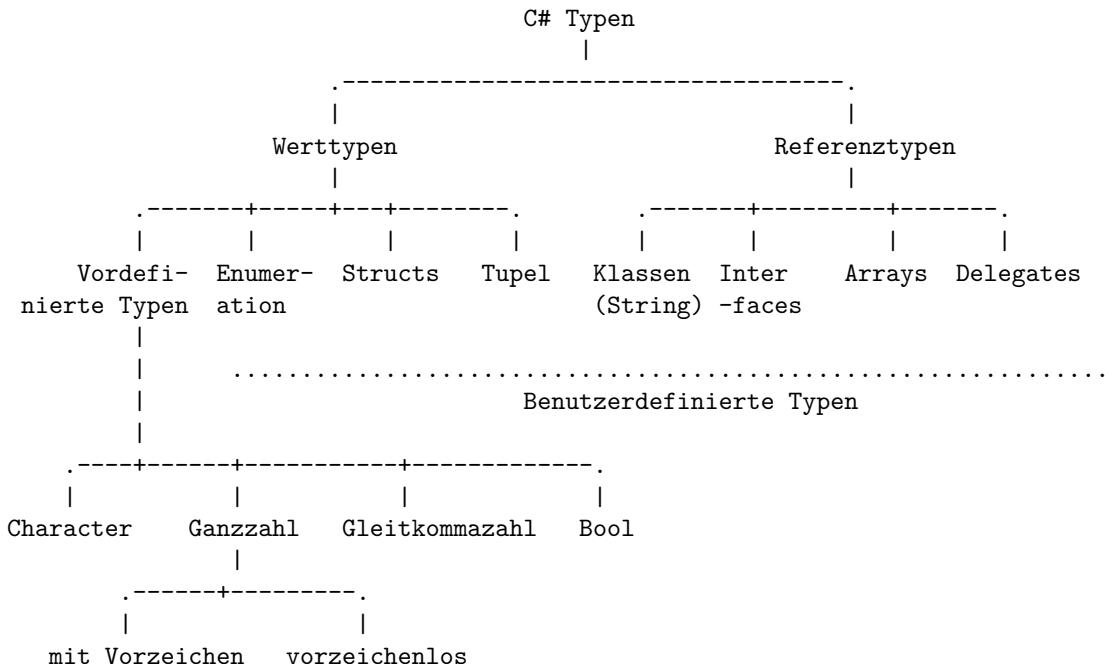
Wert	Binäre Darstellung
65.535	11111111 11111111

Wert	Binäre Darstellung
255	11111111

Nutzen Sie `checked{ }`, um eine Überprüfung der Konvertierung zur Laufzeit vornehmen zu lassen Link auf die Dokumentation.

## Wertdatentypen und Operatoren (Fortsetzung)

Aufbauend auf den Inhalten der Vorlesung 3 setzen wir unseren Weg durch die Datentypen und Operatoren unter C# fort.



## Boolscher Datentyp und Operatoren

In anderen Sprachen kann die `bool` Variable (logischen Werte `true` and `false`) mit äquivalent Zahlenwerten kombiniert werden.

```

x = True
y = 1

print(y==True)

In C# existieren keine impliziten cast-Operatoren, die numerische Werte und umgekehrt wandeln!
using System;

public class Program
{
    static void Main(string[] args)
    {
        bool x = true;
        Console.WriteLine(x);
        Console.WriteLine(!x);
        Console.WriteLine(x == true);      // Rückgabe eines "neuen" bool Wertes
        int y = 1;
        //Console.WriteLine(x == y);       // Funktioniert nicht
        // Lösungsansatz I bool -> int
        int bool2int = x ? 1 : 0;
        Console.WriteLine(bool2int);
        // Lösungsansatz II
  
```

```

        bool2int = Convert.ToInt32(x);
        Console.WriteLine(bool2int);
        Console.WriteLine(bool2int == y); // Funktioniert
    }
}

```

Im Codebeispiel wird der sogenannte tertiäre Operator ? verwandt, der auch durch eine if Anweisung abgebildet werden könnte (vgl. [Dokumentation](#)).

Welchen Vorteil/Nachteil sehen Sie zwischen den beiden Lösungsansätzen?

Die Vergleichsoperatoren == und != testen auf Gleichheit oder Ungleichheit für jeden Typ und geben in jedem Fall einen bool Wert zurück. Dabei muss unterschieden werden zwischen Referenztypen und Wertetypen.

\* Einführung eines weiteren Objektes, dass auf student2 zeigt,  
anschließend Ausführung der Vergleichsoperation

->

```

using System;

public class Person{
    public string Name;
    public Person (string n) {Name = n;}
}

public class Program
{
    static void Main(string[] args)
    {
        Person student1 = new Person("Sebastian");
        Person student2 = new Person("Sebastian");
        Console.WriteLine(student1 == student2);
    }
}

```

Merke: Für Referenztypen evaluiert == die Addressen der Objekte, für Wertetypen die spezifischen Daten. (Es sei denn, Sie haben den Operator überladen.)

Die Gleichheits- und Vergleichsoperationen ==, !=, >=, > usw. sind auf alle numerischen Typen anwendbar.

In der Vorlesung 3 war bereits über die bitweisen boolschen Operatoren gesprochen worden. Diese verknüpfen Zahlenwerte auf Bitniveau. Die gleiche Notation (einzelne Operatorsymbole &, |) kann auch zur Verknüpfung von Boolschen Aussagen genutzt werden.

Darüber hinaus existieren die doppelten Schreibweisen als eigenständige Operatorkonstrukte - &&, , ||. Bei der Anwendung auf boolsche Variablen wird dabei zwischen "nicht-konditionalen" und "konditionalen" Operatoren unterschieden.

Bedeutung der boolschen Operatoren für unterschiedliche Datentypen:

Operation	numerische Typen	boolsche Variablen
&	bitweises UND (Ergebnis ist ein numerischer Wert!)	nicht-konditionaler UND Operator
&&	FEHLER	konditionaler UND Operator

\* Wechsel zu && -> Fehlermeldung

->

```

using System;

public class Program
{
    static void Main(string[] args)
    {
        int a = 6; // 0110
    }
}

```

```

        int b = 10; // 1010
        Console.WriteLine((a & b).GetType());
        Console.WriteLine(Convert.ToString(a & b, 2).PadLeft(8,'0'));
        // Console.WriteLine(a && b);
    }
}

```

Konditional und Nicht-Konditional, was heißt das? Erstgenannte optimieren die Auswertung. So berücksichtigt der AND-Operator `&&` den rechten Operanden gar nicht, wenn der linke Operand bereits ein `false` ergibt.

```

bool a=true, b=true, c=false;
Console.WriteLine(a || (b && c)); // short-circuit evaluation

// alternativ
Console.WriteLine(a | (b & c)); // keine short-circuit evaluation

```

Hier ein kleines Beispiel für die Optimierung der Konditionalen Operatoren:

```

using System;

public class Program
{
    public static void Main(){

        bool a=false, b= true, c=false;

        //Nicht-Konditionales UND
        DateTime start = DateTime.Now;
        for(int i=0; i<1000; i++){
            if(a & (b | c)){}
        }
        DateTime end = DateTime.Now;
        Console.WriteLine("Mit Nicht-Konditionalen Operatoren dauerte es: {0} Millisekunden", (end - start).TotalMilliseconds);

        //Konditionales UND
        start = DateTime.Now;
        for(int i=0; i<1000; i++){
            if(a && (b || c)){}
        }
        end = DateTime.Now;
        Console.WriteLine("Mit Konditionalen Operatoren dauerte es nur: {0} Millisekunden, da vereinfacht wurde", (end - start).TotalMilliseconds);
    }
}

```

## Enumerations

Enumerationstypen erlauben die Auswahl aus einer Aufstellung von Konstanten, die als Enumeratorliste bezeichnet wird. Was passiert intern? Die Konstanten werden auf einen ganzzahligen Typ gemappt. Der Standardtyp von Enumerationselementen ist `int`. Um eine Enumeration eines anderen ganzzahligen Typs, z. B. `byte` zu deklarieren, setzen Sie einen Doppelpunkt hinter dem Bezeichner, auf den der Typ folgt.

- Darstellung des Enum spezifischen Cast Operators Day startingDay = (Day) 5;
- Darstellung der Möglichkeit Constanten zuzuordnen Sat = 5 ->

```

using System;

public class Program
{
    enum Day {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
    //enum Day : byte {Sat, Sun, Mon, Tue, Wed, Thu, Fri};

    static void Main(string[] args)
    {

```

```

        Day startingDay = Day.Wed;
        Console.WriteLine(startingDay);
    }
}

```

Die Typkonvertierung von einem Zahlenwert in eine enum kann wiederum mit `checked` überwacht werden.

Dabei schließen sich die Instanzen nicht gegenseitig aus, mit einem entsprechenden Attribut können wir auch Mehrfachbelegungen realisieren.

- Hinweis auf Zahlenzuordnung mit Zweierpotenzen ->

```
// https://docs.microsoft.com/de-de/dotnet/api/system.flagsattribute?view=netframework-4.7.2

using System;

public class Program
{
    [FlagsAttribute] // <- Spezifisches Enum Attribut
    enum MultiHue : byte
    {
        None   = 0b_0000_0000, // 0
        Black  = 0b_0000_0001, // 1
        Red    = 0b_0000_0010, // 2
        Green  = 0b_0000_0100, // 4
        Blue   = 0b_0000_1000, // 8
    };

    static void Main(string[] args)
    {
        Console.WriteLine(
            "\nAll possible combinations of values with FlagsAttribute:");
        for( int val = 0; val < 16; val++ )
            Console.WriteLine( "{0,3} - {1}", val, (MultiHue)val );
    }
}

<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>
</Project>
```

## Weitere Wertdatentypen

Für die Einführung der weiteren Wertdatentypen müssen wir noch einige Grundlagen erarbeiten. Entsprechend wird an dieser Stelle noch nicht auf `struct` und `tupel` eingegangen. Vielmehr sei dazu auf nachfolgende Vorlesungen verwiesen.

## Referenzdatentypen

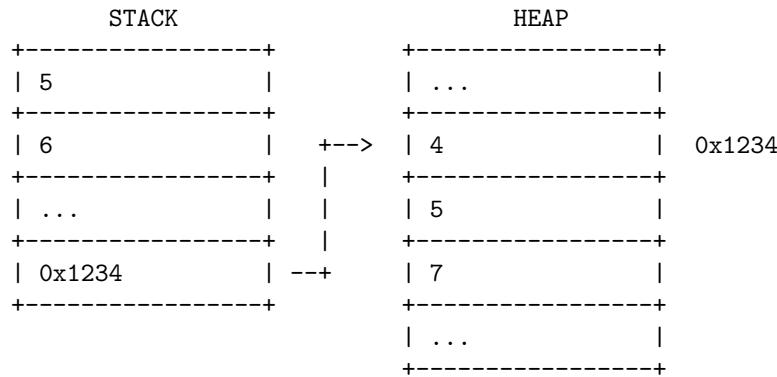
In der vergangenen Veranstaltung haben wir bereits über die Trennlinie zwischen Werttypen und Referenztypen gesprochen. Was bedeutet die Idee aber grundsätzlich?

Aspekt	Stack	Heap
Format	Es ist ein Array des Speichers. Es ist eine LIFO (Last In First Out) Datenstruktur. In ihr können Daten nur von oben hinzugefügt und gelöscht werden.	Es ist ein Speicherbereich, in dem Chunks zum Speichern bestimmter Arten von Datenobjekten zugewiesen werden. In ihm können Daten in beliebiger Reihenfolge gespeichert und entfernt werden.
Was wird abgelegt?	Wertedatentypen	Referenzdatentypen

Aspekt	Stack	Heap
Was wird auf dem Stack gespeichert?	Wert	Referenz
Kann die Größe variieren werden?	nein	ja
Zugriffsgeschwindigkeit		gering
Freigabe vom Compiler organisiert		vom Garbage Collector realisiert

### Wie werden Objekte auf dem Stack/Heap angelegt?

```
int x = 5;
int y = 6;
int[] array = new int[] { 4, 5, 7};
```

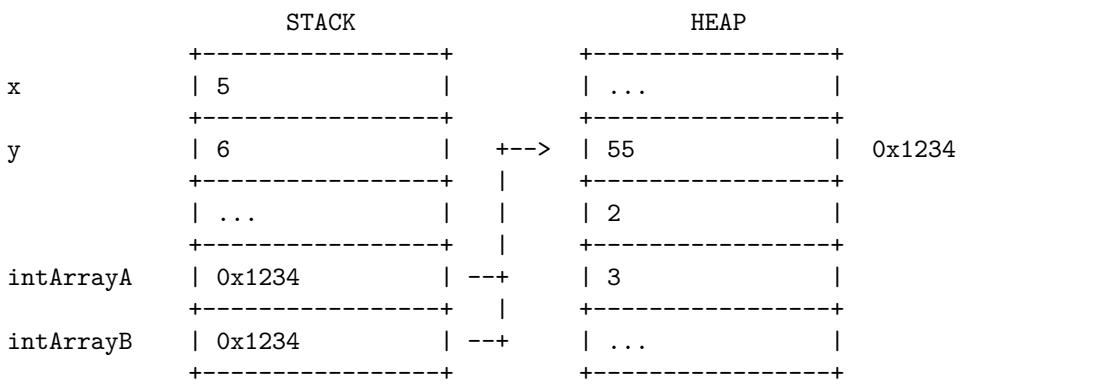


### Und was bedeutet dieser Unterschied?

Ein zentrales Element ist die unterschiedliche Wirkung des Zuweisungsoperators `=`. Analoges gilt für den Vergleichsoperator `==` den wir bereits betrachtet haben.

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        // Zuweisung für Wertetypen
        int x = 5;
        int y = 6;
        y = x;
        Console.WriteLine("{0}, {1}", x, y);
        // Zuweisung für Referenztypen
        int [] intArrayA = new int[]{1,2,3};
        int [] intArrayB = intArrayA;
        Console.WriteLine("Alter Status {0}", intArrayB[0]);
        intArrayA[0] = 55;
        Console.WriteLine("Neuer Status {0}", intArrayA[0]);
        Console.WriteLine("Neuer Status {0}", intArrayB[0]);
        // Und wenn wir beides vermischen?
        intArrayA[1] = x;
        Console.WriteLine("Neuer Status {0}", intArrayA[1]);
    }
}
```



Muss die Referenz immer auf ein Objekt auf dem Heap zeigen?

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        int [] intArrayA = new int[]{1,2,3};
        int [] intArrayB;
        // int [] intArrayB = null;
        //if (intArrayB != null){      // C#6 Syntax
        if (intArrayB is not null){   // C#9 Syntax
            Console.WriteLine("Alles ok, mit intArrayB");
        }
    }
}

<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>
</Project>
```

Mit `null` kann angezeigt werden, dass diese Referenz noch nicht zugeordnet wurde.

## Array Datentyp

Arrays sind potentiell multidimensionale Container beliebiger Daten, also auch von Arrays und haben folgende Eigenschaften:

- Ein Array kann eindimensional, mehrdimensional oder verzweigt sein.
- Die Größe innerhalb der Dimensionen eines Arrays wird festgelegt, wenn die Arrayinstanz erstellt wird. Eine Anpassung zur Lebensdauer ist nicht vorgesehen.
- Arrays sind nullbasiert: Der Index eines Arrays mit n Elementen beginnt bei 0 und endet bei n-1.
- Arraytypen sind Referenztypen.
- Arrays können mit `foreach` iteriert werden.

**Merke:** In C# sind Arrays tatsächlich Objekte und nicht nur adressierbare Regionen zusammenhängender Speicher wie in C und C++.

### Eindimensionale Arrays

Eindimensionale Arrays werden über das Format

```
<typ>[] name = new <typ>[<anzahl>];
```

deklariert.

Die spezifische Größenangabe kann entfallen, wenn mit der Deklaration auch die Initialisierung erfolgt.

```
<typ>[] name = new <typ>[] {<eintrag_0>, <eintrag_1>, <eintrag_2>};
```

- Statische Beschränkung der Loop! Fehler generieren
- Ersetzen durch intArray.Length
- Wie kann man nach mehreren Zeichen splitten? ->

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        int [] intArray = new int [5];
        short [] shortArray = new short[] { 1, 3, 5, 7, 9 };
        for (int i = 0; i < 3; i++){
            Console.WriteLine("{0, 3}", intArray[i]);
        }
        Console.WriteLine("");
        string sentence = "Das ist eine Sammlung von Worten";
        string [] stringArray = sentence.Split();
        foreach(string i in stringArray){
            Console.WriteLine("{0, -9}", i);
        }
    }
}

using System;

public class Program
{
    static void Main(string[] args)
    {
        int [] intArray = new int [5];
        short [] shortArray = new short[] { 1, 3, 5, 7, 9 };
        for (int i = 0; i < 3; i++){
            Console.WriteLine("{0, 3}", intArray[i]);
        }
        Console.WriteLine("");
        string sentence = "Das ist eine Sammlung von Worten";
        string [] stringArray = sentence.Split();
        foreach(string i in stringArray){
            Console.WriteLine("{0, -9}", i);
        }
    }
}
```

## Mehrdimensionale Arrays

C# unterscheidet zwei Typen mehrdimensionaler Arrays, die sich bei der Initialisierung und Indizierung unterschiedlich verhalten.

### Rechteckige Arrays

```
+-----+-----+-----+
a[zeile, Spalte] -->| [0,0] | [0,1] | [0,2] | [0,3] |
+-----+-----+-----+
| [1,0] | [1,1] | [1,2] | [1,3] |
+-----+-----+-----+
```

### Ausgefranste Arrays

```
+---+ +-----+-----+-----+
a[index] --> | [0] | --> | [0], [0] | [0], [1] | [0], [2] | [0], [3] |
+---+ +-----+-----+-----+
| [1] | | [1], [0] | [1], [1] |
+---+ +-----+-----+
```

```

| [2] |      | [2], [0] | [2], [1] | [2], [2] |
+---+      +-----+-----+-----+
int[,] rectangularMatrix = //entspricht int[3,3]
{
    {1,2,3},
    {0,1,2},
    {0,0,1}
};

int [][] jaggedMatrix ={ //entspricht int[3][]
    new int[] {1,2,3},
    new int[] {0,1,2},
    new int[] {0,0,1}
};

```

## String Datentyp

Als Referenztyp verweisen `string` Instanzen auf Folgen von Unicodezeichen, die durch ein Null \0 abgeschlossen sind. Bei der Interpretation der Steuerzeichen muss hinterfragt werden, ob eine Ausgabe des Zeichens oder eine Realisierung der Steuerzeichenbedeutung gewünscht ist.

```

using System;

public class Program
{
    static void Main(string[] args)
    {
        string text1 = "Das ist ein \n Test der \t über mehrere Zeilen geht!";
        string text2 = @"Das ist ein
Test der
über mehrere Zeilen geht!";
        Console.WriteLine(text1);
        Console.WriteLine(text2);
    }
}

```

Der Additionsoperator steht für 2 `string` Variablen bzw. 1 `string` und eine andere Variable als Verknüpfungsoperator (sofern für den zweiten Operanden die Methode `toString()` implementiert ist) bereit.

- Integration einer `ToString` Methode ->

```

using System;

public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("String + String = " + "StringString" );
        Console.WriteLine("String + Zahl 5 = " + 5); // Implizites .ToString()
    }
}

```

Der Gebrauch des + Operators im Zusammenhang mit `string` Daten ist nicht effektiv. eine bessere Performance bietet `System.Text.StringBuilder`.

In der nächsten Vorlesung werden wir uns explizit mit den Konzepten der Ausgabe und entsprechend den Methoden der String Generierung beschäftigen.

## Umgang mit Variablen

Wie sollten wir die variablen benennbaren Komponenten unseres Programms bezeichnen [Naming guidelines?](#)

- Nutzen Sie sinnvolle, selbsterklärende Variablennamen!

- Vermeiden Sie Abkürzungen abgesehen von verbreiteten Bezeichnungen.
- camelCasing für Methodenargumente und lokale Variablen um konsistent mit dem .NET Framework zu sein

```
public class UserLog
{
    public void Add(LogEvent logEvent)
    {
        int itemCount = 0;
        // ...
    }
}
```

- Keine Codierung von Datentypen (Ungarische Notation), ihre IDE sollte hinreichend schlau sein.

```
// Correct
int counter;
string name;
// Avoid
int iCounter;
string strName;
```

- Vermeiden Sie es Konstanten mit Screaming Caps zu definieren (diskutable Position)

```
// Correct
public const string UniName = "TU Freiberg";
// Avoid
public const string UNINAME = "TU Freiberg";
```

Ihre IDE bzw. ein Linterprogramm sollte die Einhaltung dieser Regularien überprüfen.

## Konstante Werte

Konstanten sind unveränderliche Werte, die zur Compilezeit bekannt sind und sich während der Lebensdauer des Programms nicht ändern. Der Versuch einer Änderung wird durch den Compiler überwacht.

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        const double pi = 3.14;
        pi = 5; //erzeugt Fehlermeldung, da pi konstant ist
        Console.WriteLine(pi);
    }
}
```

## Implizit typisierte Variablen

C# erlaubt bei den lokalen Variablen eine Definition ohne der expliziten Angabe des Datentyps. Die Variablen werden in diesem Fall mit dem Schlüsselwort `var` definiert, der Typ ergibt sich infolge der Auswertung des Ausdrucks auf der rechten Seite der Initialisierungsanweisung zur Compilierzeit.

```
var i = 10; // i compiled as an int
var s = "untypisch"; // s is compiled as a string
var a = new[] {0, 1, 2}; // a is compiled as int[]
```

`var`-Variablen sind trotzdem typisierte Variablen, nur der Typ wird vom Compiler zugewiesen.

Vielfach werden `var`-Variablen im Initialisierungsteil von `for`- und `foreach`- Anweisungen bzw. in der `using`-Anweisung verwendet. Eine wesentliche Rolle spielen sie bei der Verwendung von anonymen Typen.

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

```
public class Program
{
    static void Main(string[] args)
    {
        //int num = 123;
        //string str = "asdf";
        //Dictionary<int, string> dict = new Dictionary<int, string>();
        var num = 123;
        var str = "asdf";
        var dict = new Dictionary<int, string>();
        Console.WriteLine("{0}, {1}, {2}", num.GetType(), str.GetType(), dict.GetType());
    }
}
```

Weitere Infos <https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/classes-and-structs/implicitly-typed-local-variables>

## Nullable - Leere Variablen

Ein “leer-lassen” ist nur für Referenzdatentypen möglich, Wertedatentypen können nicht uninitialisiert bleiben (Compilerfehler)

- \* Der Ausgangszustand generiert einen Fehler
- \* Initialisierung mit string text = null
- \* Evaluation von int i = null;

->

```
using System;

public class Program
{
    static void Main(string[] args){
        string text = null;    // Die Referenz zeigt auf kein Objekt im Heap
        //int i = null;
        if (text == null) Console.WriteLine("Die Variable hat keinen Wert!");
        else Console.WriteLine("Der Wert der Variablen ist {0}", text);
    }
}
```

Aus der Definition heraus kann zum Beispiel eine int Variable nur einen Wert zwischen int.MinValue und int.MaxValue annehmen. Eine null ist nicht vorgesehen und eine 0 gehört zum “normalen” Wertebereich.

Um gleichermaßen “nicht-besetzte” Werte-Variablen zu ermöglichen integriert C# das Konzept der sogenannte null-fähigen Typen (*nullable types*) ein. Dazu wird dem Typnamen ein Fragezeichen angehängt. Damit ist es möglich diesen auch den Wert null zuzuweisen bzw. der Compiler realisiert dies.

- \* einfache Variable ist mit null initialisierbar

->

```
using System;

public class Program
{
    static void Main(string[] args){
        int? i = null;
        if (i == null) Console.WriteLine("Die Variable hat keinen Wert!");
        else Console.WriteLine("Der Wert der Variablen ist {0}", i);
    }
}
```

Wie wird das Ganze umgesetzt? Jeder Typ? wird vom Compiler dazu in einen generischen Typ Nullable<Typ> transformiert, der folgende Methoden implementiert:

```

public struct Nullable <T>{
    private bool defined;
    public bool HasValue {get;}
    ...
    private T value;
    public T Value {get;}
    ...
    public T GetValueOrDefault() // value oder default Value entsprechend der
                                // der Liste unter dem untenstehenden Link
    ...
}

```

<https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/default-values-table>

## Aufgaben

- [ ] Experimentieren Sie mit Arrays und Enumerates. Schreiben Sie Programme, die Arrays nach bestimmten Einträgen durchsuchen. Erstellen Sie Arrays aus Enum Einträgen und zählen Sie die Häufigkeit des Vorkommens.
- [ ] Welche Funktion realisiert das folgende Codebeispiel?

```

using System;

public class Program
{
    static void Main(string[] args)
    {
        for (int number = 0; number < 20; number++)
        {
            bool prime = true;
            for (int i = 2; i <= number / 2; i++)
            {
                if (number % i == 0)
                {
                    prime = false;
                    break;
                }
            }
            if (prime == true) Console.WriteLine("{0}, ", number);
        }
    }
}

```

- [ ] Studieren Sie C# Codebeispiele. Einen guten Startpunkt bieten zum Beispiel die "1000 C# Examples" unter <https://www.sanfoundry.com/csharp-programming-examples/>

# Chapter 6

## C# Grundlagen III

Parameter	Kursinformationen
<b>Veranstaltung:</b>	Vorlesung Softwareentwicklung
<b>Semester:</b>	Sommersemester 2022
<b>Hochschule:</b>	Technische Universität Freiberg
<b>Inhalte:</b>	Einführung in die Basiselemente der Programmiersprache C# - Eingabe/Ausgabe, Ausnahmen
<b>Link auf den GitHub:</b>	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/05_CsharpGrundlagenIII.md">https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/05_CsharpGrundlagenIII.md</a>
<b>Autoren</b>	@author

## I/O Schreiboperation

C# selbst besitzt keine Anweisungen für die Ein- und Ausgabe von Daten, dazu existieren aber mehrere Bibliotheken, die im folgenden für die Bildschirmausgabe und das Schreiben in Dateien vorgestellt werden sollen.

Für das Schreiben stehen zwei Methoden `System.Console.WriteLine` und `System.Console.WriteLine`.

Diese decken erstens eine **große Bandbreite von Übergabeparametern** und bedienen zweitens verschiedene **Ausgabeschnittstellen**.

WriteLine() Methoden	Anwendung
1 <code>WriteLine()</code>	Zeilenumbruch
2 <code>WriteLine(UInt64)</code> , <code>WriteLine(Double)</code> , <code>WriteLine(Object)</code>	Ausgabe von Variablen der Basisdatentypen
3 <code>WriteLine(Object)</code>	Ausgabe einer <code>string</code> Variable
4 <code>WriteLine(String)</code>	Kombinierte Formatierung- String mit
5 <code>WriteLine(String, Object)</code>	Formatinformationen und nachfolgendes Objekt ...
6 <code>WriteLine(String, Object, Object)</code>	... nachfolgende Objekte
7 <code>WriteLine(String, Object, Object, Object)</code>	...
8 <code>WriteLine(String, Object[])</code>	...

## Einzelner Datentyp

Die Varianten 1-4 aus vorhergehender Tabelle übernehmen einen einzelnen Datentypen und geben dessen Wert aus. Dabei wird implizit die Methode `toString()` aufgerufen.

- Erinnerung an Steuerzeichen /n /t ->

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("One");
        Console.WriteLine("Two");
        Console.WriteLine("Three");
        Console.WriteLine();
        Console.WriteLine("Four");
    }
}
```

Achtung, für Referenzdatentypen bedeutet dies, dass die Referenz ausgegeben wird.

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        double[] values = new double[] {43.2234, 1.23123243, -123234.09};
        Console.WriteLine(values);
        Console.WriteLine(values.ToString());
    }
}
```

Lassen Sie uns schon mal etwas in die Zukunft schauen und die objektorientierte Implementierung dieser Idee erfassen:

```
using System;

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return "Person: " + Name + " " + Age;
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Person person = new Person { Name = "Bernhard von Cotta", Age = 55 };
        Console.WriteLine(person);
    }
}
```

## Kombinierte Formatierung von Strings

Die “Kombinierte Formatierung” unter C# ermöglicht eine breite Festlegung bezüglich des Formats der Ausgaben. Die folgenden Aussagen beziehen sich dabei aber nicht nur auf die Anwendung im Zusammenhang mit `WriteLine()` und `Write()` sondern können auch auf:

- `String.Format` (und damit `ToString()!`)
- `String.Builder`
- `Debug.WriteLine` ...

angewandt werden.

Die Ausdrücke folgen dabei folgenden Muster. Achtung, die rechteckigen Klammern illustrieren hier optionale Elemente [...]!

```
{ index [, alignment] [width][:format][precision]}
```

Element	Bedeutung
index	Referenz auf die nachfolgenden Folge der Objekte
alignment	Ausrichtung links- (-) oder rechtsbündig
width	Breite der Darstellung
format	siehe nachfolgende Tabellen
precision	Nachkommastellen bei gebrochenen Zahlenwerten

- Diskussion der Indizes, Durchtauschen der Indizes,
- Erzeugung Fehler Indizes
- Darstellung der Breite
- Positives negatives Vorzeichen der Breite
- Angabe der Formate, precision ->

Probieren sie es doch in folgendem Beispiel mal aus:

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        double[] values = new double[] {43.2234, 1.23123243, -123234.09};
        for (int index = 0; index < values.Length; index++)
        {
            Console.WriteLine("{0} -> {1}", index, values[index]);
        }
    }
}
```

Format	Symbol	Bedeutung	Beispiel
G		Default	
E, e		Exponentiell	1.052033E+003
F, f		Festkomma	123.45
X, x		Hexadezimal	1FF
P, p		Prozent	-38.8%
D, d		Dezimal	1231
N, n		Dezimal mit Trennzeichen	1.23432,12

Eine komplette Auflistung findet sich unter <https://docs.microsoft.com/de-de/dotnet/standard/base-types/standard-numeric-format-strings>

**Achtung:** Die Formatzeichen sind typspezifisch, es existieren analoge Zeichen mit unterschiedlicher Bedeutung für Zeitwerte

- Illustration, dass die Formatierungsmethoden auf verschiedenen Ebenen funktionieren
- Hinweis auf fehlender Möglichkeit einer Breitenangabe
- Einführung einer cultural Instanz

->

```
using System;
using System.Globalization;
using System.Threading;

public class Program
```

```

{
    static void Main(string[] args)
    {
        DateTime thisDate = new DateTime(2020, 3, 12);
        Console.WriteLine(thisDate.ToString("d")); // d = kurzes Datum
                                                // D = langes Datum
                                                // f = vollständig

        string[] cultures = new string [] {"de-DE", "sq-AL", "hy-AM"};
        foreach(string culture in cultures){
            Thread.CurrentThread.CurrentCulture = new CultureInfo(culture);
            Console.WriteLine("{0:5} - {1:D}", culture, thisDate);
        }
    }
}

```

**Merke:** Die kulturbbezogene Ausgabe ist auch für das Komma bei den gebrochenen Zahlen relevant.  
Dieser Aspekt wird in den Übungen thematisiert [Link](#).

## Zeichenfolgeninterpolation

Wesentliches Element der Zeichenfolgeninterpolation ist die “Ausführung” von Code innerhalb der Ausgabespezifikation. Die Breite der Möglichkeiten reicht dabei von einfachen Variablennamen bis hin zu komplexen Ausdrücken. Hier bitte Augenmaß im Hinblick auf die Lesbarkeit walten lassen! Angeführt wird ein solcher Ausdruck durch ein \$.

```

using System;

public class Program
{
    static void Main(string[] args)
    {
        // Composite formatting:
        var date = DateTime.Now;
        string city = "Freiberg";
        Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", city, date.DayOfWeek, date);
        // String interpolation:
        Console.WriteLine($"Hello, {city}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");
    }
}

```

Die Darstellung des Ausdrucks folgt dabei der Semantik:

{<interpolatedExpression>[,<alignment>] [:<formatString>]}

Damit lassen sich dann sehr mächtige Ausdrücke formulieren vgl. [Link](#) oder aber in den Beispielen von

```
using System;
```

```

public class Program
{
    static void Main(string[] args)
    {
        int ivalue = 33;
        double fvalue = 43.1231;

        // Berechnungen
        Console.WriteLine($"{ivalue * 10000,20}");
        Console.WriteLine($"{ivalue * 10000,20:E});

        // Fallunterscheidungen mit ternärem Operator
        Console.WriteLine($"{ivalue < 5 ? ivalue.ToString() : "invalid"}");
        Console.WriteLine($"{(fvalue < 1000 ? fvalue : fvalue/1000):f2}"+
                        $"{fvalue < 1000 ? "Euro" : "kEuro"}");
    }
}

```

```

    }
}
```

## Ziele der Schreiboperationen

Üblicherweise möchte man die Ausgabegeräte (Konsole, Dateien, Netzwerk, Drucker, etc.) anpassen können. `System.IO` bietet dafür bereits verschiedene Standardschnittstellen.

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Ausgabe auf das Standard-Gerät");
        Console.Out.WriteLine("Ausgabe nach Out (default die Konsole)");
        //Console.Error.WriteLine("Ausgabe an die Fehlerschnittstelle");
    }
}
```

Die dafür vorgesehenen Standardeinstellungen können aber entsprechend umgelenkt werden. Dafür greift C# analog zu vielen anderen Sprachen (und Betriebssysteme) das Stream Konzept auf, dass eine Abstraktion für verschiedene Quellen und Senken von Informationen bereitstellt. Darauf aufbauend sind dann Lese- / Schreiboperationen möglich.

Einen guten Überblick dazu bietet das nachfolgende Tutorial:

### !Streams

Von der Klasse `System.IO.Stream` leiten sich entsprechend `MemoryStream`, `FileStream`, `NetworkStream` ab. Diese bringen sowohl eigene Lese-/Operationen mit, gleichzeitig ist aber auch das "Umlenken" von Standardoperationen möglich.

```
// Das Beispiel entstammt der Dokumentation des .Net Frameworks
// https://docs.microsoft.com/de-de/dotnet/api/system.console.out?view=netframework-4.7.2

using System;
using System.IO;

public class Example
{
    public static void Main()
    {
        // Get all files in the current directory.
        string[] files = Directory.GetFiles(".");
        Array.Sort(files);

        // Display the files to the current output source to the console.
        Console.WriteLine("First display of filenames to the console:");
        Array.ForEach(files, s => Console.Out.WriteLine(s));
        Console.Out.WriteLine();

        // Redirect output to a file named Files.txt and write file list.
        StreamWriter sw = new StreamWriter(@"..\Files.txt");
        sw.AutoFlush = true;
        Console.SetOut(sw);
        Console.Out.WriteLine("Display filenames to a file:");
        Array.ForEach(files, s => Console.Out.WriteLine(s));
        Console.Out.WriteLine();

        // Close previous output stream and redirect output to standard output.
        Console.Out.Close();
        sw = new StreamWriter(Console.OpenStandardOutput());
        sw.AutoFlush = true;
```

```

Console.SetOut(sw);

// Display the files to the current output source to the console.
Console.Out.WriteLine("Second display of filenames to the console:");
Array.ForEach(files, s => Console.Out.WriteLine(s));
}
}

```

Darüber hinaus stehen aber auch spezifische Zugriffsmethoden zum Beispiel für Dateien zur Verfügung.

```

using System;
using System.Collections.Generic;
using System.Text;
using System.IO; //Erforderlicher Namespace
namespace Wiki
{
    class Program
    {
        static void Main(string[] args)
        {
            // Unmittelbare Ausgabe in eine Datei
            File.WriteAllText(@"C:\test1.txt", "Sehr einfach");

            // Einlesen des gesamten Inhalts einer Textdatei und
            // Ausgabe auf dem Bildschirm
            string text = File.ReadAllText(@"C:\test1.txt");
            Console.WriteLine(text);
        }
    }
}

```

**Zur Erinnerung:** Das @ vor einen String markiert ein verbatim string literal - alles in der Zeichenfolge, was normalerweise als Escape-Sequenz interpretiert werden würde, wird ignoriert.

## Beispiel

Zur Erinnerung, in Markdown werden Tabellen nach folgendem Muster aufgebaut:

Name	Alter	Aufgabe
Peter	42	C-Programmierer
Astrid	23	Level Designer

Name	Alter	Aufgabe
Peter	42	C-Programmierer
Astrid	23	Level Designer

Geben Sie die Daten bestimmte Fußballvereine in einer Markdown-Tabelle aus.

```

using System;

public class Program
{
    static void Main(string[] args)
    {
        string [] clubs = {"Blau Weiß", "Grün Gelb 1905", "Borussia Tralla Trulla", "Eintracht"};
        int [] punkte = {12, 10, 9, 5};
        // Wie lang ist ein Clubname maximal?
        int maxlenlength = 0;
        foreach(string club in clubs)
        {

```

```

        maxlen = club.Length < maxlen ? maxlen : club.Length ;
    }
    maxlen += 1;
    // Ausgabe
    string output;
    output = "| ";
    output += "Verein".PadRight(maxlen, ' ') + "| Punkte |\n";
    output += "|:" + "".PadRight(maxlen, '-') + "|-----|\n";
    for (int i = 0; i < clubs.Length; i++){
        output += String.Format("| {0}| {1}, -7}|\n", clubs[i].PadRight(maxlen, ' '), punkte[i]);
    }
    Console.WriteLine(output);
}
}

```

**Frage:** Welche Annahmen werden implizit bei der Erstellung der Tabelle getroffen? Wo sehen Sie Verbesserungsbedarf?

## I/O Leseoperationen

Eine Software ist zumeist von Nutzereingaben abhängig. Diese können:

- in Form von Dateiinhalten daherkommen,
- als Eingaben beim Start des Programms vorgegeben oder
- zur Laufzeit eingelesen werden.

Vergegenwärtigen Sie sich die Vor- und Nachteile der unterschiedlichen Formen mit Blick auf die Entwicklung einer Software!

## Kommandozeilenargumente

Eine grundlegende Eingabemöglichkeit ist die Übergabe von Parametern beim Aufruf des Programms von der Kommandozeile.

```

using System;

public class Program
{
    static int Main(string[] args)
    {
        System.Console.WriteLine($"How many arguments are given? - {args.Length}");
        foreach (string argument in args)
        {
            System.Console.WriteLine(argument);
        }
        return 0;
    }
}

```

Bei der Vorgabe von Commandline Parametern unterscheidet sich der Ablauf etwas, je nachdem wie Sie Ihren Code “bauen”.

### Kommandozeile

1. mono Compiler

```
mcs CmdLineParams.cs
mono CmdLineParams.exe Das ist "ein Test"
```

2. dotnet Umgebung

```
dotnet new console --name dotnet_cmdLinePara --framework net5.0
cd dotnet_cmdLinePara
dotnet run -- -1 -zwei --drei vier"
```

## Entwicklungsumgebung am Beispiel von VS Code

Suchen Sie den zugehörigen `.vscode` Ordner innerhalb Ihres Projektes und öffnen Sie `launch.json`. Ergänzen Sie die Kommandozeilenparameter für den Punkt `args`.

```
"configurations": [
  {
    "name": ".NET Core Launch (console)",
    "args": [], // PUT YOUR ARGUMENTS HERE
    ...
  }
]
```

## Auswertung der Kommandozeilenparameter

**Achtung:** Im folgenden bemühen wir uns selbst um das Parsen der Kommandozeilenargumente. Dies sollte in realen Vorhaben entsprechenden Bibliotheken überlassen werden [Using System.CommandLine!](#)

```
using System;

public class Program
{
    static int Main(string[] args)
    {
        System.Console.WriteLine("Geben Sie einen Ganzahlwert und einen String als Argumente ein!");
        if (args.Length == 0)
        {
            System.Console.WriteLine("Offenbar keine Eingabe - Fehler!");
            return 1;
        }
        if ((args.Length == 1) || (args.Length > 2))
        {
            System.Console.WriteLine("Falsche Zahl von Parametern - Fehler!");
            return 1;
        }
        if (args.Length == 2) // Erwartete Zahl von Parametern
        {
            // hier müssen wir jetzt die Daten parsen und die Datentypen evaluieren
        }
        return 0;
    }
}
```

Im übernächsten Abschnitt prüfen wir den Datentyp und den Inhalt der Parameter auf Ihre Korrektheit.

## Leseoperationen von der Console

Leseoperationen von der Console (oder anderen Streams) werden durch zwei Methoden abgebildet:

```
public static int Read();
public static string ReadLine();

using System;

public class Program
{
    static void Main(string[] args)
    {
        char ch;
        int x;
        Console.WriteLine("Print Unicode-Indizes");
        do
        {
            x = Console.Read(); // Lesen eines Zeichens
```

```

        ch = Convert.ToChar(x);
        Console.WriteLine($"Unicode {x}- Sign {Convert.ToInt32(x)}\n");
        // Hier könnte man jetzt eine Filterung realisieren
    } while (ch != '+');
}
} // zu Demonstrationszwecken

```

Das Beispiel zeigt sehr schön, wie verschiedene Zeichensätze auf unterschiedlich lange Codes abgebildet werden. Das chinesische Zeichen, dass vor dem Escape-Zeichen “+” steht generiert einen 2Byte breiten Wert.

## Transformation der Eingaben

```

using System;

public class Program
{
    static int Main(string[] args)
    {
        Console.WriteLine("Geben Sie einen Ganzahlwert und einen String als Argumente ein!");
        string inputstring = Console.ReadLine();
        string[] param = inputstring.Split(' ');

        foreach(string s in param){
            Console.Write(s + " ");
        }

        if (param.Length == 0)
        {
            Console.WriteLine("Offenbar keine Eingabe - Fehler!");
            return 1;
        }
        if (param.Length == 2) // Erwartete Zahl von Parametern
        {
            long num1 = long.Parse(param[0]);
            long num2 = System.Convert.ToInt64(param[0]);
            long num3;
            long.TryParse(param[0], out num3);
            Console.WriteLine($"{num1} {num2} {num3}");
            string text = param[1];
            Console.WriteLine($"{text}");
        }

        return 0;
    }
}

```

Dabei nutzt das obige Beispiel 3 Formen der Interpretation der Daten. In den beiden ersten Fällen ist der Entwickler für das Abfangen der *Exceptions* verantwortlich. Die letzte Variante kapselt dies intern und gibt die möglicherweise eingetretene Ausnahme über die Rückgabewerte aus. Der Code auf Seiten der Anwendung wird kompakter.

1. `long.Parse` parst die Eingabe in einen ganzzahligen Wert, wirft jedoch eine Ausnahme aus, wenn dies nicht möglich ist, wenn die bereitgestellten Daten nicht numerisch sind.
2. `Convert.ToInt64()` konvertiert die Zeichenkettendaten in einen korrekten echten int64-Wert und wirft eine Ausnahme aus, wenn der Wert nicht konvertiert werden kann.
3. Einen alternativen Weg schlägt `int.TryParse()` ein. Die TryParse-Methode ist wie die Parse-Methode, außer dass die TryParse-Methode keine Ausnahme auslöst, wenn die Konvertierung fehlschlägt.

Auf die Verwendung der Ausnahmen wird im folgenden Abschnitt eingegangen.

## Ausnahmebehandlungen

Ausnahmen sind Fehler, die während der Ausführung einer Anwendung auftreten. Die C#-Funktionen zur Ausnahmebehandlung unterstützen bei der Handhabung von diesen unerwarteten oder außergewöhnlichen Situationen, die beim Ausführen von Programmen auftreten.

```
using System;
using System.Globalization;

public class Program
{
    static void Main(string[] args)
    {
        // Beispiel 1: Zugriff auf das Filesystem eines Rechners aus dem Netz
        System.IO.FileStream file = null;
        //System.IO.FileInfo fileInfo = new System.IO.FileInfo(@"NoPermission.txt");
        // Beispiel 2: Division durch Null
        int a = 0, b = 5;
        //a = b / a;
        //long num = long.Parse("NN");
    }
}
```

**Merke:** Wenn für eine spezifische Ausnahme kein Ausnahmehandler existiert, beendet sich das Programm mit einer Fehlermeldung.

---

### Schlüsselwörter

	Bedeutung
try	Ein try-Block wird verwendet, um einen Bereich des Codes zu kapseln. Wenn ein Code innerhalb dieses try-Blocks eine Ausnahme auslöst, wird diese durch den zugehörigen catch-Block behandelt.
catch	Hier haben Sie die Möglichkeit, die (spezifische) Ausnahme zu behandeln, zu protokollieren oder zu ignorieren.
finally	Der finally-Block ermöglicht es Ihnen, bestimmten Code auszuführen, wenn eine Ausnahme geworfen wird oder nicht. Zum Beispiel das Entsorgen eines Objekts aus dem Speicher.
throw	Das throw-Schlüsselwort wird verwendet, um eine neue Ausnahme zu erzeugen, die in einem try-catch-finally-Block aufgefangen wird.

---

```
try
{
    // Statement which can cause an exception.
}
catch(Type x)
{
    // Statements for handling the exception
}
finally
{
    //Any cleanup code
}
```

Dabei gelten folgende Regeln für den Umgang mit Ausnahmen:

- Alle Ausnahmen sind von `System.Exception` abgeleitet und enthalten detaillierte Informationen über den Fehler, z.B. den Zustand der Aufrufliste und eine Textbeschreibung des Fehlers.
- Ausnahmen, die innerhalb eines try-Blocks auftreten, werden auf einen Ausnahmehandler, der mit dem Schlüsselwort `catch` gekennzeichnet ist, umgeleitet.
- Ausnahmen werden durch die CLR ausgelöst oder in Software mit dem `throw` Befehl.
- ein finally-Block wird im Anschluss an die Aktivierung eines catch Blockes ausgeführt, wenn eine Ausnahme ausgelöst wurde. Hier werden Ressourcen freizugeben, beispielsweise ein Stream geschlossen.

```
using System;
```

```

public class Program
{
    static int Berechnung(int a, int b)
    {
        int c=0;
        try
        {
            checked {c = a + b;}          // Fall 1
            // c = a / b;                // Fall 2
        }
        catch (OverflowException e)
        {
            Console.WriteLine("[Overflow] " + e.Message);
            //throw;
        }
        finally
        {
            Console.WriteLine("Finally Block");
        }
        Console.WriteLine("Hier sind wir am Ende der Funktion!");
        return c;
    }

    static void Main(string[] args)
    {
        // try
        // {
        Berechnung(int.MaxValue, 1);      // Fall 1
        //Berechnung(2, 0);              // Fall 2
        // }
        /*
        catch (DivideByZeroException e)
        {
            Console.WriteLine("[DivideByZero] " + e.Message);
        }
        catch (Exception e)
        {
            Console.WriteLine("[GeneralExcept] " + e.Message);
        }
        */
        Console.WriteLine("Hier sind wir am Ende des Hauptprogrammes!");
    }
}

```

## Best Practice

Die folgende Darstellung geht auf die umfangreiche Sammlung von Hinweisen zum Thema Exceptions unter <https://docs.microsoft.com/de-de/dotnet/standard/exceptions/best-practices-for-exceptions> zurück.

- Differenzieren Sie zwischen Ausnahmevermeidung und Ausnahmebehandlung anhand der erwarteten Häufigkeit und der avisierten „Signalwirkung“

```

// Ausnahmevermeidung
if (conn.State != ConnectionState.Closed)
{
    conn.Close();
}

// Ausnahmebehandlung
try
{
    conn.Close();
}

```

```

    }
  catch (InvalidOperationException ex)
  {
    Console.WriteLine(ex.GetType().FullName);
    Console.WriteLine(ex.Message);
  }
}

```

- Nutzen Sie die Möglichkeit von Ausnahmen statt einen Fehlercode zurückzugeben
- Verwenden Sie dafür die vordefinierten .NET-Ausnahmetypen!
- Selbst definierte Ausnahmen enden auf das Wort *Exception*!
- Vermeiden Sie unklare Ausgaben für den Fall einer Ausnahmebehandlung, stellen Sie alle Informationen bereit, die für die Analyse des Fehlers nötig sind
- Stellen Sie den Status einer Methode wieder her, die von einer Ausnahmebehandlung betroffen war (Beispiel: Code für Banküberweisungen, Abbuchen von einem Konto und Einzahlung auf ein anderes. Scheitert die zweite Aktion muss auch die erste zurückgefahren werden.)
- Testen Sie Ihre Ausnahmebehandlungsstrategie!

## Beispiel Exception-Handling

Schreiben Sie die Einträge eines Arrays in eine Datei!

Lösung unter [ExceptionHandling.cs](#)

Schritt	Fragestellungen
1	Welche Fehler können überhaupt auftreten? Welche Fehler werden durch die Implementierung abgefangen?
2	Wo sollen die Fehler abgefangen werden?
3	Gibt es Prioritäten bei der Abarbeitung?
4	Sind abschließende "Arbeiten" notwendig?

[Link auf die Dokumentation der StreamWriter Klasse](#)

## Aufgaben

- [ ] Entwickeln Sie ein Programm, dass als Kommandozeilen-Parameter eine Funktionsnamen und eine Ganzzahl übernimmt und die entsprechende Ausführung realisiert. Als Funktionen sollen dabei **Square** und **Reciprocal** dienen. Der Aufruf erfolgt also mit

```
mono Calculator Square 7
mono Calculator Reciprocal 9
```

Welche Varianten der Eingaben müssen Sie prüfen? Erproben Sie Ihre Lösung mit einem besonders "böswilligen" Nutzer :-)

```

using System;

class MainClass
{
  static double Square(int num) => num * num;
  static double Reciprocal (int num) => 1f / num;
  static void Main(string[] args)
  {
    bool Error = false;
    double result = 0;
    int num = 1;
    if (args.Length == 2)
    {
      // Hier geht es weiter, welche Fälle müssen Sie bedenken?
      // int.TryParse(args[1], out num) erlaubt ein fehlertolerantes Parsen
      // eines strings
    }
  }
}
```

```
else Error = true;
if (Error)
{
    Console.WriteLine("Please enter a function and a numeric argument.");
    Console.WriteLine("Usage: Square    <int> or\n                  Reciprocal <int>");
}
else
{
    Console.WriteLine("{0} Operation on {1} generates {2}", args[0], num, result );
}
}
```



## Chapter 7

# Programmfluss und Funktionen

---

Parameter Kursinformationen

---

**Veranstaltung:** Vorlesung Softwareentwicklung

Semester Sommersemester 2022

**Hochschule:** Technische Universität Freiberg

**Inhalte:** Programmfluss und Funktionsstrukturen in C#

**Link auf den** [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/06\\_ProgrammflussUndFunktionen.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/06_ProgrammflussUndFunktionen.md)

**GitHub:**

**Autoren** @author

---

## Anweisungen

In den vorangegangenen Beispielen haben wir zu Illustrationszwecken bereits mehrfach auf Kontrollstrukturen, die den Programmfluss steuern zurückgegriffen. Nunmehr sollen diese in einem kurzen Überblick systematisch eingeführt werden.

Anweisungen setzen sich zusammen aus Zuweisungen, Methodenaufrufen, Verzweigungen Sprunganweisungen und Anweisungen zur Fehlerbehandlung.

Der letztgenannte Bereich wurde im Zusammenhang mit der Ein- und Ausgabe von Daten bereits thematisiert.

## Verzweigungen

**if**

Verzweigungen in C# sind allein aufgrund von boolschen Ausdrücken realisiert. Eine implizite Typwandlung wie in C **if** (*value*) ist nicht vorgesehen. Dabei sind entsprechend kombinierte Ausdrücke möglich, die auf boolsche Operatoren basieren.

```
if (BooleanExpression) Statement else Statement  
using System;  
  
public class Program  
{  
    static void Main(string[] args)  
    {  
        int a = 23, b = 3;  
        if (a > 20 && b < 5)  
        {  
            Console.WriteLine("Wahre Aussage ");  
        }  
    }  
}
```

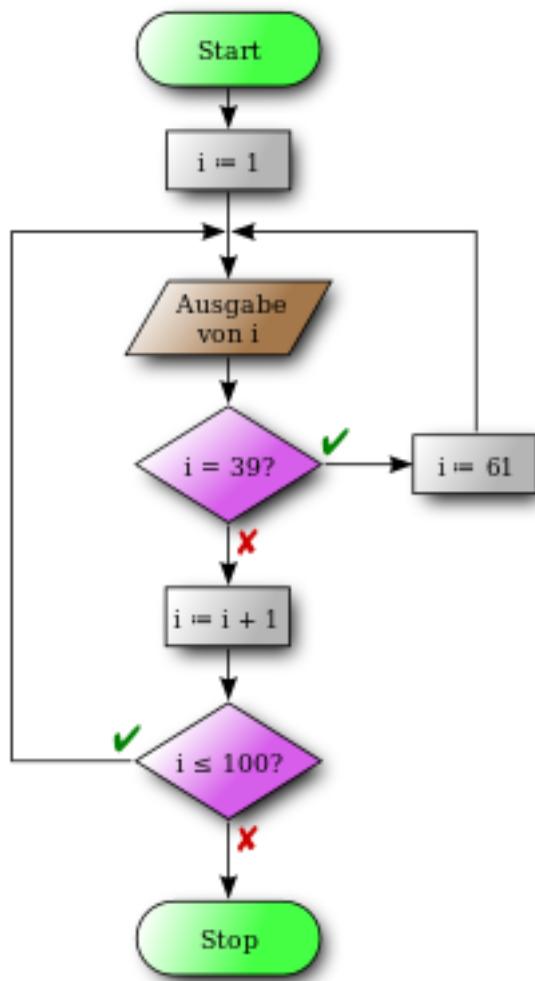


Figure 7.1: Flussdiagramm

```
        else
    {
        Console.WriteLine("Falsche Aussage ");
    }
}
```

Warum sollte ich in jedem Fall Klammern um Anweisungen setzen, gerade wenn diese nur ein Zeile umfasst?

- Was passiert wenn  $A == \text{false}$ ? Es gibt keine Ausgabe mehr. Wenn man aber eine Klammer um das innere if setzt, folgt daraus eine gänzlich andere Bedeutung. ->

```
using System;
```

```
public class Program
{
    static void Main(string[] args)
    {
        bool A = false, B = false;
        if (A)
        //{
            if (B)
                Console.WriteLine("Fall 1"); // A & B
        //}
        else
            Console.WriteLine("Fall 2"); // A & not B

        Console.WriteLine("Programm abgeschlossen");
    }
}
```

Versuchen Sie nachzuvollziehen, welche Wirkung die Klammern in Zeile 9 und 12 hätten. Wie verändert sich die Logik des Ausdruckes?

	Variante mit Klammern	Variante ohne Klammern
Ausgabe 1	$A \wedge B$	$A \wedge B$
Ausgabe 2	$\overline{A}$	$A \wedge \overline{B}$

**Merke:** Das setzen der Klammern steigert die Lesbarkeit ... nur bei langen `else if` Reihen kann drauf verzichtet werden.

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Enter a character: ");
        char ch = (char)Console.Read();
        if (Char.IsUpper(ch))
            Console.WriteLine("The character is an uppercase letter.");
        else if (Char.IsLower(ch))
            Console.WriteLine("The character is a lowercase letter.");
        else if (Char.IsDigit(ch))
            Console.WriteLine("The character is a number.");
        else
            Console.WriteLine("The character is not alphanumeric.");
    }
}

switch
```

Die **switch**-Anweisung ist eine Mehrfachverzweigung. Sie umfasst einen Ausdruck und mehrere Anweisungsfolgen.

gen, die durch `case` eingeleitet werden.

```
using System;

public enum Color { Red, Green, Blue }

public class Program
{
    static void Main(string[] args)
    {
        Color c = (Color) (new Random()).Next(0, 3);
        switch (c)
        {
            case Color.Red:
                Console.WriteLine("The color is red");
                break;
            case Color.Green:
                Console.WriteLine("The color is green");
                break;
            case Color.Blue:
                Console.WriteLine("The color is blue");
                break;
            default:
                Console.WriteLine("The color is unknown.");
                break;
        }
    }
}
```

**Aufgabe:** Realisieren Sie das Beispiel als Folge von `if` `else` Anweisungen.

Anders als bei vielen anderen Sprachen erlaubt C# `switch`-Verzweigungen anhand von `strings` (zusätzlich zu allen Ganzzahl-Typen, `bool`, `char`, `enums`). Interessant ist die Möglichkeit auf `case: null` zu testen!

Es fehlt hier aber die Möglichkeit sogenannte *Fall Through* durch das Weglassen von `break`-Anweisungen zu realisieren.

**Jeder switch muss mit einem `break`, `return`, `throw`, `continue` oder `goto` beendet werden.**

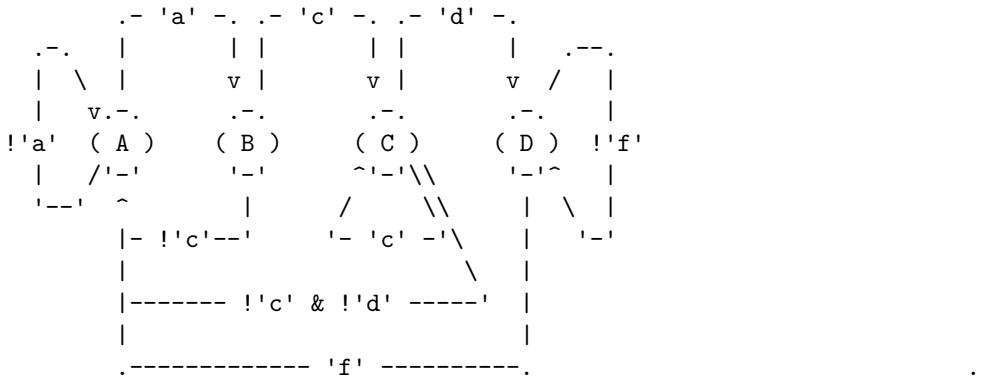
```
using System;

public class Program
{
    static void Main(string[] args)
    {
        string day = "Sonntag";
        string output;
        switch (day){
            case "Montag": case "Dienstag":
            case "Mittwoch": case "Donnerstag": case "Freitag":
                output = "Wochentag";
                break;
            case "Samstag": case "Sonntag":
                output = "Wochenende";
                break;
            default:
                output = "Kein Wochentag!";
                break;
        }
        Console.WriteLine(output);
    }
}
```

**Achtung:** Beachten Sie den Unterschied zwischen dem `switch`-Ausdruck [Link](#) und der `switch`-

Anweisung, die wir hier besprechen.

Ein weiteres Anwendungsbeispiel ist die Implementierung eines Automaten. Nehmen wir an, dass wir einen Sequenzdetektor entwerfen wollen, der das Auftreten des Musters  $a\{c\}d$  in einem Signalverlauf erkennt! Der Zustand kann mit  $f$  verlassen werden.



Wir nutzen dafür ein `enum`, dessen Inhalt mittels `switch` ausgewertet und fortgeschrieben wird. Das `enum` umfasst die vier Zustände A-D.

```
using System;

public class Program
{
    enum states {A, B, C, D};
    static void Main(string[] args)
    {
        string inputs;
        states state = states.A;
        Console.WriteLine("Geben Sie die Eingabefolge für die State-Machine vor: ");
        inputs = "aaabcccf";
        Console.WriteLine(inputs);
        bool sequence = false;
        foreach(char sign in inputs){
            Console.WriteLine("{0} -> {1} ", state, sign);
            switch (state){
                case states.A:
                    if (sign == 'a') state = states.B;
                    break;
                case states.B:
                    if (sign == 'c') state = states.C;
                    else state = states.A;
                    break;
                case states.C:
                    if (sign == 'd') state = states.D;
                    else if (sign != 'c') state = states.A;
                    break;
                case states.D:
                    if (sign == 'f') {
                        state = states.A;
                        sequence = true;
                    }
                    break;
            }
            Console.WriteLine("-> {0}", state);
            if (sequence)
            {
                Console.WriteLine("Sequenz erkannt!");
                sequence = false;
            }
        }
    }
}
```

```

        }
    }
}
}
```

**Frage:** Sehen Sie in der Lösung eine gut wartbare Implementierung?

C# 7.0 führt darüber hinaus das *pattern matching* mit switch ein. Damit werden komplexe Typ und Werteprüfungen innerhalb der case Statements möglich.

Eine beispielhafte Anwendung sei im folgenden Listing dargestellt.

```
public static double ComputeArea_Version(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
        case Triangle t when t.Base == 0 || t.Height == 0:
        case Rectangle r when r.Length == 0 || r.Height == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Triangle t:
            return t.Base * t.Height / 2;
        case Rectangle r:
            return r.Length * r.Height;
    }
}
```

Codebeispiel aus [MSDoku](#)

**Achtung:** C# 8.0 erweitert das Spektrum hier noch einmal mit einem **is** Keyword, dass die Lesbarkeit und Eindeutigkeit erhöht. Zudem wird **switch** mit der *fat arrow* Syntax erweitert und kann unmittelbar Werte zurückgeben.

## Schleifen

Eine Schleife wiederholt einen Anweisungs-Block – den sogenannten Schleifenrumpf oder Schleifenkörper –, solange die Schleifenbedingung als Laufbedingung gültig bleibt bzw. als Abbruchbedingung nicht eintritt. Schleifen, deren Schleifenbedingung immer zur Fortsetzung führt oder die keine Schleifenbedingung haben, sind Endlosschleifen.

### Zählschleife - for

```
for (initializer; condition; iterator)
    body
```

Üblich sind für alle drei Komponenten einzelne Anweisungen. Das erhöht die Lesbarkeit, gleichzeitig können aber auch komplexere Anweisungen integriert werden.

- Hinweis, dass Dekrementieren und Inkrementieren durch beliebige andere Funktionen ersetzt werden können. ->

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        for (int i = 0, j = 10;
             i < 10 && j > 5;
             Console.WriteLine("Start: i={0}, j={1}", i, j), i++, j--)
```

```
        {
            //empty
        }
    }
}

Lassen Sie uns zwei ineinander verschachtelte Schleifen nutzen, um eine  $[m \times n]$  Matrix zu befüllen.

using System;

public class Program
{
    static void Main(string[] args)
    {
        Random rnd = new Random();
        int[,] matrix = new int[3,5];
        for(int i=0;i<3;i++)
        {
            for(int j=0;j<5;j++)
            {
                matrix[i, j]= rnd.Next(1, 10);
                Console.WriteLine("{0},{1}", matrix[i,j]);
                if (j == 4) Console.WriteLine();
            }
        }
        //Console.WriteLine($"{matrix.GetLength(0)}, {matrix.GetLength(1)}");
    }
}
```

## Kopf- Fußgesteuerte schleife - while/do while

Eine **while**-Schleife führt eine Anweisung oder einen Anweisungsblock so lange aus, wie ein angegebener boolescher Ausdruck gültig ist. Da der Ausdruck vor jeder Ausführung der Schleife ausgewertet wird, wird eine while-Schleife entweder nie oder mehrmals ausgeführt. Dies unterscheidet sich von der do-Schleife, die ein oder mehrmals ausgeführt wird.

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        int counter = 0;
        while (counter < 5) // "Kopfgesteuert"
        {
            Console.WriteLine($"While counter! The counter is {counter}");
            counter++;
        }

        //counter = 0;
        do
        {
            Console.WriteLine($"Do while counter! The counter is {counter}");
            counter++;
        } while (counter < 5); // "Fußgesteuert"
    }
}
```

## Iteration - foreach

Als alternative Möglichkeit zum Durchlaufen von Containern, die `IEnumerable` implementieren bietet sich die Iteration mit `foreach` an. Dabei werden alle Elemente nacheinander aufgerufen, ohne dass eine Laufvariable nötig ist.

- foreach (char in “TU Bergakademie”) ->

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        int [] array = new int [] {1,2,3,4,5,6};
        foreach (int entry in array){
            Console.WriteLine("{0} ", entry);
        }
    }
}
```

## Sprünge

Während `label` bestimmte Positionen im Code adressiert, lassen sich mit `break` Schleifen beenden, dient `continue` der Unterbrechung des aktuellen Blockes.

---

### Sprunganweisung Wirkung

---

<code>break</code>	beendet die Ausführung der nächsten einschließenden Schleife oder <code>switch</code> -Anweisung
<code>continue</code>	realisiert einen Sprung in die nächste Iteration der einschließenden Schleife
<code>goto &lt;label&gt;</code>	Sprung an eine Stelle im Code, die durch das Label markiert ist
<code>return</code>	beendet die Ausführung der Methode, in der sie angezeigt wird und gibt den optionalen nachfolgenden Wert zurücksetzen

---

->

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        int dummy = 0;
        for (int y = 0; y < 10; y++)
        {
            for (int x = 0; x < 10; x++)
            {
                if (x == 5 && y == 5)
                {
                    goto Outer;
                }
                dummy++;
            }
        Outer:
            Console.WriteLine(dummy);
    }
}
```

Vgl. Links zur Diskussion um `goto` auf <https://de.wikipedia.org/wiki/Sprunganweisung>

## Funktionen in C

Im Grunde ist die separate von Operationen, ohne die Einbettung in entsprechende Klassen nur beschränkt zielführend. In C# können Funktionen und Prozeduren nur als Methoden innerhalb von Klassen angelegt werden. Allerdings lassen sich insbesondere die Konzepte der Parameterübergaben auch ohne dass zuvor die OO-Konzepte besprochen wurden, erläutern.

C# kennt *benannte* und *anonyme* Methoden, in diesem Abschnitt wird nur auf die benannten Methoden eingegangen, letztgenannte folgen zu einem späteren Zeitpunkt. Prozeduren sind Funktionen ohne Rückgabewert, sie werden entsprechend als **void** deklariert.

- Bedeutung von void static
- static void Calc(float p) Überladen von Funktionen ->

```
using System;

public class Program
{
    static void Calc(int p)          // Funktions / Methodendefinition
    {
        p = p + 1;
        Console.WriteLine(p);
    }

    static void Main(string[] args)
    {
        Calc(5f);                  // Funktions / Methodenaufruf
    }
}
```

## Verkürzte Darstellung

Methoden können seit C#6 in Kurzform in einer einzigen Zeile angegeben werden (*Expression-bodied function members*). Dafür nutzt C# die Syntax von Lambda Ausdrücken, die für anonyme Funktionen verwendet werden. Dem => entsprechend wird von der *Fat Arrow Syntax* gesprochen.

```
using System;

public class Program
{
    static string Combine(string fname, string lname) => $"{fname.Trim()} {lname.Trim()}";

    static void Main(string[] args)
    {
        // Nutzereingaben mit Leerzeichen
        Console.WriteLine(Combine("      Sebastian", "Zug      "));
    }
}
```

Die oben genannte Funktion, die die Kundendaten von Lehrzeichen befreit, ist also äquivalent zur Darstellung von:

```
public static string CombineNames(string fname, string lname)
{
    return $"{fname.Trim()} {lname.Trim()}";
```

Damit lassen sich einfache Funktionen sehr kompakt darstellen.

```
public class Program
{
    // Prozedur
    static void Print(int p)  => Console.WriteLine(p);
    // Funktion
    static int Increment(int p)    => p+1;

    static void Main(string[] args){
        int p = 6, result;
        result = Increment(p);
        print(result);
```

```

    }
}
```

## Übergeben von Parametern

Ohne weitere Refenzparameter werden Variablen an Funktionen bei

- Wertetypen (Basistypen, Enumerationen, structs, Tupel) mittels *pass-by-value*
- Referenztypen (Klassen, Interfaces, Arrays, Delegates) mittels *pass-by-reference*

an eine Funktion übergeben.

Ersetzen Sie die integer Variable p durch ein Array der Größe 1 und beobachten Sie das veränderte Ergebnis. Nutzen Sie das Schlüsselwort ref um eine datentypunabhängige pass-by-reference Übergabe zu realisieren. ->

```
using System;
```

```
public class Program
{
    static void Calc(int p)
    {
        p = p + 1;
        Console.WriteLine("Innerhalb von Calc {0}", p);
    }

    static void Main(string[] args){
        int p = 6; // Wertedatentyp
        //int [] p = new int [] {6}; // Referenzdatentyp
        Calc(p);
        Console.WriteLine("Innerhalb von Main {0}", p);
    }
}
```

**Merke:** Die Namesgleichheit der Variablennamen in der Funktion und der Main ist irrelevant.

Welche Lösungen sind möglich den Zugriff einer Funktion auf eine übergebene Variable generell sicherzustellen?

### Ansatz 0 - Globale Variablen

... sind in C# als isoliertes Konzept nicht implementiert, können aber als statische Klassen realisiert werden.

```
using System;

public static class Counter
{
    public static int globalCounter = 0;
}

public class Program
{
    static void IncrementsCounter(){
        Counter.globalCounter++;
    }

    static void Main(string[] args){
        Console.WriteLine(Counter.globalCounter);
        IncrementsCounter();
        Console.WriteLine(Counter.globalCounter);
    }
}
```

### Ansatz 1 - Rückgabe des modifizierten Wertes

Wir benutzen return um einen Wert aus der Funktion zurückzugeben. Allerdings kann dies nur ein Wert sein. Der Datentyp muss natürlich mit dem der Deklaration übereinstimmen.

```
using System;

public class Program
{
    static int Calc( int input)
    {
        // operationen über P
        int output = 5 * input;
        return output;
    }

    static void Main(string[] args){
        int p = 5;
        int a = Calc(p);
        Console.WriteLine(a);
    }
}
```

### Ansatz 2 - Übergabe als Referenz

Bei der Angabe des `ref`-Attributes wird statt der Variablen in jedem Fall die Adresse übergeben. Es ist aber lediglich ein Attribut der Parameterübergabe und kann isoliert nicht genutzt werden, um die Adresse einer Variablen zu bestimmen (vgl C: `int a=5; int *b=&a`).

Vorteil: auf beliebig viele Parameter ausweisbar, keine Synchronisation der Variablennamen zwischen Übergabeparameter und Rückgabewert notwendig.

```
using System;

public class Program
{
    static void Calc(ref int x)
    {
        x = x + 1;
        Console.WriteLine("Innerhalb von Calc {0}", x);
    }

    static void Main(string[] args){
        int p = 1;
        Calc(ref p);
        Console.WriteLine("Innerhalb von Main {0}", p);
    }
}
```

`ref` kann auch auf Referenzdatentypen angewendet werden. Dort wirkt es sich nur dann aus, wenn an den betreffenden Parameter zugewiesen wird.

```
using System;

public class Program
{
    static void Test1(int [] w)
    {
        w[0] = 22;
        w = new int [] {50,60,70};
    }

    static void Test2(ref int [] w)
    {
        w[0] = 22;
        w = new int [] {50,60,70};
    }
}
```

```

static void Main(string[] args)
{
    //Test1:
    int [] array = new int [] {1,2,3};
    Test1(array);
    Console.WriteLine("Test1() without ref: array[0] = {0}", array[0]);
    //Test2:
    array = new int [] {5,6,7};
    Test2(ref array);
    Console.WriteLine("Test2() with ref : array[0] = {0}", array[0]);
}
}

```

Nur der Vollständigkeit halber sei erwähnt, dass Sie auch unter C# die Pointer-Direktiven wie unter C oder C++ verwenden können. Allerdings müssen Sie Ihre Methoden dann explizit als `unsafe` deklarieren.

```

using System;

public class Program
{
    static unsafe void MIncrement(int* x)
    {
        *x = *x + 1;
    }
    unsafe static void Main(string[] args){
        int i = 42;
        MIncrement(&i);
        Console.WriteLine(i);
    }
}

```

### Ansatz 3 - Übergabe als out-Referenz

`out` erlaubt die Übernahme von Rückgabewerten aus der aufgerufenen Methode.

```

using System;

public class Program
{
    static void Calc(int p, out int output)
    {
        output = p + 1;
    }
    static void Main(string[] args){
        int p = 6, r;
        Calc(p, out r);
        Console.WriteLine(r);
    }
}

```

Interessant wird dieses Konzept durch die in C# 7.0 eingeführte Möglichkeit, dass die Deklaration beim Aufruf selbst erfolgt. Im Zusammenhang mit impliziten Variablen Deklarationen kann man dann typunabhängig Rückgabewerte aus Funktionen entgegennehmen.

```

using System;

public class Program
{
    static void Calc(int p, out int output)
    {
        output = p + 1;
    }
    static void Main(string[] args){
        int p = 6;

```

```

    Calc(p, out int r);
    Console.WriteLine(r);
}

<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
<OutputType>Exe</OutputType>
<TargetFramework>net5.0</TargetFramework>
</PropertyGroup>
</Project>

```

Zudem sollte für eine sehr umfangreiches Set von Rückgabewerten geprüft werden, ob diese wirklich alle benötigt werden. Mit dem *discard* Platzhalter `out _` werden unnötige Deklarationen eingespart.

```

// Defintion
static void SuperComplexMethod(out string result,
                                out int countA,
                                out int countB)
{
    // super complex
}

// Aufruf der Methode
SuperComplexMethod(out _, out _, out int count);

```

## Parameterlisten

C# erlaubt es Methoden zu definieren, die eine variable Zahl von Parametern haben. Dabei wird der letzte Parameter als Array deklariert, so dass die Informationen dann systematisch zu evaluieren sind. Dafür wird der `params` Modifikator eingefügt.

```

using System;

public class Program
{
    static void Add(out int sum, params int [] list)
    {
        sum = 0;
        foreach(int i in list) sum+=i;
    }
    static void Main(string[] args){
        int sum = 0;
        Add(out sum, 3, 3, 5, 6);
        Console.WriteLine(sum);
    }
}

```

Letztendlich wird damit eine Funktionalität realisiert, wie sie für `Main(string[] args)` obligatorisch ist.

## Benannte und optionale Argumente

Funktionsdeklarationen können mit Default-Werten spezifiziert werden. Dadurch wird auf der einen Seite Flexibilität über ein breites Interface garantiert, auf der anderen aber lästige Tipparbeit vermieden. Der Code bleibt damit übersichtlich.

```

static void Sort(string [] s, int from, int to,
                 bool ascending, bool ignoreCases){}

static void Sort(string [] s,
                int from = 0,
                int to = -1,
                bool ascending = true,
                bool ignoreCases= false){}

```

Die *default*-Werte müssen aber der Reihenfolge nach “abgearbeitet” werden. Eine partielle Auswahl bestimmter Werte ist nicht möglich.

```
string [] s = {'Rotkäppchen', 'Hänsel', 'Gretel', 'Hexe'};
//Aufruf      // implizit
Sort(s);      // from=0, to=-1, ascending = true, ignoreCases= false
Sort(s, 3);   // to=-1, ascending = true, ignoreCases= false
```

Darüber hinaus lässt sich die Reihenfolge der Parameter aber auch auflösen. Der Variablenname wird dann explizit angegeben **variablenname:Wert**,

```
PrintDate(1, year:2019, month:12);

->

using System;

public class Program
{
    static void PrintDate(int day=1111, int month=2222, int year=3333 ){
        Console.WriteLine("Day {0} Month {1} Year {2}", day, month, year);
    }

    static void Main(string[] args){
        PrintDate(year:2019);
    }
}
```

## Überladen von Funktionen

Innerhalb der Konzepte von C# ist es explizit vorgesehen, dass Methoden gleichen Namens auftreten, wenn diese sich in ihren Parametern unterscheiden:

- Anzahl der Parameter
- Parametertypen
- Parameterattribute (ref, out)

Ein bereits mehrfach genutztes Beispiel dafür ist die `System.WriteLine`-Methode, die unabhängig vom Typ der übergebenen Variable eine entsprechende Ausgabe realisiert.

```
using System;

public class Program
{
    static int Calc(int a, int b){
        return a/b;
    }

    static float Calc(float a, float b){
        return a/b;
    }

    static void Main(string[] args){
        int a = 5, b= 2;
        float c =5.0f, d=2.0f;
        Console.WriteLine(Calc(a, b));
        Console.WriteLine(Calc(c, d));
    }
}
```

## Aufgaben

- [ ] Wenden Sie die Möglichkeit der Strukturierung des Codes in Funktionen auf die Aufgabe der letzten Vorlesung an. Integrieren Sie zum Beispiel eine Funktion, die den Algorithmus erkennt und eine andere,

die die Operanden einliest.

**Bemühen Sie sich für die Übungsaufgaben Lösungen vor der Veranstaltung zu realisieren, um dort über Varianten möglicher Lösungen zu sprechen!**



# Chapter 8

## OOP Motivation

---

Parameter	Kursinformationen
Veranstaltung	Vorlesung Softwareentwicklung
Semester	Sommersemester 2022
Hochschule:	Technische Universität Freiberg
Inhalte:	Einführung der Konzepte OOP am Beispiel von Structs
Link auf den GitHub:	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/07_OOPGrundlagenI.md">@author</a>

---

## Structs

Ein struct-Typ ist ein Werttyp, der in der Regel verwendet wird, um eine Gruppe verwandter Variablen zusammenzufassen. Beispiele dafür können sein:

- kartesische Koordinaten (x, y, z)
- Merkmale eines Produktes (Größe, Name, Preis)
- Charakteristik einer Datei (Speicherort, Name, Größe, Rechteinformationen)

Ausgangspunkt für die weiteren Überlegungen ist die Konfiguration von `structs` in C.

Machen Sie sich, wenn Ihnen hier die Grundidee noch unklar ist mit den `structs` anhand weiterer Materialien vertraut.

!/?Einführung in Structs

```
#include <stdio.h>
#include <stdlib.h>

typedef struct RectangleStructure{
    double length;
    double width;
    double area;
} Rectangle;

int main(int argc, char const *argv[])
{
    Rectangle rect1, rect2, rect3;
    // Initialisierung:
    rect1.length = 2.5; rect1.width = 5;
    rect2.length = 4; rect2.width = 8;
    rect3.length = 1.5; rect3.width = 2.1;
    // Berechnung:
}
```

```

rect1.area = rect1.length * rect1.width;
rect2.area = rect2.length * rect2.width;
rect3.area = rect3.length * rect3.width;
// Ausgabe:
printf("Rectangle 1 has an area of %.1f\n",rect1.area);
printf("Rectangle 2 has an area of %.1f\n",rect2.area);
printf("Rectangle 3 has an area of %.1f\n",rect3.area);
return 0;
}

```

Wir können also Daten unterschiedlichen Typs situationsspezifisch zusammenfassen ...  
Was fehlt uns?

Richtig, ein Set zugehöriger Funktionen!

```

#include <stdio.h>
#include <stdlib.h>

typedef struct RectangleStructure{
    double length;
    double width;
    double area;
} Rectangle;

void initializeRectangle(Rectangle *actualRect, double length,
                        double width){
    actualRect->length = length;
    actualRect->width = width;
}

void computeArea(Rectangle *actualRect){
    actualRect->area = actualRect->length * actualRect->width;
}

void printRectangleArea(Rectangle *actualRect){
    printf("The Rectangle has an area of %f\n",actualRect->area);
}

int main(int argc, char const *argv[])
{
    Rectangle rect1;
    // Initialisierung:
    initializeRectangle(&rect1, 30, 40);
    computeArea(&rect1);
    printRectangleArea(&rect1);
    return 0;
}

```

C sieht keine Möglichkeit vor Funktion und Daten “dichter” zusammenzubringen, wenn man von Funktionspointern im **struct** absieht.

## Structs in C

Wir fokussieren uns zunächst auf **structs** und übertragen das dabei gewonnene Verständnis dann auf die Klassen. Dabei gilt es einige Unterschiede zu beachten!

Und in C#? Die Sprache erweitert das Konzept in Richtung der Konzepte der objektorientierten Programmierung und integriert Operationen über den Daten in das **struct**-Konzept. Dabei können sie folgende Elemente umfassen:

- Felder und Konstanten
- Methoden
- Konstruktoren und Destruktoren
- Eigenschaften(Properties)

- Indexer
- Events
- überladene Operatoren
- geschachtelte Typen

Konzentrieren wir im ersten Beispiel auf die Felder und die Methoden. Wie sieht eine entsprechende Definition des Bauplanes aller Instanzen von `Animal` aus?

```
public struct Animal
{
    public string name;           // Felder / Konstanten
    public string sound;          //
    public void MakeNoise() {      // Methode
        Console.WriteLine($"{name} makes {sound}");
    }
} // <- Keine Semikolon liebe C++ Programmierer!
```

Sowohl die ganze Struktur als auch die einzelnen Felder sind mit entsprechenden Sichtbarkeitsattributen versehen. Hier wurde explizit `public` vorgesehen, damit ist `Animal` aber auch alle Elemente uneingeschränkt "sichbar". In der Regel ist das aber nicht gewünscht.

Wie legen wir nun ein Objekt entsprechend der Spezifikation an? Wir haben mit dem `struct` einen neuen Typ definiert. Der Aufruf kann (!) anhand des Formats `<datentyp> Variablenname` erfolgen.

```
using System;

public struct Animal
{
    public string name;           // Felder / Konstanten
    public string sound;          //
    public void MakeNoise() {      // Methode
        Console.WriteLine($"{name} makes {sound}");
    }
}

public class Program
{
    static void Main(string[] args){
        Animal kitty;
        kitty.name = "Kitty";
        kitty.sound = "Miau";
        kitty.MakeNoise();
    }
}
```

Erstellen Sie eine weiteres Objekt vom Typ `Animal` und erstellen Sie eine Kopie von `Kitty`.

## Herausforderung `this`

Warum nutzen einige Beispiele das Schlüsselwort `this`? Es wird verwendet, um auf die aktuelle Instanz der Klasse zu verweisen. Obiges Beispiel kann also auch wie folgt geschrieben werden.

```
using System;

public struct Animal
{
    public string name;
    public string sound;
    public void MakeNoise() {
        Console.WriteLine($"{this.name} makes {this.sound}");
    }

    public void setName(string name) {
        this.name = name;
        this.MakeNoise();
    }
}
```

```

        }
    }

public class Program
{
    static void Main(string[] args){
        Animal kitty;
        kitty.name = "Kitty";
        kitty.sound = "Miau";
        kitty.MakeNoise();
        kitty.setName("Tom");
    }
}

```

Das Verständnis von `this` macht die grundlegende Idee deutlich. Das `struct Animal` ist ein genereller Bauplan. Die Instanzen davon haben ein “Selbstverständnis” in Form der `this` Referenz.

## Konstruktoren

Nun umfasst unsere Datenstruktur möglicherweise - anders als unser Beispiel - eine große Zahl von individuellen Variablen.

```

using System;

public struct Animal
{
    public string name;
    public string sound;
}

public class Program
{
    static void Main(string[] args){
        Animal kitty;
        //kitty.name = "Kitty";    // <- fehlende "manuelle" Initialisierung
        //kitty.sound = "Miau";
        Console.WriteLine(kitty.name);
    }
}

```

Wie sieht unser Speicher nach Zeile 12 aus?

STACK		
	+-----+	
	...	
	+-----+ -.	
kitty.name	undefiniert	
	+-----+   Instanz von Animal	
kitty.sound	undefiniert	
	+-----+ -'	
	...	
	+-----+	

Wie können wir also sicherstellen, dass die Initialisierung vollständig vorgenommen wurde?

Konstruktoren sind spezielle Methoden für die Initialisierung eines Objektes. Sie werden einmalig aufgerufen.

Und wie erfolgt der Aufruf des Konstruktors, einer Funktion, die auf einer Datenstruktur wirkt, die es noch gar nicht gibt? Das Schlüsselwort `new` übernimmt diese Aufgabe für uns.

```

<Type> <Variablenname> = new <Konstruktorsignatur>;
using System;

public struct Animal

```

```

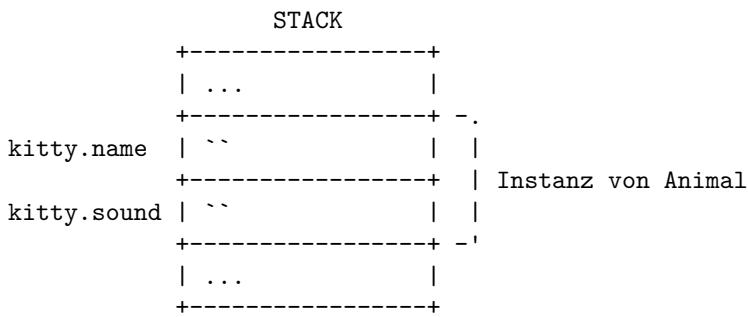
{
    public string name;
    public string sound;

    public void MakeNoise() {
        Console.WriteLine($"->{this.name}<- makes ->{this.sound}<-");
    }
}

public class Program
{
    static void Main(string[] args){
        Animal cat = new Animal();
        cat.MakeNoise();
    }
}

```

Nach dem Aufruf besteht eine mit den Standardwerten der Datentypen vorinitialisierte Instanz auf dem Stack.



In **structs** dürfen allerdings (im Unterschied zu Klassen) keine parameterlosen Methoden sein. Der Compiler erzeugt diese automatisch, die Methode beschreibt alle Felder mit den datentypspezifischen Nullwerten.

Der Standardkonstruktor hilft aber nur bei der Vermeidung von uninitialisierten Werten. In der Regel wollen wir den Feldern aber konkrete Werte zuordnen.

- + Deklaration eines parameterlosen Konstruktors
- + Deklaration eines Konstruktors mit einzelinem Parameter
- + Überladen des Konstruktors

->

```

using System;

public struct Animal
{
    public string name;
    public string sound;

    public Animal(string name, string sound) {
        this.name = name;
        this.sound = sound;
    }

    public void MakeNoise() {
        Console.WriteLine($"{this.name} makes {this.sound}");
    }
}

public class Program
{
    static void Main(string[] args){
        Animal dog = new Animal("Roger", "Wuff");
        dog.MakeNoise();
    }
}

```

```

    }
}

```

*Anmerkung:* Konstruktoren sind Methoden und folglich steht das gesamte Spektrum der Variabilität bei deren Definition zur Verfügung (Überladen, vordefinierte Variablen, Parameterlisten, usw.)

Konstruktor	Aufruf
Aufruf des (impliziten) Standardkonstruktors	Animal kitty = new Animal()
public Animal(name, sound)	Animal kitty = new Animal("kitty", "Miau")
public Animal(name, sound = "Miau")	Animal kitty = new Animal("kitty")

Im Hinblick auf die Verwendung des Schlüsselwortes `new` und die Konsequenzen in Bezug auf die Nutzung des Speichers, sei auf den exzellenten [Beitrag](#) von Clark Kromenaker verwiesen.

In Ergänzung sei auch noch auf die kompakte *Fat Arrow* Darstellung im Zusammenhang mit Konstruktoren, die ja Funktionen wie alle anderen sind verwiesen. Wenn nur eine Anweisung ausgeführt wird kann dies in einer Zeile realisiert werden.

```

using System;

public struct Animal
{
    public string name;
    public Animal(string name) => this.name = name;
    public void MakeNoise() {
        Console.WriteLine("{0} makes Miau", name);
    }
}
public class Program
{
    static void Main(string[] args){
        Animal cat = new Animal("Kitty");
        cat.MakeNoise();
    }
}

<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>
</Project>

```

Ein alternatives Vorgehen bietet die sogenannte Object Initialization Syntax aus C# 3.0 nutzen. Der Compiler generiert den zugehörigen Code für die spezifische Initialisierung.

Dies kann zum Beispiel ein Array mit allen Tieren unseres virtuellen Bauernhofes sein. Wie werden diese dann initialisiert?

```

using System;

public struct Animal
{
    public string name;
    public string sound;
    public void MakeNoise() {
        Console.WriteLine("{0} makes {1}", name, sound);
    }
}
public class Program
{
    static void Main(string[] args){
        Animal[] myAnimals = new Animal[] {

```

```

    new Animal{ name = "Kitty", sound = "Miau"},      // Object Initialization Syntax
    new Animal{ name = "Wally", sound = "Wuff"}, 
    new Animal{ name = "Berta", sound = "Muuh"} 
};

foreach(Animal animal in myAnimals){
    animal.MakeNoise();
}
}

```

**Aufgabe:** Versuchen Sie das Beispiel um einen Konstruktor und einen zugehörigen Aufruf für die Initialisierung zu ergänzen!

## Veränderliche und nicht veränderliche Felder

Aspekt	readonly Felder	const Felder
Zweck	... wird verwendet, um ein schreibgeschütztes Feld zu erstellen.	... wird verwendet, um konstante Felder zu erstellen.
Typ	... ist eine zur Laufzeit definierte Konstante.	... wird verwendet, um eine Konstante zur Kompilierzeit zu erstellen.
Wert	... kann nach der Deklaration geändert werden.	... kann nach der Deklaration nicht geändert werden.
Wertzuweisung	... werden als Instanzvariable deklariert und im Konstruktor mit Werten belegt.	... sind zum Zeitpunkt der Deklaration zuzuweisen.
Einbettung in Methoden	... können nicht innerhalb einer Methode definiert werden.	... können innerhalb einer Methode deklariert werden.

```

using System;

public class Animal
{
    //public const string name = "Kitty";
    public string name = "Kitty";
    public readonly int[] legCount = new int[1]; // Referenzdatentyp (etwas konstruiert)
    public readonly int age; // Wertdatentyp
    //public const long[] hairCount = new int[1]; //geht nicht, da const nicht auf Referenztyp verweisen darf

    public Animal(string name, int age, int legs) {
        this.name = name;
        this.age = age;
        this.legCount[0]= legs;
    }
}

public class Program
{
    static void Main(string[] args){
        Animal kitty = new Animal("Miau", 5, 4);
        kitty.legCount[0]=5;
        Console.WriteLine(kitty.legCount[0]);
    }
}

```

**Achtung:** Der readonly-Modifizierer verhindert, dass das Feld durch eine andere Instanz des Verweistyps ersetzt wird. Der Modifizierer verhindert jedoch nicht, dass die Instanzdaten des Felds durch das schreibgeschützte Feld geändert werden.

**Achtung:** Verwendet man const ist dies jedoch nicht möglich, da eine Konstante nur ein numerischer Typ, ein Boolean, ein String oder ein Enum sein kann. const kann also nicht auf einen Referenztyp verwendet werden (außer string -> Ausnahme) bzw. wenn doch, muss es dann immer null sein.

C# 9 geht einen Schritt weiter und ermöglicht die pauschale Deklaration von readonly structs. Damit wird

der Entwickler beauftragt JEDES Feld entsprechen mit `readonly` zu schützen.

```
using System;

public readonly struct Animal
{
    public string name;
    public string sound;
    public int age;

    public Animal(string name, string sound, int age) {
        this.name = name;
        this.sound = sound;
        this.age = age;
    }
}

public class Program
{
    static void Main(string[] args){
        Animal kitty = new Animal("Kitty", "Miau", 5);
        Console.WriteLine(kitty.sound);
    }
}

<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>
</Project>
```

In der nächsten Vorlesung werden die sogenannten Eigenschaften genauer untersucht. Diese eröffnen nochmals weitere Möglichkeiten der Kapselung.

## Sichtbarkeitsattribute

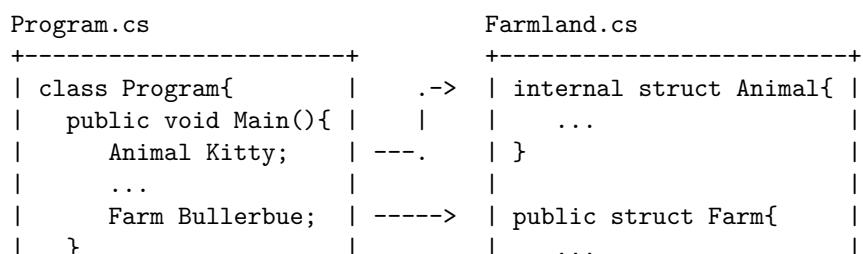
Strukturen und Klassen abstrahieren Daten und verbergen konkrete Realisierungen von Programmteilen. Um zu steuern, welche Elemente eines Programms aus welchem Kontext heraus sichtbar sind, werden diese mit Attributen versehen.

### Sichtbarkeit der Struktur

Strukturen, die innerhalb eines *Namespace* (mit anderen Worten, die nicht in anderen Klassen oder Strukturen geschachtelt sind) direkt deklariert werden, können entweder `private`, `public` oder `internal` sein.

Bezeichner	Konsequenz
<code>public</code>	Keine Einschränkungen für den Zugriff
<code>internal</code>	Der Zugriff ist auf die aktuelle <i>Assembly</i> beschränkt.
<code>private</code>	Der Zugriff kann nur aus dem Code der gleichen Struktur erfolgen.

Wenn kein Modifizierer angegeben ist, wird standardmäßig `internal` verwendet.





```
Schritt 1           mcs -target:library Farmland.cs  
Schritt 2   mcs -reference:Farmland.dll Program.cs
```

Das struct "Animal" soll in einem anderen Assembly nicht aufrufbar sein. Wir wollen die Implementierung kapseln und verbergen. Folglich generiert der entsprechende Aufruf einen Compiler-Fehler. Zugehörige Dateien sind unter [GitHub](#) zu finden.

## Variante 1: Compilieren von Farmland und Programm in ein Assembly

```
mcs Program.cs Farmland.cs  
My name ist Kitty.
```

Variante 2: Compilieren von Farmland als externe Bibliothek

```
mcs -target:library Farmland.cs  
mcs -reference:Farmland.dll Program.cs  
Program.cs(8,13): error CS0122: `Farm.Animal' is inaccessible due to its protection level  
Program.cs(9,26): error CS0841: A local variable `cat' cannot be used before it is declared
```

## Sichtbarkeit der Felder und Member einer Struktur

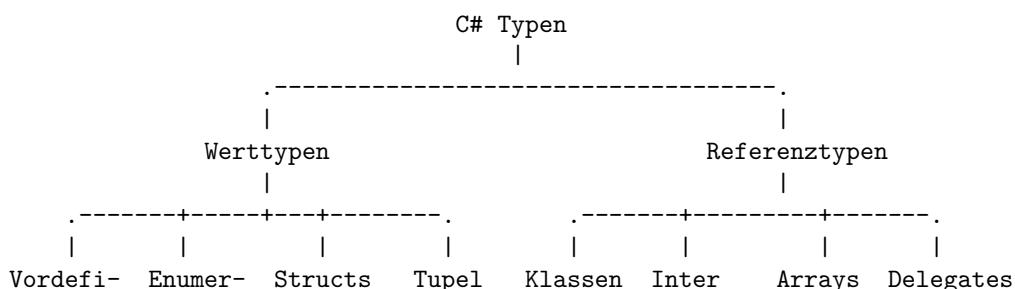
Für die Sichtbarkeit auf der Ebene der Felder und Member können für structs drei Attribute verwendet werden **private**, **internal** und **public**. Standard ist **private**.

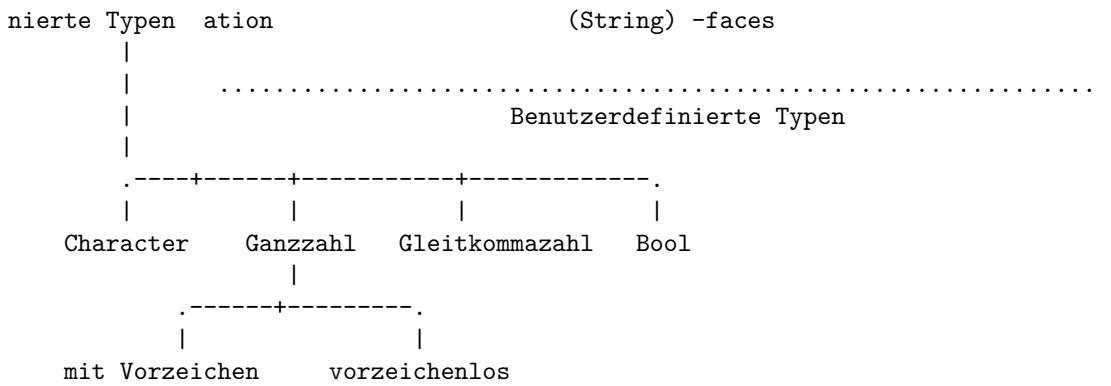
```
using System;
```

```
public struct Animal
{
    public string name;
    internal string sound;
    private byte age;
    public Animal(string name, uint born, string sound = "Miau") {
        this.name = name;
        this.sound = sound;
        age = (byte) (2019 - born);
    }
    public void MakeNoise() {
        Console.WriteLine("{0} ({1} years old) makes {2}", name, age, sound);
    }
}

public class Program
{
    static void Main(string[] args){
        Animal Wally = new Animal ("Wally", 2014, "Wau");
        Wally.MakeNoise();
        Wally.age = 5;
    }
}
```

## Vergleich mit Klassen





Das vorangegangene Beispiel als Klassenimplementierung:

```

using System;

//public struct Animal
public class Animal
{
    public string name;
    private string sound;
    private byte age;
    public Animal(string name, uint born, string sound = "Miau"){
        this.name = name;
        this.sound = sound;
        age = (byte) (2019 - born);
    }
    public void MakeNoise() {
        Console.WriteLine("{0} ({1} years old) makes {2}", name, age, sound);
    }
}

public class Program
{
    static void Main(string[] args){
        Animal Wally = new Animal ("Wally", 2014, "Wau");
        Wally.MakeNoise();
    }
}
  
```

Structs	Klassen
Werttyp (Variablen enthalten das Objektes)	Referenzdatentyp
Abgelegt auf dem Stack	Gespeichert auf dem Heap
Unterstützen keine Vererbung	Unterstützen Vererbung
können Interfaces implementieren	können Interfaces implementieren
<code>internal</code> und <code>public</code> als Struct-Attribute	<code>private</code> , <code>internal</code> und <code>public</code> als Klassenattribute
<code>private</code> , <code>internal</code> und <code>public</code> als Feld und Member-Attribute	analog plus 3 weitere Attribute
keine parameterlosen Konstruktoren deklarierbar	

Welche Konsequenzen hat das zum Beispiel?

**Merke:** Strukturen sind Wertdatentypen!

```

using System;

public class Animal
{
    public string name;
  
```

```

    ...
}

public struct Human
{
    public string name;
    ...
}
}

public class Program
{
    static void Main(string[] args){
        Animal kitty = new Animal(); // <- Referenzvariable
        Human farmer = new Farmer(); // <- Wertvariable
    }
}

```

## Beispiel der Woche

Im folgenden Beispiel werden Instanzen des Structs "Animal" von einem anderen Struct "Farm" genutzt und dort in einer (generischen) Liste gespeichert.

```

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

public struct Animal
{
    public string name;
    public string sound;

    public Animal(string name, string sound = "Miau"){
        this.name = name;
        this.sound = sound;
    }

    public void MakeNoise() {
        Console.WriteLine("{0} makes {1}", name, sound);
    }
}

public struct Farm{
    public string adress;
    public List<Animal> animalList;

    public Farm(string adress) {
        animalList = new List<Animal>();
        this.adress = adress;
    }

    public void AddAnimal(Animal newanimal){
        animalList.Add(newanimal);
    }

    public void PrintAnimals(){
        foreach (Animal pet in animalList){
            pet.MakeNoise();
        }
    }
}

```

```
}
```

```
public class Program
{
    static void Main(string[] args){
        Animal Wally = new Animal ("Wally", "Wau");
        Animal Kitty = new Animal ("Kitty", "Miau");
        Farm myFarm = new Farm("Biobauernhof Freiberg");
        myFarm.AddAnimal(Wally);
        myFarm.AddAnimal(Kitty);
        myFarm.PrintAnimals();
    }
}
```

## Aufgaben

Vollziehen Sie die Überlegungen rund um **structs** nach! Das ist Elementar für das weitere Vorgehen.

!?**Konstruktoren**

!?**Klassen**

Für diejenigen, die bei der Befragung beim letzten mal “Langweilig” ausgewählt hatten, eine Anregung für die Verwendung von C# ...

!?**C# und Unity**

# Chapter 9

# OOP Konzepte I

---

Parameter Kursinformationen

---

Veranstaltung Vorlesung Softwareentwicklung  
Semester Sommersemester 2022  
Hochschule: Technische Universität Freiberg  
Inhalte: Konzepte OOP Programmierung und Umsetzung in C#  
Link auf [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/08\\_OOPGrundlagenII.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/08_OOPGrundlagenII.md)  
GitHub:  
Autoren @author

---

## Auf Nachfrage ...

Wann machen private Klassen oder structs Sinn?

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

public struct Farm{
    public string adress;
    public List<Animal> animalList;

    public Farm(string adress) {
        animalList = new List<Animal>();
        this.adress = adress;
    }

    public void AddAnimal(string name, string sound){
        animalList.Add(new Animal(name, sound));
    }

    public void PrintAnimals(){
        foreach (Animal pet in animalList){
            pet.MakeNoise();
        }
    }

    private struct Animal
    {
        public string name;
```

```

public string sound;

public Animal(string name, string sound = "Miau"){
    this.name = name;
    this.sound = sound;
}

public void MakeNoise() {
    Console.WriteLine("{0} makes {1}", name, sound);
}
}

public class Program
{
    static void Main(string[] args){
        Farm myFarm = new Farm("Biobauernhof Freiberg");
        myFarm.AddAnimal("Wally", "Wau");
        myFarm.AddAnimal("Kitty", "Miau");
        myFarm.PrintAnimals();
    }
}

```

Wie verhält es sich mit mehreren Dateien in einem Ordner und wie stellt man die Relationen zwischen separaten Assemblies her?

```

dotnet new sln -o assemblies_dotnet
cd assemblies_dotnet
dotnet new console -o MyApp
dotnet new classlib -o MyClass
dotnet sln add MyApp
dotnet sln add MyClass
cd MyApp
dotnet add reference ../MyClass

```

Kopieren Sie noch die Dateien aus dem [Mono Verzeichnis](#) in die entsprechenden Ordner.

Starten Sie die Kompilierung, in dem Sie `dotnet run` im Ordner `MyApp` aufrufen. Was beobachten Sie?

## Visionen der Objektorientierung

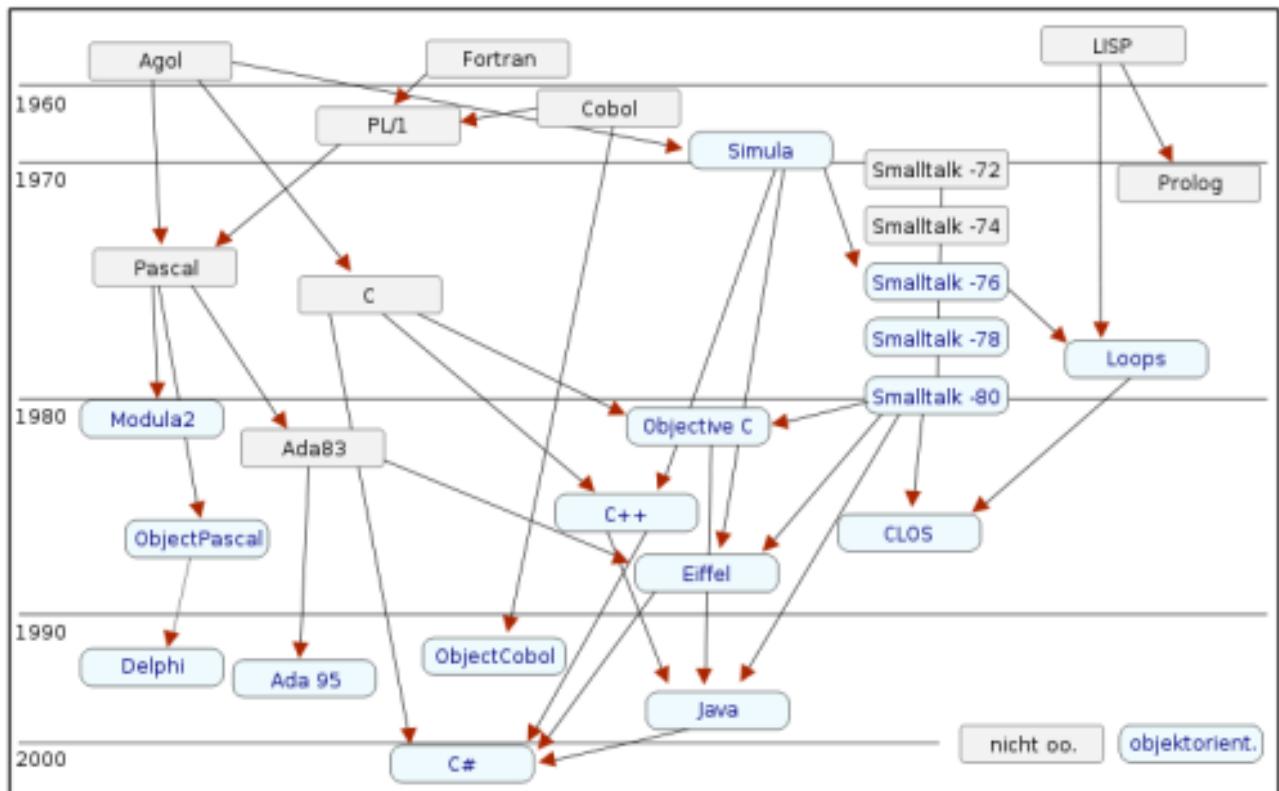
Ein Objekt ist ein Bestandteil eines Programms, der Zustände enthalten kann. Diese Zustände werden von dem Objekt vor einem Zugriff von außen versteckt und damit geschützt. Außerdem stellt ein Objekt anderen Objekten Operationen zur Verfügung. Von außen kann dabei auf das Objekt ausschließlich zugegriffen werden, indem eine Operation auf dem Objekt aufgerufen wird. Ein Objekt legt dabei selbst fest, wie es auf den Aufruf einer Operation reagiert. Die Reaktion kann in Änderungen des eigenen Zustands oder dem Aufruf von Operationen auf weiteren Objekte bestehen.

**Merke** Ein Objekt ist eine zur Ausführungszeit vorhandene und für ihre Member Speicher allozierende Instanz, die sich aus der Spezifikation einer Klasse erschließt.

Ideen der OOP:

- \* Objekte der *realen Welt* müssen sich in der Programmierung widerspiegeln
- \* Es geht nicht um das Manipulieren von Daten, sondern um Zustandsänderungen von Objekten
- \* Im Zentrum der objektorientierten Programmierung stehen Objekte, die miteinander kommunizieren

**Merke** Wir haben zwei Herausforderungen zu meistern - Modellierung und Realisierung.



### Beispiel - Simulationsumgebung Fußballspiel:

- 1 Objekt vom Typ "Spielsituation"
- 1 Objekt vom Typ "Ball"
- 2 Objekte vom Typ "Trainer"
- 3 Objekte vom Typ "Schiedsrichter"
- 22 Objekte vom Typ "Fußballspieler"

### Welche Eigenschaften hat jedes Objekt des Typen "Spieler", "Trainer" bzw. "Schiedsrichter"?

- Name, Alter, Geschlecht, Gewicht, Größe
- Position (x, y, z),
- im Spiel, Geschwindigkeit
- Mannschaft, Rolle (Stürmer, Tormann, Verteidiger), Nummer
- physischer Zustand (topfit, ausgepowert, verletzt)

Einige der Eigenschaften ...

- ... ändern sich im Spielkontext, andere bleiben konstant
- ... lassen sich durchaus allen Personen zuordnen, anderen nur spezifischen Kategorien von Beteiligten.

### Welche Eigenschaften und Methoden (Fähigkeiten) sind für die Instanzen aller Teilnehmer gleich??

- Name, Alter, Geschlecht, Gewicht, Größe
- physischer Zustand (topfit, ausgepowert, verletzt)
- ändertPosition()

### Welche Eigenschaften und Methoden (Fähigkeiten) sind unterschiedlich??

- Rolle in der Mannschaft und Trikotnummer gibt es nur für Spieler
- Mitglied einer Mannschaft bezieht Spieler und Trainer mit ein
- ...

### Welche Methoden sollten dem Objekt "Spieler" erlaubt sein und wie verändert dies deren Zustand?

- FängtDenBall() -> Wirkt sich auf den Zustand von Ball aus, die Position des Balles ist identisch mit der des Spielers ... und es gibt nur einen Ball!
- WirftDenBall()

- Foul(Spieler gefoulterSpieler) -> Wirkt sich auf die Fitness von gefoulterSpieler aus

Welche Schwachstellen sehen Sie bei unserem Modellierungsansatz / der Realisierung?

## Kapselung

Die Verkapselung bezieht sich auf die "Einhüllung" von Daten und Methoden innerhalb einer Struktur (Klasse), die die Objektimplementierung verbirgt und den unmittelbaren Datenzugriff außerhalb vorbestimmter Dienste unterbindet.

Vom Innenleben einer Klasse soll der Verwender – gemeint sind sowohl die Algorithmen, die mit der Klasse arbeiten, als auch der Programmierer, der diese entwickelt – möglichst wenig wissen müssen (Geheimnisprinzip). Durch die Kapselung werden nur Angaben über das „Was“ (Funktionsweise) einer Klasse nach außen sichtbar, nicht aber das „Wie“ (die interne Darstellung).

Standardidentifier für Daten- und Methodenzugriffe sind dabei:

UML		
Bezeichner	Kürzel	Bedeutung
public	+	Zugreifbar für alle Objekte (auch die anderer Klassen)
private	-	Nur für Objekte der eigenen Klasse zugreifbar
protected	#	Nur für Objekte der eigenen Klasse und von Spezialisierungen derselben zugreifbar
internal		Der Zugriff ist auf die aktuelle Assembly beschränkt

## Vorteile

- Da die Implementierung einer Klasse anderen Klassen nicht bekannt ist, kann ihre Implementierung geändert werden, ohne die Zusammenarbeit mit anderen Klassen zu beeinträchtigen.
- Beim Zugriff über eine Zugriffsfunktion spielt es von außen keine Rolle, ob diese Funktion komplett im Inneren der Klasse existiert, das Ergebnis einer Berechnung ist oder möglicherweise aus anderen Quellen (z. B. einer Datei oder Datenbank) stammt.
- Es ergibt sich eine erhöhte Übersichtlichkeit, da nur die öffentliche Schnittstelle einer Klasse betrachtet werden muss.
- Deutlich verbesserte Testbarkeit, Stabilität und Änderbarkeit der Software bzw. deren Teile (Module).

## Nachteile

- In Abhängigkeit vom Anwendungsfall Geschwindigkeitseinbußen durch den Aufruf von Zugriffsfunktionen (direkter Zugriff auf die Datenelemente wäre schneller).
- Zusätzlicher Programmieraufwand für die Erstellung von Zugriffsfunktionen.

## Beispiel Fußballsimulation

1. Die Position x,y eines jeden Spielers und des Balls sollte nur über entsprechende Zugriffsmethoden manipuliert werden.
2. Die Foul kann nur aus dem Spieler heraus aufgerufen werden :-)

```

public struct Position{float x; float y};

+-----+
| Spieler          |   public class Spieler{ // oder struct !!!
+-----+           |       enum Rolle {Stürmer, Tormann, Verteidiger};
| - name: string  |           private string Name;           |
| - alter: byte   |           private byte? Alter;        | Geschützte
| - player: Rolle|           private Rolle player;    | Felder
| ...             |           private Position position; |
+-----+
| FängtDenBall(): void |   public get ... set ...           | Zugriffsmethoden für
| SchießtDenBall(): Kraft |           public void FängtDenBall(Ball);   Felder
| - Foul()           |           public Kraft = SchießtDenBall(Ball);
| ...               |           private Foul(SpielerX);      | Event an SpielerX im
+-----+                   }                                "Erfolgsfall"

```

## Vererbung

Die Vererbung dient dazu, aufbauend auf existierenden Klassen neue zu schaffen. Aus der Klassen-Spezifikation einer Klasse wird eine neue Klasse hergeleitet. Diese ist dann entweder eine Erweiterung oder eine Einschränkung der ursprünglichen Klasse.

Die vererbende Klasse wird meist Basisklasse (auch Super-, Ober- oder Elternklasse) genannt, die erbende abgeleitete Klasse (auch Sub-, Unter- oder Kindklasse). Den Vorgang des Erbens nennt man meist Ableitung oder Spezialisierung.

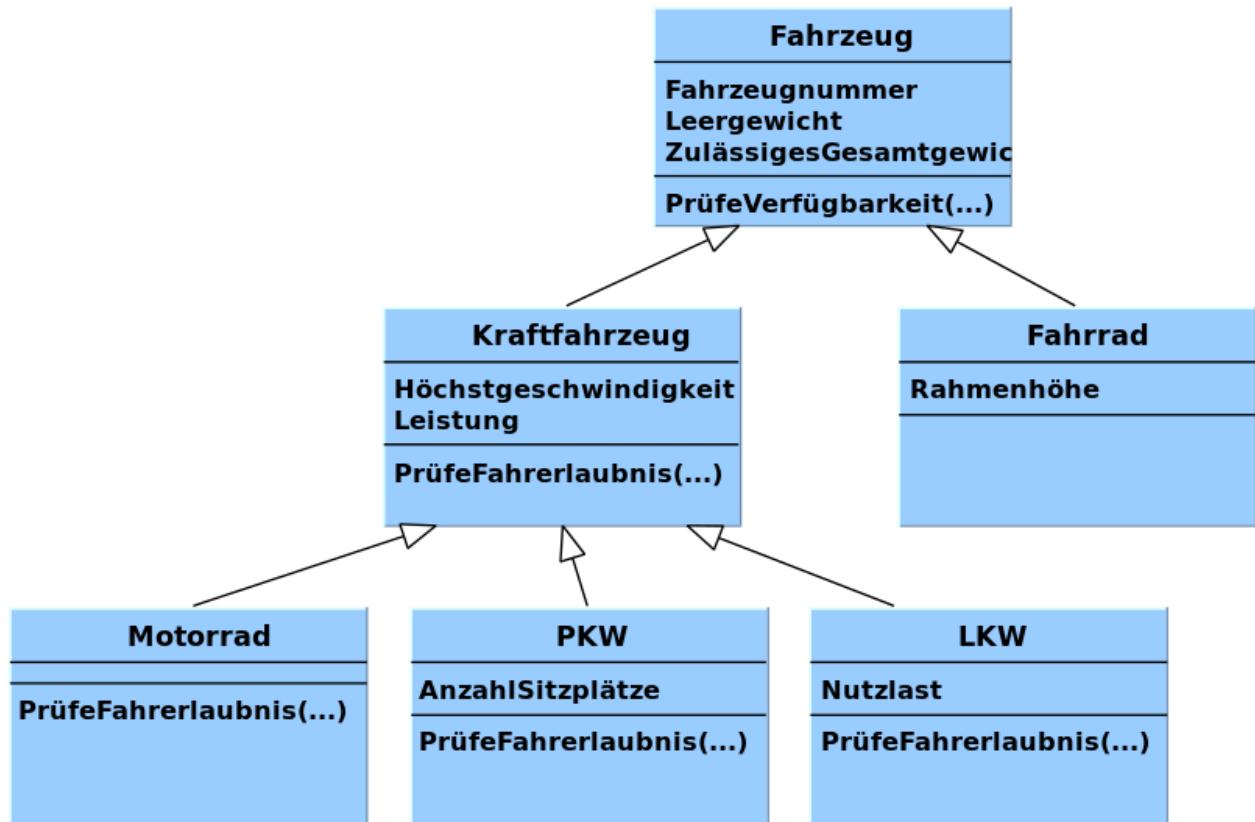


Figure 9.1: Vererbungsbeispiel

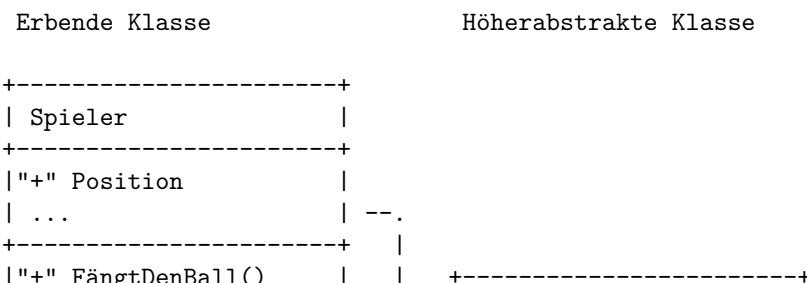
### Vorteile

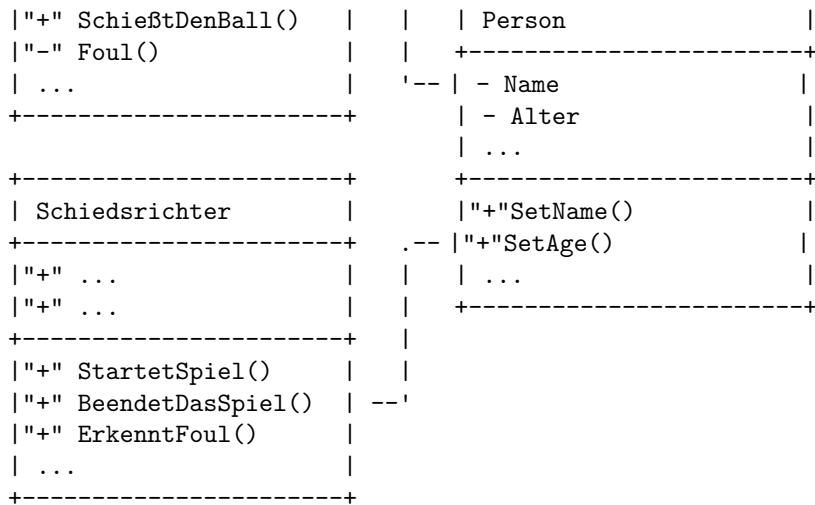
- Abbildung der Eigenschaften und Daten in einem hierarchischen Konzept
- Steigerung der Wartbarkeit und Erweiterbarkeit

### Nachteile

- Eine Klasse, die als Subklasse aus anderen Klassen entsteht, ist kein autonomer Baustein. Bei der Verwendung der Klasse wird es immer wieder zu Rückgriffen auf die Basisklasse(n) kommen.
- Bisweilen schwierige Modellierung
- Schaffung von Abhängigkeitsverhältnissen, die dem Modularisierungsgedanken nicht entsprechen.

### Am Beispiel Fußballspiel





### Zentrale Objekt-Klasse

<https://docs.microsoft.com/de-de/dotnet/api/system.object?view=netframework-4.8>

OOP Sprachen verfügen meist über eine zentrale Klasse, von der alle Klassen in letztlich abgeleitet sind. Diese heißt bei diesen Sprachen Object. In Eiffel wird sie mit ANY bezeichnet. Zu den wenigen Ausnahmen, in denen es keine solche Klasse gibt, zählen C++ oder Python.

In den Sprachen mit zentraler Basisklasse erbt eine Klasse, für die keine Basisklasse angegeben wird, implizit von dieser besonderen Klasse. Ein Vorteil davon ist, dass allgemeine Funktionalität, beispielsweise für die Serialisierung, Ausgaben, Hashwerte oder die Typinformation, dort untergebracht werden kann. Weiterhin ermöglicht es die Deklaration von Variablen, denen ein Objekt jeder beliebigen Klasse zugewiesen werden kann. Dies ist besonders hilfreich zur Implementierung von Containerklassen, wenn eine Sprache keine generische Programmierung unterstützt.

```

using System;

public class Program
{
    static void Main(string[] args){
        Console.WriteLine(typeof(int));
        Console.WriteLine(typeof(int).BaseType);
        Console.WriteLine(typeof(int).BaseType.BaseType);
    }
}

using System;

public class Program
{
    static void Main(string[] args){
        Type t = typeof(Program);
        Console.WriteLine("----- Methods -----");
        System.Reflection.MethodInfo[] methodInfo = t.GetMethods();
        foreach (System.Reflection.MethodInfo mInfo in methodInfo)
            Console.WriteLine(mInfo.ToString());
    }
}

```

### Polymorphie

Polymorphie oder Polymorphismus (griechisch für Vielgestaltigkeit) ermöglicht, dass ein Objekt sich in seiner Funktionalität in Abhängigkeit von den Datentypen verändert.

Die Polymorphie der objektorientierten Programmierung ist eine Eigenschaft, die immer im Zusammenhang mit Vererbung und Schnittstellen (Interfaces) auftritt. Eine Methode ist polymorph, wenn sie in verschiedenen Klassen die gleiche Signatur hat, jedoch erneut implementiert ist.

```

public class Shape
{
    public int X { get; private set; }

    ...
    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}

class Square : Shape {}
...

```

In C# ist jeder Typ polymorph, da alle Typen, einschließlich benutzerdefinierten Typen, von `Object` erben.

Beim Vererben erhält die abgeleitete Klasse alle Methoden, Felder, Eigenschaften und Ereignisse der Basisklasse. Dabei gilt es zu entscheiden, welche davon unverändert übernommen und welche auf die spezifischen Anforderungen angepasst werden sollen.

### Am Beispiel Fußballspiel

```

public class Person{
    private int alter;
    public virtual void setAge(int alter) {
        this.alter = alter;
    }
}

public class Spieler: Person {
    public override void setAge(int alter) {
        // hier wird noch getestet ob der Spieler älter als 16 ist
        // und überhaupt eingesetzt werden darf
    }
}

public class Zuschauer: Person {
    public override void setAge(int alter) {
        // hier wird noch getestet ob ein Zuschauer jünger als 6 ist und
        // kostenlos ins Stadion darf
    }
}

```

### Weitere Beispiele

Beispiel	Kapselung	Vererbung	Polymorphie
Auto	Interne Daten (CAN-Nachrichten zwischen Motor und Getriebe, Zündzeitpunkte, etc.) sind für mich nicht interessant	Kleinwagen und Kombis sind lediglich spezielle Arten von Autos	Ein Maserati Quattroporte und ein C Corsa können beide fahren. Trotzdem sind die Auswirkungen verschieden

Beispiel	Kapselung	Vererbung	Polymorphie
Säuge	Die genauen Vorgänge der Verdauung interessieren nicht. Nur das benötigte Futter ist wichtig	Eine Springmaus und eine Hausratte sind beide aus der Ordnung der Nagetiere. Beide Tiere teilen viele gleiche Eigenschaften	Fortbewegen ist eine Eigenschaft jedes Säugetieres. Ein Wal schwimmt jedoch und ein Känguru hüpfst. Ein Pferd galoppiert.

## Begriffe

Begriff	Bedeutung
Klassen..	sind Vorlagen (Baupläne), aus denen Instanzen Objekte erzeugt werden.
Objekt ...	ist ein Element, welches Funktionen, Methoden, Prozeduren, einen inneren Zustand, oder mehrere dieser Dinge besitzt. Es leitet sich von einer Spezifikation ab.
Entität...	ist ein Objekt, welches eine Identität besitzt, welche unveränderlich ist. Beispielsweise kann eine Person ihre Adresse, Telefonnummer oder Namen ändern, ohne zu einer anderen Person zu werden. Eine Person ist also eine Entität.
Eigenschaft	bestimmt den Zustand eines Objekts. Der Zustand des Objektes setzt sich aus seinen Eigenschaften und Verbindungen zu anderen Objekten zusammen.
Prozedur	verändert den Zustand eines Objektes, ohne einen Rückgabewert zu liefern. Eine Prozedur kann andere Objekte als Parameter entgegennehmen.
Funktion	ordnet einer gegebenen Eingabe einen bestimmten Rückgabewert zu. Eine Funktion zeichnet sich insbesondere dadurch aus, dass sie nicht den Zustand eines Objekts verändert.
Methode	ist ein Unterprogramm (Funktion oder Prozedur), welches das Verhalten von Objekten beschreibt und implementiert. Über Methoden können Objekte untereinander in Verbindung treten.

## Klassen in C

Klassen [und Strukturen] sind zwei der grundlegenden Konstrukte des allgemeinen Typsystems in .NET Framework. Bei beiden handelt es sich um eine Datenstruktur, die einen als logische Einheit zusammengehörenden Satz von Daten und Verhalten kapselt.

Entsprechend können Klassenspezifikationen folgende Elemente umfassen:

Member / Elemente	englische Bezeichnung	Funktion
Felder	<i>fields</i>	Daten
Konstanten	<i>constant</i>	konstante Daten
Eigenschaften	<i>property</i>	Daten und Zugriffsmethoden
Methoden	<i>method</i>	Funktionen / Prozeduren
Konstruktoren	<i>constructor</i>	Instanziierung einer Klasse
Ereignisse	<i>event</i>	Informationsaustausch zwischen Klassen
Finalizer	<i>finalizer</i>	“Destruktoren”
Indexer	<i>indexer</i>	Ähnlich Eigenschaften, Adressierung über Indizes
Operatoren	<i>operators</i>	Set von ‘==’, ‘+’ etc. mit eigener Bedeutung
Geschachtelte Typen	<i>embedded types</i>	Integrierte Klassen oder Structs, die nur innerhalb einer Klasse/ Structs angewendet werden

```
class Person{
    string name;                                // eine häufige Konvention, kleine Anfangs-
    public int Geburtsjahr;                      // buchstaben = privat, groß = public

    public Person(string name, int geburtsjahr){
        this.name = name;
        Geburtsjahr = geburtsjahr;
    }
}
```

```

}

int AktuellesAlter () => DateTime.Today.Year - Geburtsjahr;

public override string ToString(){
    return name + " ist " + AktuellesAlter().ToString() + "Jahre alt."
}

public static bool operator< (Person person1, Person Person2){
    // TODO Hausaufgabe
}
}

```

Und wie legen wir eine Instanz an? Dazu sind mehrere Schritte notwendig:

```

Person p; // Generierung einer Referenzvariablen p auf dem Stack
p = new Person(); // Generierung einer Instanz im Heap
// alles zusammen
// Person p = new Person();

```

Als Operanden erwartet der new-Operator einen Klassennamen und eine Parameterliste, die an den entsprechenden Konstruktor übergeben wird.

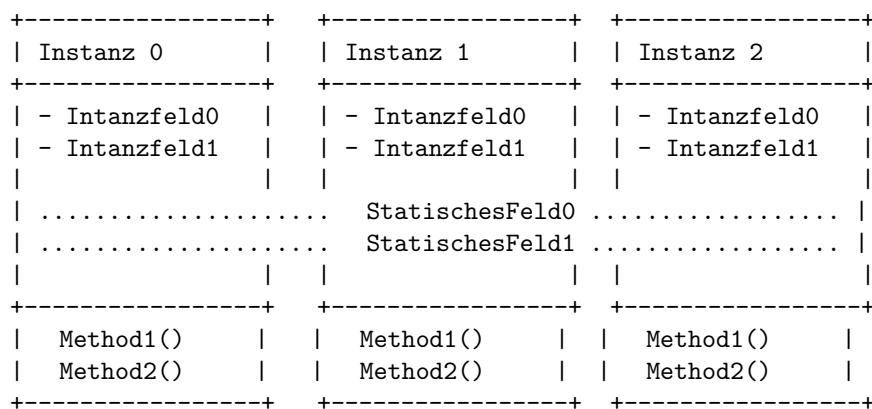
	Fields	Methods
Statistisches Attribut	static	static
Zugriffsattribute	public, internal, private, protected	public, internal, private, protected
Vererbungsattribut	new	new, virtual, abstract, override, sealed
Unsafe Attribute	unsafe	
Attribut		partial
Teilimplementierung		
Unmanaged Code		unsafe extern
Attribute		
Read-only Attribute	readonly	
Threading Attribute	volatile	

## Felder

Felder sind Variablen eines beliebigen Typs, die einer Klasse unmittelbar zugeordnet sind. In Feldern werden die Daten abgelegt, die übergreifend Verwendung finden.

Der Idee der Kapselung folgend, sollten nur methodenlokal relevante Variablen auch dort deklariert werden.

Eine Klasse oder Struktur kann Instanzenfelder, statische Felder oder beides gemischt verfügen.



Instanzenfelder beziehen sich als Datensatz individuell auf die “eigene” Instanz, statisches Felder gehören zur Klasse selbst und werden von allen Instanzen einer Klasse gemeinsam verwendet. “Lokale” Änderungen, werden

somit übergreifend sichtbar.

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

public class Person{
    string name;
    int index;
    public int Geburtsjahr;
    public static int Count;           // <- Statische Variable Count
    public Person(string name, int geburtsjahr){
        this.name = name;
        Geburtsjahr = geburtsjahr;
        index = Count;
        Count = Count + 1;
    }
    public override string ToString(){
        return name + " ist die " + (index+1).ToString() + " von " + Count.ToString() + " Personen";
    }
}
public class Program
{
    static void Main(string[] args){
        Person Student1 = new Person("Mickey", 1935);
        Console.WriteLine(Student1);
        Person Student2 = new Person("Donald", 1927);
        Console.WriteLine(Student1);
        Console.WriteLine(Student2);
    }
}
```

Felder können mit der Deklaration oder im Konstruktor initialisiert werden. Des Weiteren kann mit `readonly` der Wert nach dem Ende des Konstruktorabarbeitung geschützt werden. Eine solche Variable kann als static deklariert werden, um zu vermeiden, dass eine entsprechende Zahl von Kopien erstellt wird.

```
public class Person{
    string name;
    int index = 0;
    readonly string Kategorie = "Student";
    readonly string Hochschule;

    public Person(){
        //...
        Hochschule = "TU Freiberg";
        //...
    }
}
```

## Konstanten

Konstanten sind unveränderliche Datensätze, die zur Kompilierzeit(!) bekannt sind und sich danach nicht mehr verändern lassen. Nur die in C# integrierten Typen - einfache Datentypen und string können als `const` deklariert werden.

Varianten “konstanter” Variablen in C#

	Konstante	Readonly	Readonly statisch
Attribute	<code>const</code>	<code>readonly</code>	<code>readonly static</code>
Veränderbar bis ...	Kompilierung	Ende des Konstruktors	Ende des Konstruktors
Statisch	Standard, ja	Nein	Ja
Zugriff	Klasse	Instanz	Instanz

- Versuchen Sie die Variable innerhalb von Main zu manipulieren
- Wechseln Sie readonly gegen const? Welche Anpassungen müssen Sie vornehmen? ->

```
using System;

public class Person{
    public readonly string name;
    public Person(string name){
        this.name = name;
    }
}
public class Program
{
    static void Main(string[] args){
        Person Student1 = new Person("Mickey");
        Console.WriteLine(Student1.name);
    }
}
```

## Konstruktoren

Beim Erzeugen einer Instanz einer **class** oder eines **structs** wird deren Konstruktor aufgerufen. Dieser ist für die Initialisierung der Instanz auf der Zustandsebene verantwortlich. Konstruktoren können überladen werden und verschiedene Signaturen abbilden.

Wenn für eine Klasse kein Konstruktor vorgegeben wird, erstellt der Kompiler standardmäßig einen, der das Objekt instanziert und Membervariablen auf die Standardwerte festlegt.

```
public class Wine
{
    public decimal Price;
    public int Year;

    // public Wine() // <- Implizit vorhanden, kann aber überschrieben werden
    // Standardkonstruktor
    public Wine (decimal price){Price = price;}
    public Wine (decimal price, int year) : this (price) {Year = year;}
}
```

Der Standardkonstruktor wird implizit generiert, wenn kein anderer Konstruktor durch den Entwickler spezifiziert wurde. Sofern das geschieht, steht dieser auch nicht mehr bereit.

```
using System;

public class Animal
{
    public string name;
    public string sound;
    public int age;

    public Animal(string name, string sound, int age) {
        this.name = name;
        this.sound = sound;
        this.age = age;
    }
}

public class Program
{
    static void Main(string[] args){
        Animal kitty = new Animal("Kitty", "Miau", 5);
        Console.WriteLine(kitty.sound);
        Animal tom = new Animal();
```

```

        Console.WriteLine(tom.sound);
    }
}

Ein Konstruktor kann einen anderen Konstruktor der gleichen Klasse über das Schlüsselwort this aufrufen.  

Dabei kann der Aufruf mit oder ohne Parameter erfolgen.

using System;

class Car
{
    public readonly int NumberOfSeats;
    public readonly int MaxSpeed;
    private int CurrentSpeed;

    public Car(int maxSpeed, int numberOfSeats)
    {
        Console.WriteLine("2 arg ctor");
        this.MaxSpeed = maxSpeed;
        this.NumberOfSeats = numberOfSeats;
    }

    public Car(int maxSpeed) : this(maxSpeed, 5)
    {
        Console.WriteLine("1 arg ctor");
    }

    public Car() : this(100)
    {
        Console.WriteLine("0 arg ctor");
    }
}

public class Program
{
    static void Main(string[] args){
        Car myVehicle = new Car(5);
    }
}

```

### Statische Konstruktoren

- ... werden verwendet, um static-Daten zu initialisieren oder um eine bestimmte Aktion auszuführen, die nur einmal ausgeführt werden muss.
- ... können nicht über Zugriffsmodifizierer oder Parameter verfügen.
- ... können nicht vererbt oder überladen werden.
- ... werden automatisch vor dem Erzeugen der ersten Instanz ausgeführt und können nicht direkt aufgerufen werden. Damit hat der Nutzer keine Kontrolle, wann der Konstruktor ausgeführt wird.
- ... werden kein zweites mal aufgerufen, wenn eine Ausnahme ausgelöst wird.

```

using System;

public class BAFStudent
{
    public static string Universität;
    public string NameStudent;
    static BAFStudent(){
        Console.WriteLine("Universität wird initialisiert");
        Universität = "TU BAF Freiberg";
    }
    public BAFStudent(string name){

```

```

        Console.WriteLine("Name wird initialisiert");
        NameStudent = name;
    }
}

public class Program
{
    static void Main(string[] args){
        BAFStudent student0 = new BAFStudent("Humboldt");
        Console.WriteLine("{0,20} - {1,-10}", BAFStudent.Universität, student0.NameStudent);
        BAFStudent student1 = new BAFStudent("Winkler");
        Console.WriteLine("{0,20} - {1,-10}", BAFStudent.Universität, student1.NameStudent);
    }
}

```

Für die Objektinitialisierung besteht neben den Konstruktoren und dem unmittelbaren Zugriff auf die Membervariablen (vermeiden!) die Möglichkeit direkt nach dem Konstruktoraufgriff die Belegung abzubilden.

```

using System;

public class Wine
{
    public decimal Price;
    public int Year;
    public string Vinyard;
    public Wine () {}
    public Wine (decimal price){Price = price;}
    public Wine (decimal price, int year, string vinyard = "Chateau Lafite" ){
        Price = price;
        Year = year;
        Vinyard = vinyard;
    }
    public override string ToString()
    {
        return String.Format("| {0,5} Euro | {1,5} | {2,-18}|", Price, Year, Vinyard );
    }
}

public class Program
{
    static void Main(string[] args){
        // Initialisierung über Standardkonstruktor und direkten Feldzugriff
        Wine bottle0 = new Wine();
        bottle0.Vinyard = "Chateau Latour";
        Console.WriteLine(bottle0);
        // Initialisierung über die Konstruktoren
        Wine bottle1 = new Wine(23);
        Console.WriteLine(bottle1);
        Wine bottle2 = new Wine(3432, 1956);
        Console.WriteLine(bottle2);
        // Initialisierung über Initializer
        Wine bottle3 = new Wine() {Price = 19, Year = 1910};
        Console.WriteLine(bottle3);
    }
}

```

3 Varianten, und was ist nun besser? Der Aufruf über den Konstruktor ermöglicht die Initialisierung von **readonly** Variablen.

Initializer werden als atomare Funktion realisiert, sind damit Thread-sicher, sind damit aber auch schwieriger zu debuggen. Zudem können nur **public** Member damit adressiert werden. An dieser Stelle wird deutlich, dass Initializer ggf. beim schnellen Testen Tipparbeit sparen, in realen Anwendungen aber nicht zum Einsatz kommen sollten.

## Destruktoren / Finalizer

Mit Finalizern (die auch als Destruktoren bezeichnet werden) werden alle erforderlichen endgültigen Bereinigungen durchgeführt, wenn eine Klasseninstanz vom Garbage Collector gesammelt wird.

```
using System;

public class Person
{
    public string name;
    public Person(string name){this.name = name;}
    ~Person() {Console.WriteLine("The {0} destructor is executing.", ToString());}
}

public class Program
{
    static void Main(string[] args)
    {
        Person Student1 = new Person("Mickey");
        Console.WriteLine(Student1.name);
        Console.WriteLine("Aus die Maus!");
    }
}
```

Der Finalizer ruft implizit die Methode `Finalize` aus der Basisklasse des Typs `Objekt` auf.

## Eigenschaften

Eigenschaft (Properties) organisieren den Zugriff auf private Felder über einen flexiblen Mechanismus zum Lesen, Schreiben oder Berechnen des Wertes. Entsprechend können Eigenschaften wie öffentliche Datenmember verwendet werden. Damit wird das Konzept der Kapselung auf effiziente Zugriffsmethoden abgebildet.

Ausgangspunkt:

- Fügen Sie eine Lese / Schreibmethode für die Variable Wochentag ein, die Prüft, ob die Eingabe zwischen Mo = 0 und Freitag = 4 liegt. ->

```
using System;

public class Vorlesung{
    private byte wochentag;
}

public class Program
{
    static void Main(string[] args){
        Vorlesung SoWi = new Vorlesung();
        SoWi.wochentag = 4;
    }
}
```

C# hält, wie andere OOP Sprachen auch dafür eine eigene kompakte Syntax bereit, die Aspekte der Felder und der Methoden kombiniert. Der aufrufende Nutzer sieht eine Feld, der Zugriff kann aber über eine Methode konfiguriert werden. Dabei können durchaus mehrere Eigenschaften auf eine private Variable verweisen.

Für den Benutzer eines Objekts erscheint eine Eigenschaft wie ein Feld; der Zugriff auf die Eigenschaft erfordert dieselbe Syntax.

```
using System;

public class Vorlesung
{
    private byte wochentag;           // Private Variable
    public byte Wochentag             // Öffentliche Variable
    {
```

```

    get { return wochentag; }           // Property accessors
    set {
        if ((value < 7) & (value >= 0))
            wochentag = value;
        else
            Console.WriteLine("Fehlerhafte Eingabe!");
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Vorlesung SoWi = new Vorlesung();
        SoWi.Wochentag = 4;
        Console.WriteLine(SoWi.Wochentag);
    }
}

```

Die Assessoren können beliebig kombiniert werden. Eine Eigenschaft ohne einen `set`-Accessor ist schreibgeschützt. Eine Eigenschaft ohne einen `get`-Accessor ist lesegeschützt.

Zudem lassen sich mit der *Fat Arrow* Notation die Darstellungen wiederum verkürzen. Beispielhaft ist an folgendem Beispiel auch, dass sich die Properties eine vollkommen andere Informationsstruktur bedienen als die eigentlichen privaten Variablen abbilden (= Kapselung).

```

decimal currentPrice, sharesOwned;

public decimal Worth
{
    get => currentPrice * SharesOwned;
    set => sharesOwned = value / currentPrice
}

// Kompakt für

public decimal Worth
{
    get { return currentPrice * SharesOwned; }
    set { sharesOwned = value / currentPrice; }
}

```

`set` verwendet dabei einen impliziten Parameter mit dem Namen `value`, dessen Typ der Typ der Eigenschaft ist.

Und wie sieht es mit dem Zugriffsschutz der Eigenschaften aus? Insbesondere `set` sollte soweit wie möglich eingeschränkt werden. Dafür können `internal`, `private` und `protected` genutzt werden.

Wenn in den Eigenschaftenzugriffsmethoden keine zusätzliche Logik erforderlich ist, bietet sich die Verwendung von automatisch implementierten Eigenschaften.

```
public int CustomerID { get; set; }
```

In diesem Fall erstellt der Compiler ein privates, anonymes, dahinter liegendes Feld, auf das nur über `get` und `set`-Accessoren zugegriffen werden kann.

## Indexer

Indexer bilden die Zugriffsmethodik für Arrays `MyArray[3]` auf Klassen ab, um den Zugriff auf Arrays, Listen oder andere Container zu kapseln. Dabei wird folgende Notation benutzt:

```

string [] words = "Das ist ein beispielhafter Text".Split();
//      Typ der RückgabevARIABLEN
//          /   this Referenz auf das eigene Objekt

```

```
//      /      /      Typ der Indexvariable
//      /      /      Bezeichner der Variable
//      v      v      v
public string this [int index]{
    get {return words[index]; }
    set {words[index] = value; }
}
```

Auch hier wird das Schlüsselwort `value` verwendet, um den Wert zu definieren, der zugewiesen wird.

Indexer müssen nicht durch einen Ganzahlwert indiziert werden, es können auch andere Typen verwendet werden.

```
using System;
```

```
public class Months
{
    string[] months = {"Jan", "Feb", "März", "April", "Mai", "Juni", "Juli",
                        "Aug", "Sep", "Okt", "Nov", "Dez"};

    public string this[byte index]{
        get {return months[index]; }
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Months MonthList = new Months();
        Console.WriteLine(MonthList[5]);
    }
}
```

Was ist der Vorteil der Klasse + Indexer Lösung? Wie würden Sie die Indizierung noch absichern?

## Operatorenüberladung in C#

Operatoren sind ein Set von Tokens, die grundlegende Operationen für Grunddatentypen beschreiben.

```
int a = 4;
int b = 7;
int c = a + b;                                // + Addition

string s1 = "Hello";
string s2 = "World";
string s3 = s1 + " " + s2;                      // + für String Konkatenation
```

Analog zu Methoden werden können Operatoren überladen werden. Entsprechend wird den Operatoren eine spezifische Bedeutung für die Klassen gegeben.

- Operatoren werden in der Klasse überladen
- Operatoren-Überladung ist immer static
- Nutzung des Schlüsselwortes `operator`

## Überladbare Operatoren

Operator	Bedeutung
<code>+, -, !, ~, ++, --, true, false</code>	unäre Operatoren, überladbar
<code>+, -, \*, /, %, &amp;, ^, &lt;&lt;, &gt;&gt;</code>	binäre Operatoren, überladbar
<code>==, !=, &lt;, &gt;, &lt;=, &gt;=</code>	Vergleichsoperatoren, überladbar
<code>[]</code>	nicht überladbar, aber selbe Funktion mit Indexern
<code>()</code>	nicht überladbar, aber mittels custom conversion gleiche Funktionalität

Operator	Bedeutung
<code>+=, -=, \*=, /=, %=, &amp;=, ^=, &lt;&lt;=, &gt;&gt;=</code>	Werden durch die zugehörigen binären Operatoren automatisch überladen

**Beispiel**

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

public class Vector {
    public double X;
    public double Y;

    public Vector (double x, double y){
        this.X = x;
        this.Y = y;
    }

    //public static Vector operator +(Vector p1, Vector p2){
    //    return new Vector(p1.X + p2.X, p1.Y + p2.Y);
    //}

    public static Vector operator -(Vector p1, Vector p2){
        return new Vector(p1.X - p2.X, p1.Y - p2.Y);
    }

    public override string ToString(){
        return "x = " + X.ToString() + ", y = " + Y.ToString();
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Vector a = new Vector (3,4);
        Vector b = new Vector (9,6);
        Console.WriteLine (a+b);
    }
}
```

Die Operatoren `+=` und `-=` werden dabei automatisch mit überladen.

**Merke:** Die Typen der Operanden beim Überladen von Operatoren müssen nicht übereinstimmen!

Nehmen wir an, dass wir eine Skalierung  $r$  unseres Vektors einfügen wollen und dafür dessen Länge manipulieren.

```
// Es müssen beide Varianten implementiert werden!
public static Point operator *(Point p1, double ratio)
{
    new Point(p1.X * ratio, p1.Y * ratio);
}

public static Point operator *(int ratio, Point p1)
{
    new Point(p1.X * ratio, p1.Y * ratio);
}

static void Main(string[] args)
{
    Point ptOne = new Point(100, 100);
```

```

    Point ptTwo = new Point(40, 40);
}
Console.WriteLine((ptOne * 2.5));
Console.WriteLine((1 * ptOne));

```

Unäre Operatoren  $(+, -)$  können in gleicher Art und Weise überschrieben werden.

Wann sind zwei Klasseninstanzen gleich? Müssen alle Inhalte übereinstimmen? Gibt es besondere Felder, deren Übereinstimmung relevanter sind?

class	Felder
Haus	Farbe der Fenster, Markise (ja/nein), Zahl der Räume
Tier	Art, Rasse, Geschlecht
Datei	Typ, Inhalt, Namen

```

using System;
using System.Reflection;
using System.ComponentModel.Design;

public class Vector {
    public double X;
    public double Y;
    public Vector (double x, double y){
        this.X = x;
        this.Y = y;
    }

    public static bool operator ==(Vector p1, Vector p2){
        return (p1.X == p2.X) && (p1.Y == p2.Y);
    }

    public static bool operator !=(Vector p1, Vector p2){
        return (p1.X != p2.X) || (p1.Y != p2.Y);
    }

    //public override bool Equals(object p){
    //    return ???;
    //}
}

public class Program
{
    static void Main(string[] args)
    {
        Vector a = new Vector (3,4);
        Vector b = new Vector (9,6);
        Console.WriteLine (a == b);
    }
}

```

Die unären Operatoren `True` und `False` nehmen eine kleine Sonderrolle ein:

```

public static bool operator true(Point p1) => (p1.X>0) && (p1.Y>0);
public static bool operator false(Point p1) => (p1.X < 0) && (p1.Y < 0);

Point pt1 = new Point(10, 10);
if (pt1) Console.WriteLine("true"); // true

// Point is neither true nor false:
Point pt2 = new Point(10, -10);

```

## Beispiel der Woche ...

Entwickeln Sie eine Klassenstruktur für die Speicherung der Daten eines Studenten.

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

public class Student
{
    private static int globalerZähler;
    private readonly int uid;
    public string Name { get; set; }
    private bool eingeschrieben;
    private List<string> fächer;

    static Student(){
        globalerZähler = 0;
    }

    public Student(string name)
    {
        Name = name;
        Eingeschrieben = true;
        uid = globalerZähler;
        fächer = new List<string>();
        Console.WriteLine("Der Student {0} (Nr. {1}) ist angelegt!", Name, uid);
        globalerZähler++;
    }

    public bool Eingeschrieben
    {
        get {return eingeschrieben;}
        set
        {
            if (eingeschrieben != value)
                eingeschrieben = value;
            else
            {
                if (value) Console.WriteLine("!Student {0} ist schon eingeschrieben!", Name);
                else Console.WriteLine("!Student {0} ist schon exmatrikuliert!", Name);
            }
        }
    }

    public void addTopic(string Fächername){
        fächer.Add(Fächername);
    }

    public void printTopics(){
        Console.WriteLine("Student {0} hat folgende Fächer absolviert:", Name);
        foreach (string topic in fächer){
            Console.Write(topic + " ");
        }
        Console.WriteLine();
    }
}

public class Program
{
    static void Main(string[] args){
        Student student0 = new Student("Humboldt");
    }
}
```

```
student0.addTopic("Softwareentwicklung");
student0.addTopic("Höhere Mathematik I");
student0.addTopic("Prozedurale Programmierung");
student0.printTopics();
student0.Eingeschrieben = true;
}
}
```

## Aufgaben

- [ ] Setzen Sie sich anhand von [Tutorials](#) mit den Konzepten der objektorientierten Programmierung auseinander!

[!?alt-text](#)

# Chapter 10

## Vererbung

---

Parameter Kursinformationen

---

**Veranstaltung:** Vorlesung Softwareentwicklung

Semester Sommersemester 2022

**Hochschule:** Technische Universität Freiberg

**Inhalte:** Implementierung der Vererbung in C#

**Link auf den** [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/09\\_Vererbung.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/09_Vererbung.md)

**GitHub:**

**Autoren** @author

---

### Auf Nachfrage ...

Hinweis auf die Lauffähigkeit der *Fat Arrow Syntax* unter C# 7.0. siehe Beispiel aus Vorlesung 7

### Vererbung in C

Vererbung bildet neben Kapselung und Polymorphie die zentrale Säule des objektorientierten Programmierens. Die Vererbung ermöglicht die Erstellung neuer Klassen, die ein in existierenden Klassen definiertes Verhalten wieder verwenden, erweitern und ändern. [MS.NET Programmierhandbuch]

#### Beispiele

Die Klasse, deren Member vererbt werden, wird **Basisklasse** genannt, die erbende Klasse als **abgeleitete Klasse** bezeichnet.

---

Basisklasse	abgeleitete Klassen	Gemeinsamkeiten
Fahrzeug	Flugzeug, Boot, Automobil	Position, Geschwindigkeit, Zulassungsnummer, Führerscheinpflicht
Datei	Foto, Textdokument, Datenbankauszug	Dateiname, Dateigröße, Speicherort
Nachricht	Email, SMS, Chatmessage	Adressat, Inhalt, Datum der Versendung

---

### Umsetzung in C#

```
using System;
using System.Reflection;
using System.ComponentModel.Design;
```

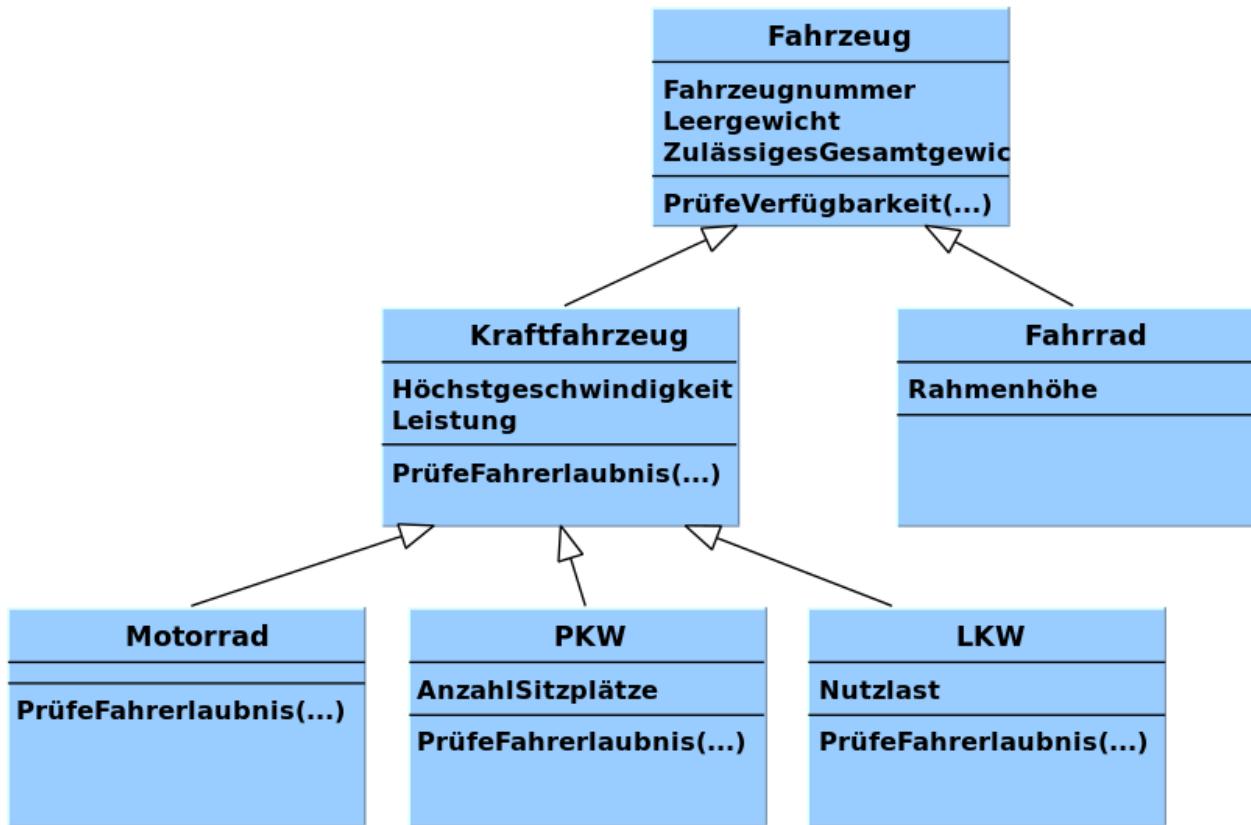


Figure 10.1: Vererbungsbeispiel

```

public class Person {
    public int geburtsjahr;
    public string name;
}

public class Fußballspieler : Person {
    public byte rückenumemr;
}

public class Schiedsrichter : Person {
    public bool assistent = true;
}

public class Program
{
    public static void Main(string[] args){
        Person Mensch = new Person {geburtsjahr = 1956, name = "Löw"};
        Console.WriteLine("{0,4} - {1}", Mensch.geburtsjahr, Mensch.name );
        Console.WriteLine("Felder in der Instanz '{0}' von '{1}'", Mensch.name, Mensch);
        var fields = Mensch.GetType().GetFields();
        foreach (FieldInfo field in fields){
            Console.WriteLine(" x " + field.Name);
        }
    }
}
  
```

*Merke:* Im Unterschied zu Klassen ist für Structs unter C# keine Vererbung möglich!

In C# kann jede Klassendefinition nur eine Basisklasse referenzieren. Im Sinne einer realitätsnahen Modellierung wären Mehrfachvererbungen aber durchaus zielführend. Ein Amphibienfahrzeug leitet sich aus den Basisklassen Wasserfahrzeug und Landfahrzeug ab, ein Touchpad integriert die Member von Eingabegerät und Ausgabegerät.

C# verzichtet drauf um Mehrdeutigkeiten und Fehler ausschließen zu können, die aus gleichnamige Membern hervorgehen.

\*\* ... und wie erfolgt die Initialisierung?\*\*

Konstruktoren werden nicht vererbt, jedoch

- kann mit dem Schlüsselwort **base** auf die Konstruktoren der Basisklasse zurückgegriffen werden.
- wird sofern aus der abgeleiteten Klasse kein expliziter Aufruf erfolgt, der Standardkonstruktor der Basisklasse aufgerufen.

Ein Beispiel für den impliziten Aufruf des Standardkonstruktors:

- Fügen Sie einen leeren Standardkonstruktor mit einer Ausgabe in Fußballspieler ein public Fußballspieler(){ Console.WriteLine("ctor of Fußballspieler"); }
- nutzen Sie nun base um den zweiten in Person existierenden Konstruktor zu adressieren. public Fußballspieler() : base(1) ->

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

public class Person {
    public int geburtsjahr;
    public string name;

    public Person(){
        geburtsjahr = 1984;
        name = "Orwell";
        Console.WriteLine("ctor of Person");
    }

    public Person(int auswahl){
        if (auswahl == 1) {name = "Micky Maus";}
        else {name = "Donald Duck";}
    }
}

public class Fußballspieler : Person {
    public byte rückenummer;
}

public class Program
{
    public static void Main(string[] args){
        Fußballspieler champ = new Fußballspieler();
        Console.WriteLine("{0,4} - {1}", champ.geburtsjahr, champ.name );
    }
}
```

## Zugriffsmechanismen

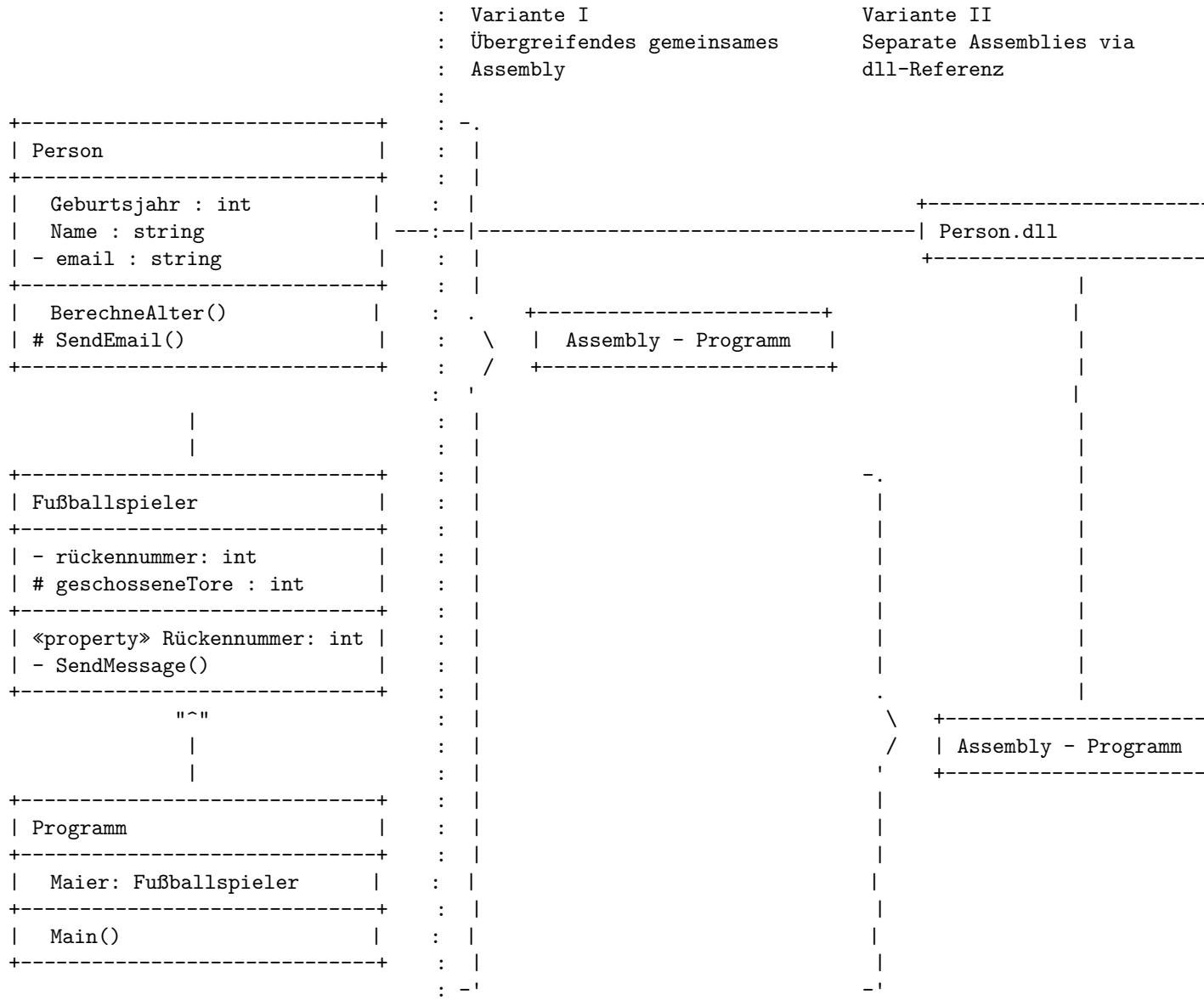
Wer darf auf welche Methoden, Properties, Variablen usw. zurückgreifen? Mit der Einführung der Vererbung steigt die Komplexität der Sichtbarkeitsregeln nochmals an.

Zugriffsmodifizierer	Innerhalb eines Assemblies	Vererbung	Instanzierung	Vererbung	Instanzierung
-----	-----	-----	-----	-----	-----
`public`	ja	ja	ja	ja	ja
`private`	nein	nein	nein	nein	nein
`protected`	ja	nein	ja	ja	nein
`internal`	ja	ja	ja	nein	nein
`internal protected`	ja	ja	ja	ja	nein

`protected` definiert eine differenzierten Zugriff für geerbte und Instanz-Methoden. Während bei geerbten Elementen uneingeschränkt zugegriffen werden kann, bleiben diese bei der bloßenn Anwendung geschützt.

Die Konzepte von `internal` setzen diese Überlegung fort und kontrollieren den Zugriff über Assembly-Grenzen.

## Member der Klasse



Kriterien der Zugriffsattribute:

- innerhalb/außerhalb einer Klasse
- innerhalb der Vererbungshierarchie einer Klasse / außerhalb (“nutzt”)
- innerhalb des Assemblies / außerhalb

Für Methoden, Membervariablen etc. ist das klar, aber macht es Sinn geschützte private Konstruktoren zu definieren?

Private Konstruktoren werden verwendet, um die Instanziierung einer Klasse zu verhindern, die ausschließlich statische Elemente hat. Ein Beispiel dafür ist die `Math` Klasse, die Methoden definiert, die ohne eine Instanz der Klasse aufgerufen werden. Wenn alle Methoden in der Klasse statisch sind, wäre es ggf. sinnvoll die gesamte Klasse statisch anzulegen.

```

using System;

public class Counter
{
    private Counter() { }
}

```

```

    public static int currentCount;

    public static int IncrementCount()
    {
        return ++currentCount;
    }
}

public class Program
{
    public static void Main(string[] args){
        Counter myCounter = new Counter();
        //Console.WriteLine()
    }
}

```

## Klasse

Auch für Klassen selbst können Zugriffsattribute das Verhalten bestimmen:

- Jede Klasse kann entweder als `public` oder `internal` deklariert sein (Standard: `internal`)
- Klassen können mit `sealed` versiegelt werden. Damit ist das Erben davon ausgeschlossen (Bsp.: `System.String`)

## Polymorphie in C

Strukturieren Sie die Klassen “Zug”, “GüterZug”, “PersonenZug” und “ICE” in einer sinnvolle Vererbungshierarchie. Wie setzen Sie diese in C# Code um?

```

using System;
using System.Reflection;
using System.ComponentModel.Design;

class Zug
{
    string nummer;
    public Zug()
    {
        Console.WriteLine("Zug-ctor");
    }
    public Zug(string nummer)
    {
        this.nummer = nummer;
        Console.WriteLine("Spezifischer Zug-ctor");
    }
}

class PersonenZug : Zug
{
    public PersonenZug() : base("Freiberg")
    {
        Console.WriteLine("PersonenZug-ctor");
    }
}

class Ice : PersonenZug
{
    public Ice()
    {
        Console.WriteLine("ICE-ctor");
    }
}

```

```

}

class GueterZug : Zug
{
    public GueterZug()
    {
        Console.WriteLine("GueterZug-ctor");
    }
}

```

```

public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Generiere neuen ICE ");
        Ice ice = new Ice();
        Console.WriteLine("Generieren neuen Güterzug");
        GueterZug gueter = new GueterZug();
    }
}

```

**Merke:** Konstruktoren werden nicht geerbt! Jede Unterklasse deklariert (implizit) eigene Konstrukturen.

Die Konstruktoren der Basisklasse können jeweils mit `base()` aufgerufen werden. Erfolgt dies nicht, wird der parameterlose Konstruktor der Basisklasse automatisch aufgerufen.

Die Ausgabe des oben aufgeführten Beispiels illustriert diese Aufrufhierarchie. Entfernen Sie dem `base` Aufruf in Zeile 23 und erklären Sie den Unterschied.

In diesem Fall ist `Zug` die Basisklasse und `PersonenZug`, `GueterZug` und `ICE` sind abgeleitete Klassen.

Eine Variablen vom Basisdatentyp kann immer eine Instanz einer abgeleiteten Klasse zugewiesen werden. Entsprechend unterscheidet man dann zwischen dem statischen und dem dynamischen Typ der Variablen. Der statische Typ ist immer der, der auch deklariert wurde. Der dynamische Typ wird durch die aktuelle Referenz einer Instanz einer abgeleiteten Klasse von `Zug` bestimmt und ist veränderlich.

Zuweisung	statischer Typ von Zug	dynamischer Typ von Zug
<code>Zug RB51 = new Zug()</code>	<code>Zug</code>	<code>Zug</code>
<code>RB51 = new PersonenZug()</code>	<code>Zug</code>	<code>PersonenZug</code>
<code>RB51 = new Ice</code>	<code>Zug</code>	<code>ICE</code>

## Laufzeitprüfung

Entsprechend brauchen wir eine Typprüfung, die untersucht, ob die Variable von einem bestimmten dynamischen Typ oder einem daraus abgeleiteten Typ ist.

- der dynamische Typ einer Klasse kann zur Laufzeit geprüft werden
- Typtest liefert bei null-Werten immer `false`

```

using System;
using System.Reflection;
using System.ComponentModel.Design;

class Zug
{
    public Zug()
    {
        Console.WriteLine("Zug-ctor");
    }
}

```

```

class PersonenZug : Zug
{
    public PersonenZug() : base()
    {
        Console.WriteLine("PersonenZug-ctor");
    }
}

class Ice : PersonenZug
{
    public Ice()
    {
        Console.WriteLine("ICE-ctor");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Zug IC239 = new Ice();
        Console.WriteLine("IC239 ist ein Zug? " + (IC239 is Zug)); // true
        Console.WriteLine("IC239 ist ein PersonenZug? " + (IC239 is PersonenZug)); // true
        Console.WriteLine("IC239 ist ein Ice? " + (IC239 is Ice)); // true
        IC239 = null;
        Console.WriteLine("IC239 ist ein Ice? " + (IC239 is Ice)); // false
    }
}

```

## Grundidee der Polymorphie

Objekte einer Basisklasse können somit Instanzen einer abgeleiteten Klassen umfassen. Damit lassen sich ähnlich einem Container sehr unterschiedliche Objekte einer Vererbungslinie bündeln. Welche Frage ergibt sich dann?

**Wir haben schon gesehen, dass die Vererbung unter anderem Funktionen umfasst. Auf welche Klassenmember greife ich überhaupt zurück?**

**Merke** Polymorphie bezeichnet die Tatsache, dass Klassenmember ausgehend von Ihrer Nutzung ein unterschiedliches Verhalten erzeugen.

Das heißt, die Methoden der Klassen einer Vererbungshierarchie können auf verschiedenen Ebenen gleiche Signatur, aber unterschiedliche Implementierungen haben. Welche der Methoden für ein gegebenes Objekt aufgerufen wird, wird erst zur Laufzeit bestimmt (dynamische Bindung).

**Merke** Dynamische Bindung bezeichnet die Tatsache, dass bei Aufruf einer überschriebenen Methode über eine Basisklassenreferenz oder ein Interface trotzdem die Implementierung der abgeleiteten Klasse zum Einsatz kommt.

Dynamische Bindung erlaubt den Aufruf von überschriebenen Methoden aus der Basisklasse heraus, wobei das Überschreiben muss in der Basisklasse explizit erlaubt werden muss.

## Überschreiben von Methoden

In C# können abgeleitete Klassen Methoden mit dem gleichen Namen wie Basisklassen-Methoden enthalten. Diese Methoden müssen dann in der Basisklasse mittels `virtual` als explizit überschreibbar deklariert werden:

```
public virtual void makeSound() => Console.WriteLine("I'm an Animal");
```

Zum Überschreiben wird das Schlüsselwort `override` genutzt, welches ein erneutes Deklarieren ermöglicht:

```
public override void makeSound() => Console.WriteLine("Quack!");
```

Dabei müssen beide Methoden die gleiche Signatur haben, d.h. sie sollen die den gleichen Namen und eine identische Parameterliste besitzen. Ansonsten ist es nur Überladung!

```

using System;

class Animal
{
    public string Name;
    public Animal(string name){
        Name = name;
    }
    public virtual void makeSound(){
        Console.WriteLine("I'm an Animal");
    }
}

class Duck : Animal
{
    public Duck(string name) : base(name) { }
    public override void makeSound(){
        Console.WriteLine("{0} - Quack ({1})", Name, this.GetType().Name);
    }
}

class Cow : Animal
{
    public Cow(string name) : base(name) { }
    public override void makeSound(){
        Console.WriteLine("{0} - Muh ({1})", Name, this.GetType().Name);
    }
}

public class Program
{
    public static void Main(string[] args){
        Animal[] animals = new Animal[3]; // <- Statischer Typ Animal
        animals[0] = new Duck("Alfred"); // <- Dynamischer Typ Duck
        animals[1] = new Cow("Hilde");
        animals[2] = new Animal("Bernd");
        foreach (Animal anim in animals)
            anim.makeSound();
    }
}

```

Die verschiedenen Tierklassen werden auf ihre Basisklasse gecastet, trotzdem aber die individuelle Implementierung von `makeSound` ausgeführt. Damit erlaubt die Polymorphie ein gleichartiges Handling unterschiedlicher Klassen, die über die Vererbung miteinander verknüpft sind.

Interessant ist die Möglichkeit die ursprüngliche Implementierung der Methode aus der Basisklasse weiterhin zu nutzen und zu erweitern:

```

class Horse : Animal
{
    public Horse(string name) : base(name) { }
    public override void makeSound()
    {
        base.makeSound();
        Console.WriteLine("Ich ziehe Kutschen");
    }
}

```

Dazu kann die Methode aus der Basisklasse über `base.<Methodenname>` aufgerufen werden

## Verdecken von Methoden

Sollen die spezifischen Methoden aber nur im Kontext der Klasse realisierbar sein, so werden sie vor der Basisklasse "verdeckt". Dazu ist das Schlüsselwort `new` erforderlich. In diesem Fall wird keine dynamische Bindung realisiert, sondern die Methode der Basisklasse aufgerufen.

```
using System;

class Animal
{
    public string Name;
    public Animal(string name){
        Name = name;
    }
    public virtual void makeSound(){
        Console.WriteLine("I'm an Animal");
    }
}

class Cat : Animal
{
    public Cat(string name) : base(name) { }
    public new void makeSound(){
        Console.WriteLine("{0} - Miau ({1})", Name, this.GetType().Name);
    }
}

public class Program
{
    public static void Main(string[] args){
        Cat myCat = new Cat("Kity");
        myCat.makeSound();
        Animal myCatAsAnimal = new Cat("KatziTatzi");
        myCatAsAnimal.makeSound();
    }
}
```

Verdeckt werden können alle Klassenmember einer Basisklasse:

- Felder
- Properties und Indexer
- Methoden usw.

Wenn kein Schlüsselwort angegeben ist, wird implizit `new` angenommen. Im oben genannten Beispiel folgt daraus, dass die in `Cat` implementierte Ausgabe ausschließlich von Objekten des statischen Typs `Cat` aufgerufen werden kann. Testen Sie die Wirkung und ersetzen Sie `new` durch `override`.

Das folgende Beispiel entstammt dem C# Programmierhandbuch und kann unter [Link](#) nachgelesen werden.

Nehmen wir an, dass Ihre Software eine Grafikbibliothek nutzt, die folgende Funktionen bietet:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

Sie haben darauf aufbauend eine umfangreiches Framework geschieben und in einer Klasse, die von `GraphicsClass` erbt eine Methode `DrawRectangle` implementiert.

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

Nun entwickelt der Hersteller eine neue Version von `GraphicsClass` und integriert eine eigene Realisierung von

`DrawRectangle`. Sobald Sie Ihre Anwendung neu gegen die Bibliothek kompilieren, erhalten Sie vom Compiler eine Warnung. Diese Warnung informiert Sie darüber, dass Sie das gewünschte Verhalten der `DrawRectangle`-Methode in Ihrer Anwendung bestimmen müssen. Welche Möglichkeiten haben Sie - override oder new oder umbenennen? Welche Konsequenzen ergeben sich daraus?

## Versiegeln von Klassen oder Membern

Die Mechanismen der Vererbung und Polymorphie können aber auch aufgehoben werden, wenn ein Schutz notwendig ist. Das Schlüsselwort `sealed` ermöglicht es sowohl Klassen von der Rolle als Basisklasse auszuschließen als auch das Überschreiben von Methoden zu verhindern.

```
class A {}
sealed class B : A {}
```

Im Beispiel erbt die Klasse B von der Klasse A, allerdings kann keine Klasse von der Klasse B erben.

**Merke:** Da Strukturen implizit versiegelt sind, können sie nicht geerbt werden.

```
using System;

sealed public class Animal
{
    public string Name;
    public Animal(string name){
        Name = name;
    }
    public virtual void makeSound(){
        Console.WriteLine("I'm a Crocodile");
    }
}

class Cat : Animal
{
    public Cat(string name) : base(name) { }
    public sealed override void makeSound(){ // sealed schützt die Cat.makeSound methode
        Console.WriteLine("{0} - Miau ({1})", Name, this.GetType().Name);
    }
}

class Tiger : Cat
{
    public Tiger(string name) : base(name) { }
    public override void makeSound(){
        Console.WriteLine("{0} - Grrrr ({1})", Name, this.GetType().Name);
    }
}

public class Program
{
    public static void Main(string[] args){
        Tiger evilTiger = new Tiger("Shir Khan");
        evilTiger.makeSound();
    }
}
```

## Casts über Klassen

Konvertierungen zwischen unterschiedlichen Datentypen lassen sich auch auf Klassen anwenden, allerdings sind hier einige Besonderheiten zu beachten.

- implizit auf die Basisklasse (upcast)
- explizit auf die abgeleitete Klasse (downcast)

gecastet werden. Zunächst ein Beispiel für einen *upcast* anhand unseres Fußballbeispiels. Zugriffe auf Member, die in der Basisklasse nicht enthalten sind führen logischerweise zum Fehler.

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

public class Person {
    public int geburtsjahr;
    public string name;
}

public class Fußballspieler : Person {
    public byte rückensnummer;
}

public class Program
{
    public static void Main(string[] args)
    {
        Fußballspieler champ = new Fußballspieler {geburtsjahr = 1956,
                                                    name = "Maier",
                                                    rückensnummer = 13};
        Console.WriteLine("Felder in der Instanz '{0}' von '{1}'", champ.name, champ);
        var fields = champ.GetType().GetFields();
        foreach (FieldInfo field in fields){
            Console.WriteLine(" x " + field.Name);
        }
        Person human = champ;      // Castoperation Fußballspieler -> Person
        Console.WriteLine("human ist ein Fußballspieler? " + (human is Fußballspieler));
        //Console.WriteLine(human.rückensnummer);
    }
}
```

In umgekehrter Richtung vollzieht sich der *Downcast*, eine Instanz der Basisklasse wird auf einen abgeleiteten Typ gemappt.

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

public class Person {
    public int geburtsjahr;
    public string name;
}

public class Fußballspieler : Person {
    public byte rückensnummer;
}

public class Program
{
    public static void Main(string[] args)
    {
        Fußballspieler champ = new Fußballspieler {geburtsjahr = 1956,
                                                    name = "Maier",
                                                    rückensnummer = 13};

        Person mensch = champ;
        Fußballspieler champ2 = (Fußballspieler) mensch;
    }
}
```

## Beispiel

*Upcast* und *Downcast* ... wozu brauche ich das den? Nehmen wir an, dass wir eine Ausgabemethode für beide Typen - Person und Fußballspieler - benötigen. Ja, es wäre möglich diese als Memberfunktion zu implementieren, problematisch wäre aber dann, dass wir an unterschiedlichen Stellen im Code spezifische Befehle für die Ausgabe in der Konsole zu stehen haben. Sollen die Log-Daten nun plötzlich in eine Datei ausgegeben werden, müsste diese Anpassung überall vollzogen werden. Entsprechend ist eine externe (statische) Logger-Klasse wesentlich geeigneter diese Funktionalität zu kapseln. Allerdings wäre dann ein überladen der entsprechenden Ausgabefunktion mit allen vorkommenden Typen notwendig. Dies kann durch entsprechende Casts umgangen werden.

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

public class Person
{
    public int geburtsjahr;
    public string name;
}

public class Fußballspieler : Person
{
    public byte rückenummer;
}

public static class Logger
{
    public static void printPerson(Person person){
        Console.WriteLine("{0} - {1}", person.name, person.geburtsjahr);
        if (person is Fußballspieler)
            Console.WriteLine("{0} - {1}", person.name, (person as Fußballspieler).rückenummer);
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Person Mensch = new Person {geburtsjahr = 1956,
                                    name = "Maier"};
        Logger.printPerson(Mensch);
        Fußballspieler Champ = new Fußballspieler{geburtsjahr = 1967,
                                                name = "Müller",
                                                rückenummer = 13};
        Logger.printPerson(Champ);
    }
}
```

## Aufgaben

- [ ]

# Chapter 11

## Abstrakte Klassen und Interfaces

---

Parameter      Kursinformationen

---

**Veranstaltung:** Vorlesung Softwareentwicklung

**Semester:** Sommersemester 2021

**Hochschule:** Technische Universität Freiberg

**Inhalte:** Konzepte Abstrakter Klassen und Interfaces

**Link auf** [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/10\\_AbstrakteKlassenUndInterfaces.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/10_AbstrakteKlassenUndInterfaces.md)

**den** [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/10\\_AbstrakteKlassenUndInterfaces.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/10_AbstrakteKlassenUndInterfaces.md)

**GitHub:**

**Autoren** @author

---

### Auf Nachfrage ...

Wozu brauche ich das? Bisher bin ich auch gut ohne OOP ausgekommen ...

Nutzung von objektorientierten Konzepten im Python Projekt [github2pandas](#).

### Abstrakte Klassen / Abstrakte Methoden

Mit `virtual` werden einzelne Methoden spezifiziert, die durch die abgeleiteten Klassen implmentiert werden. Die Basisklasse hält aber eine “default” Implementierung bereit. Letztendlich kann man diesen Gedanken konsequent weiter treiben und die Methoden der Basisklasse auf ein reines Muster reduzieren, dass keine eigenen Implementierungen umfasst.

Diese Aufgabe übernehmen abstrakte Klassen und abstrakte Methoden. Eine abstrakte Klasse:

- kann nicht instantiiert werden
- kann abstrakte Methoden umfassen
- ist oft als Startpunkt(e) einer Vererbungshierarchie gedacht sind.

Innerhalb der Klasse können abstrakte Methoden integriert werden, die

- implizit als virtuelle Methode implementiert angelegt werden
- entsprechend keinen Methodenkörper umfassen

Eine nicht abstrakte Klasse, die von einer abstrakten Klasse abgeleitet wurde, muss Implementierungen aller geerbten abstrakten Methoden und Accessoren enthalten.

```
using System;

public abstract class Animal
{
    public string Name;
    public Animal(string name){
```

```

        Name = name;
    }
    public abstract void makeSound();
}

public class Corcodile : Animal{
    public Corcodile(string name) : base(name){
        Name = name;
    }
    public override void makeSound(){
        Console.WriteLine("I'm a Crocodile");
    }
}

public class Program
{
    public static void Main(string[] args){
        Corcodile A = new Corcodile("Tuffy");
        A.makeSound();
    }
}

```

Abstrakte Klassen dienen somit als Template für nachgeordnete Unterklassen. Neben Methoden können auch Properties und Indexer als abstrakt deklariert werden.

Warum macht es keinen Sinn eine abstrakte Klasse als `sealed` zu deklarieren?

## Interfaces

Interfaces setzen die Idee der abstrakten Klassen konsequent fort und umfassen nur abstrakte Member. Sie bilden die Signatur einer Klasse, in der Methoden, Properties, Indexer und Events erfasst werden.

Merke: Interfaces umfassen keine Felder!

Charakteristik von Interfaces:

- alle Bestandteile aus einem Interface müssen implementiert werden
- Klassen „implementieren“ Interfaces und „erben“ von Basisklassen
- Interfaces haben das Schlüsselwort `interface` und fangen im allgemeinen mit dem Buchstaben I an
- alle Elemente sind implizit `abstract` und `public`

```

using System;
using System.Reflection;
using System.ComponentModel.Design;

interface IShape
{
    double Area();
    double Scope();
}

class Rectangular : IShape // Rectangular implementiert das Interface IShape
{
    double area;
    double scope;
    public double Area() => area;
    public double Scope() => scope;
    public Rectangular(double sideA, double sideB)
    {
        area = sideA * sideB;
        scope = 2 * sideA + 2 * sideB;
    }
}

```

```
public class Program
{
    public static void Main(string[] args)
    {
        Rectangular rect = new Rectangular(2, 3);
        Console.WriteLine("Area: {0}, " + "Scope: {1}", rect.Area(), rect.Scope());
    }
}
```

Eine Klasse kann nur von einer anderen Klasse erben, aber beliebig viele Interfaces implementieren.

Schnittstellen werden verwendet:

- um eine lose Kopplung zu erreichen.
- um eine vollständige Abstraktion zu erreichen.
- um komponentenbasierte Programmierung zu erreichen
- um Mehrfachvererbung und Abstraktion zu erreichen.

## Vererbung

```
using System;
using System.Reflection;
using System.ComponentModel.Design;

interface IBaseInterface { void M(); }
interface IDerivedInterface : IBaseInterface { void N(); }

class A : IBaseInterface
{
    public void M()
    {
        Console.WriteLine("Methode M in {0}", this.GetType().Name);
    }
}

class B : IDerivedInterface
{
    public void M()
    {
        Console.WriteLine("Methode M in {0}", this.GetType().Name);
    }
    public void N()
    {
        Console.WriteLine("Methode N in {0}", this.GetType().Name);
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        IBaseInterface t1 = new A();      // Statischer Typ IBaseInterface, dynamischer class A
        IBaseInterface t2 = new B();      // Statischer Typ IBaseInterface, dynamischer class B
        t1.M();
        t2.M();
        Console.WriteLine(t2 is IDerivedInterface);
        (t2 as IDerivedInterface).N();
        (t2 as B).N();
    }
}
```

Es besteht keine Vererbungshierarchie zwischen den beiden Klassen A und B! Vielmehr ergibt sich ein neuer Zusammenhang, die gemeinsame Implementierung eines Musters an Membern.

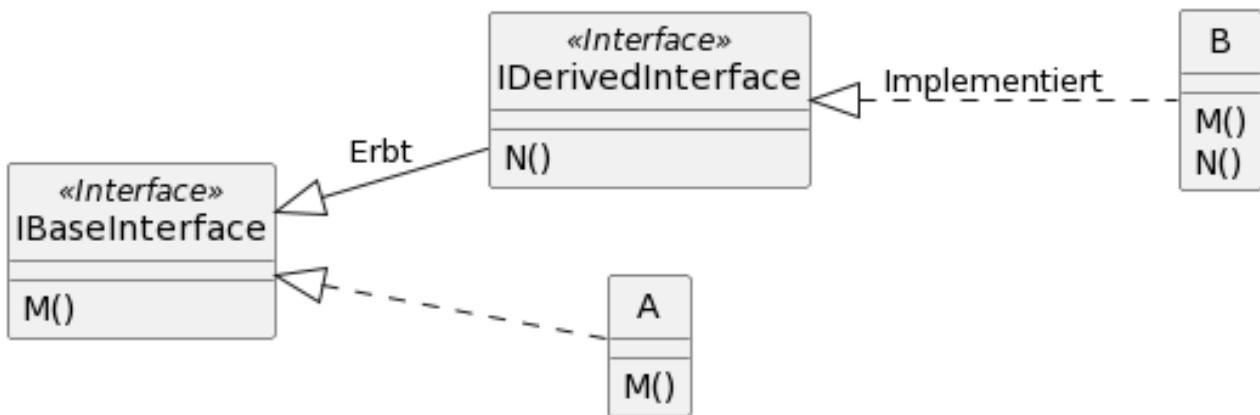


Figure 11.1: ClassStructure

Die Visualisierung von Klassen und deren Abhängigkeiten mit plantUML ist eine Möglichkeit einen raschen Überblick über bestimmte Zusammenhänge zu gewinnen. In den folgenden Materialien wird dies intensiv genutzt.

## Interfaces vs. Abstrakte Klassen

interface	abstract class
viele Interfaces möglich	immer nur eine Basisklasse
speichert keine Daten	kann Felder umfassen
keine KonstruktorenSignaturen	kann Konstruktoren umfassen
beinhaltet nur Methodensignaturen	kann Signaturen und Implementierungen integrieren
keine Zugriffsmodifizierer	beliebige Zugriffsmodifizierer
keine statischen Member	statische Member möglich

Merke: Interfaces geben keine Struktur vor, sondern nur ein Verhalten!

## Bedeutung von Interfaces

Die C# Bibliothek implementiert eine Vielzahl von Interfaces, die insbesondere für die Handhabung von Datenstrukturen in jedem Fall genutzt werden sollten.

Informieren Sie sich unter [Link](#) über die wichtigsten davon wie:

- IEnumurable, IEnuerator
- IList
- IComparable
- ICollection
- ...

```

using System;
using System.Reflection;
using System.ComponentModel.Design;

public class Cat: IComparable
{
    public string Name {get; set;}
    public int CompareTo(object obj)
    {
        if (!(obj is Cat))
        {
            throw new ArgumentException("Compared Object is not of Cat");
        }
        Cat cat = obj as Cat;
        return Name.CompareTo(cat.Name);
    }
}

public class Program
{
}
  
```

## Auflösung von Namenskonflikten

```
using System;

interface IInterfaceA{
    void M();
}

interface IInterfaceB{
    void M();
}

public class SampleClass : IInterfaceA, IInterfaceB
{
    // Hier ist die Zuordnung nicht eindeutig
    public void M()
    {
        Console.WriteLine("Gib irgendwas aus!");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        SampleClass sample = new SampleClass();
        sample.M();
        IInterfaceA A = sample;
        IInterfaceB B = sample;
        A.M();
        B.M();
    }
}
```

Wenn zwei Schnittstellenmember nicht dieselbe Funktion durchführen sollen muss diese separat implementiert werden. Hierzu wird ein Klassenmember erstellt, der sich explizit auf das Interface bezieht und den Namen der Schnittstelle benennt.

```
public class SampleClass : IInterfaceA, IInterfaceB
{
    // Hier ist die Zuordnung nicht eindeutig
    void IInterfaceA.M()
    {
        Console.WriteLine("IInterfaceA - Gib irgendwas aus!");
    }

    void IInterfaceB.M()
    {
        Console.WriteLine("IInterfaceB - Gib irgendwas aus!");
    }
}
```

Allerdings kann diese Funktion dann nur über die Schnittstelle und nicht über die Klasse aufgerufen werden.

## Aufgaben

- [ ] Setzen Sie sich mit den Konzepten von Interfaces auseinander!

!?

!?



# Chapter 12

## Versionsverwaltung I

---

Parameter Kursinformationen

---

**Veranstaltung:** Vorlesung Softwareentwicklung  
**Semester:** Sommersemester 2021  
**Hochschule:** Technische Universität Freiberg  
**Inhalte:** Motivation der Versionsverwaltung in der Softwareentwicklung  
**Link auf den GitHub:** [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/11\\_VersionsverwaltungI.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/11_VersionsverwaltungI.md)  
**Autoren:** @author

---

### Motivation

Was war das umfangreichste Dokument, an dem Sie bisher gearbeitet haben? Bei vielen sicher eine Hausarbeit am Gymnasium. Wie haben Sie Ihren Fortschritt organisiert?

1. Im schlimmsten Fall haben Sie sich gar keine Gedanken gemacht und immer wieder ins gleiche Dokument geschrieben, das in einem Ordner liegt, der alle zugehörigen Dateien umfasst.
2. Eine Spur besser ist die Idee wöchentlich neue Kopien des Ordners anzulegen und diese in etwa so zu benennen:

```
► ls
myProject
myProject_test
myProject_newTest
myProject_Moms_corrections
...
```

3. Wenn Sie “einen Plan hatte”, haben Sie täglich eine Kopie aller Dateien in einem Ordner angelegt und diese systematisch benannt.

```
► ls
myProject_01042021
myProject_02042021
myProject_03042021
...
```

In den Ordner gab es dann aber wieder das gleiche Durcheinander wie in (2), weil Sie bestimmte Texte gern kurzfristig sichern wollten. Teilweise haben sie diese dann gelöscht bevor die Kopie erstellt wurde, meistens aber einfach in einem **sonstiges** Ordner belassen.

Überlegen Sie sich kurz, wie Sie vorgehen müssen, um Antworten auf die folgenden Fragen zu finden:

- “Wann wurde der letzte Stand der Datei x.y gelöscht?”

- “In welcher Version habe ich die Anpassung der Überschriften vorgenommen?”
- “Wie kann ich dies trotz anderer zwischenzeitlicher Änderungen rückgängig machen?”
- “Warum habe ich davon keine Kopie gemacht?”
- “...”

In jedem Fall viel manuelle Arbeit ...

Und nun übertragen wir den Ansatz auf eine Softwareentwicklungsprojekt mit vielen Mitstreitern. Die Herausforderungen potenzieren sich.

1. Die Erstellung der Tageskopie müsste synchron erfolgen.
2. Ich muss in die Ordner schauen, um zu sehen welche Anpassungen vorgenommen wurden.
3. Ich weiß nicht welche die aktuelle Version einer Datei ist.
4. Es existieren plötzlich mehrere Varianten einer Datei mit Änderungen an unterschiedlichen Codezeilen.
5. Ich kann den Code nicht kompilieren, weil einzelne Dateien fehlen.
6. Ich kann eine ältere Version der Software nicht finden - “Gestern hat es noch funktioniert”.
7. Meine Änderungen wurden von einem Mitstreiter einfach überschrieben.

## Lösungsansatz

Eine Versionsverwaltung ist ein System, das zur Erfassung von Änderungen an Dokumenten oder Dateien verwendet wird. Alle Versionen werden in einem Archiv mit Zeitstempel und Benutzerkennung gesichert und können später wiederhergestellt werden. Versionsverwaltungssysteme werden typischerweise in der Softwareentwicklung eingesetzt, um Quelltexte zu verwalten.

Ein Beispiel, wie ein Versionsmanagementsystem die Arbeit von verteilten Autoren unterstützt ist die Implementierung von Wikipedia. Jede Änderung eines Artikels wird dokumentiert. Alle Versionen bilden eine Kette, in der die letzte Version als gültige angezeigt wird. Entsprechend der Angaben kann nachvollzogen werden: wer wann was geändert hat. Damit ist bei Bedarf eine Rückkehr zu früheren Version möglich.

The screenshot shows a detailed view of the Wikipedia version history interface. At the top, there's a navigation bar with links for 'Artikel', 'Diskussion', 'Lesen', 'Bearbeiten', 'Quelltext bearbeiten', 'Versionsgeschichte', and a search bar. Below the navigation, the main title is '„Versionsverwaltung“ – Versionsgeschichte'. A sub-section titled 'Versionsgeschichte eingrenzen' is visible. The central part of the page displays a table of previous edits ('Alte Versionen des Artikels') with columns for 'Zeige' (Show), 'Zeige (nächste 50)', 'vorherige 50)', '(20 | 50 | 100 | 250 | 500)', and '(neueste | älteste)'. Each row represents an edit with columns for 'Aktuell' (Current), 'Vorherige' (Previous), 'Zeit' (Time), 'IP-Adresse' (IP Address), 'Benutzername' (Username), 'Diskussion' (Discussion link), 'Größe' (Size), 'Änderungen' (Changes), and 'Markierung' (Marker). The interface also includes a 'Gewählte Versionen vergleichen' (Compare Selected Versions) section and a 'Logbücher dieser Seite anzeigen' (View Logbooks) link.

Figure 12.1: Versionsmanagement Wikipedia

Hauptaufgaben:

- Protokollierungen der Änderungen: Es kann jederzeit nachvollzogen werden, wer wann was geändert hat.
- Wiederherstellung von alten Ständen einzelner Dateien: Somit können versehentliche Änderungen jederzeit wieder rückgängig gemacht werden.
- Archivierung der einzelnen Stände eines Projektes: Dadurch ist es jederzeit möglich, auf alle Versionen zuzugreifen.
- Koordinierung des gemeinsamen Zugriffs von mehreren Entwicklern auf die Dateien.
- Gleichzeitige Entwicklung mehrerer Entwicklungszweige (engl. Branch) eines Projektes, was nicht mit der Abspaltung eines anderen Projekts (engl. Fork) verwechselt werden darf.

## Strategien zur Konfliktvermeidung

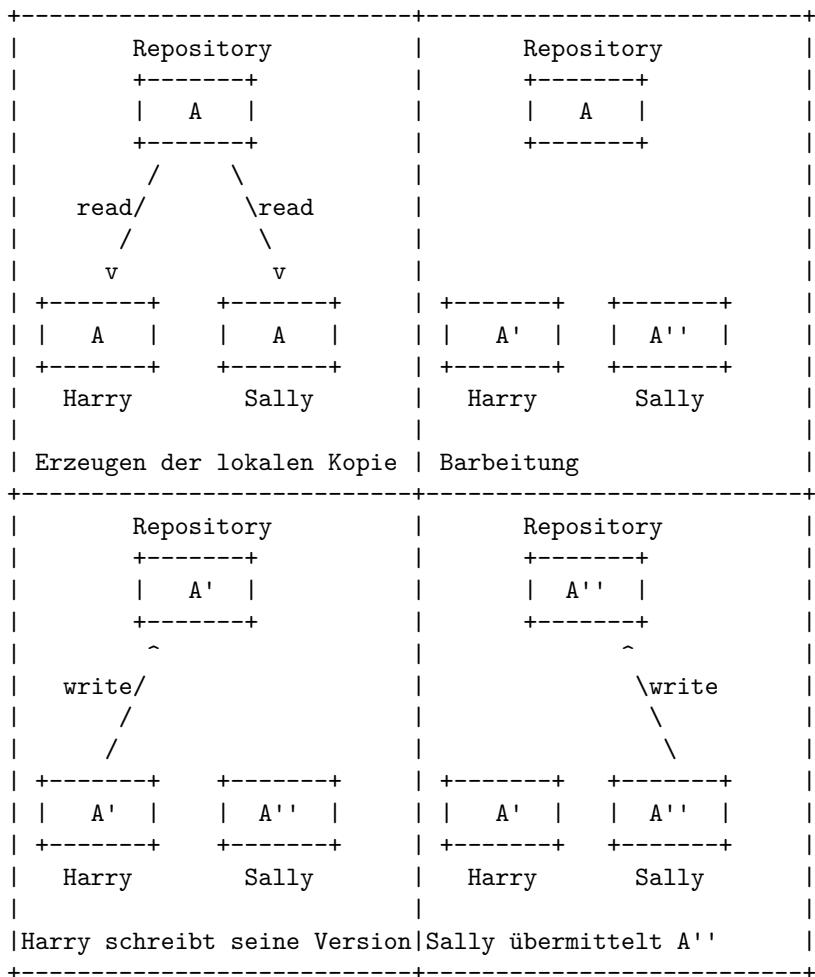
### Herausforderung

Das Beispiel entstammt dem Buch *Version Control with Subversion*<sup>1</sup>

Zwei Nutzer (Harry und Sally) arbeiten am gleichen Dokument (A), das auf einem zentralen Server liegt:

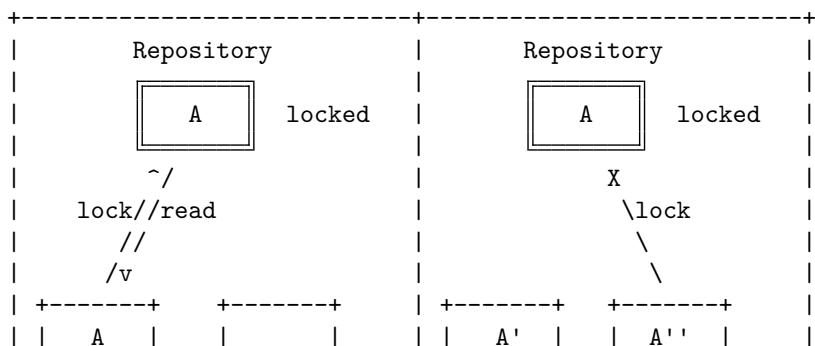
- Beide führen verschiedene Änderungen an ihren lokalen Versionen des Dokuments durch.
- Die lokalen Versionen werden nacheinander in das Repository geschrieben.
- Sally überschreibt dadurch eventuell Änderungen von Harry.

Die zeitliche Abfolge der Schreibzugriffe bestimmt welche Variante des Dokuments A überlebt.

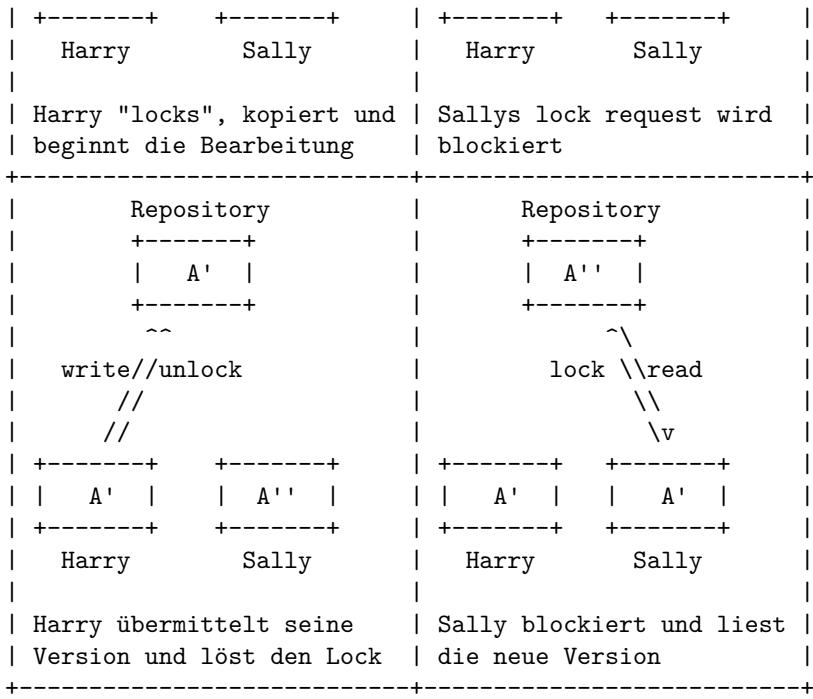


### Lösung I - Exklusives Bearbeiten (Sequenzialisierung)

Bei der pessimistischen Versionsverwaltung (*Lock Modify Unlock*) werden einzelne Dateien vor einer Änderung durch den Benutzer gesperrt und nach Abschluss der Änderung wieder freigegeben werden. Während sie gesperrt sind, verhindert das System Änderungen durch andere Benutzer. Der Vorteil dieses Konzeptes ist, dass kein Zusammenführen von Versionen erforderlich ist, da nur immer ein Entwickler eine Datei ändern kann.



<sup>1</sup>Brian W. Fitzpatrick, Ben Collins-Sussman, C. Michael Pilato, *Version Control with Subversion*, 2nd Edition, O'Reilly Media

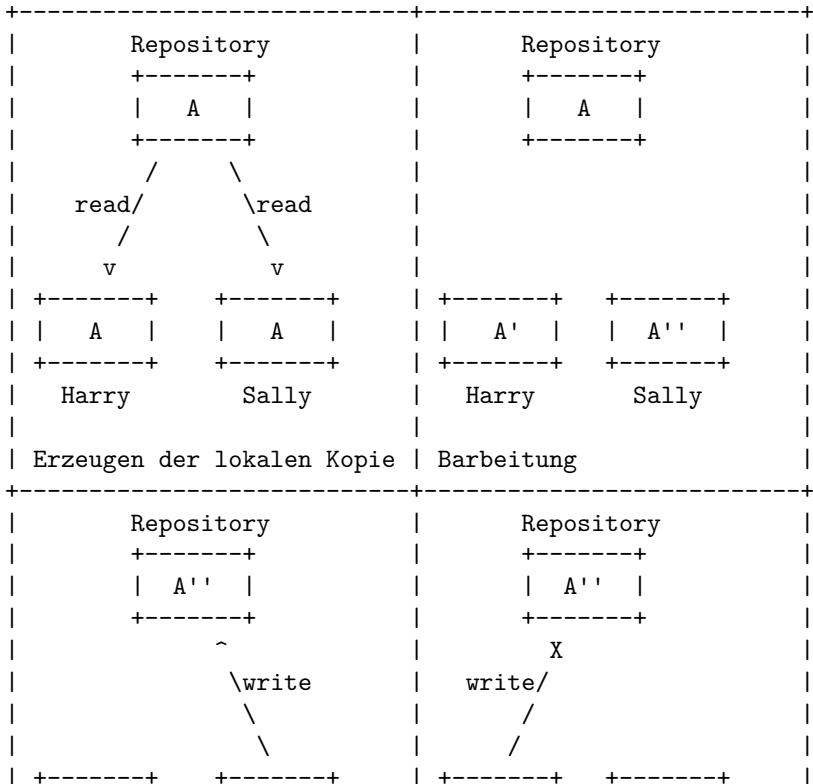


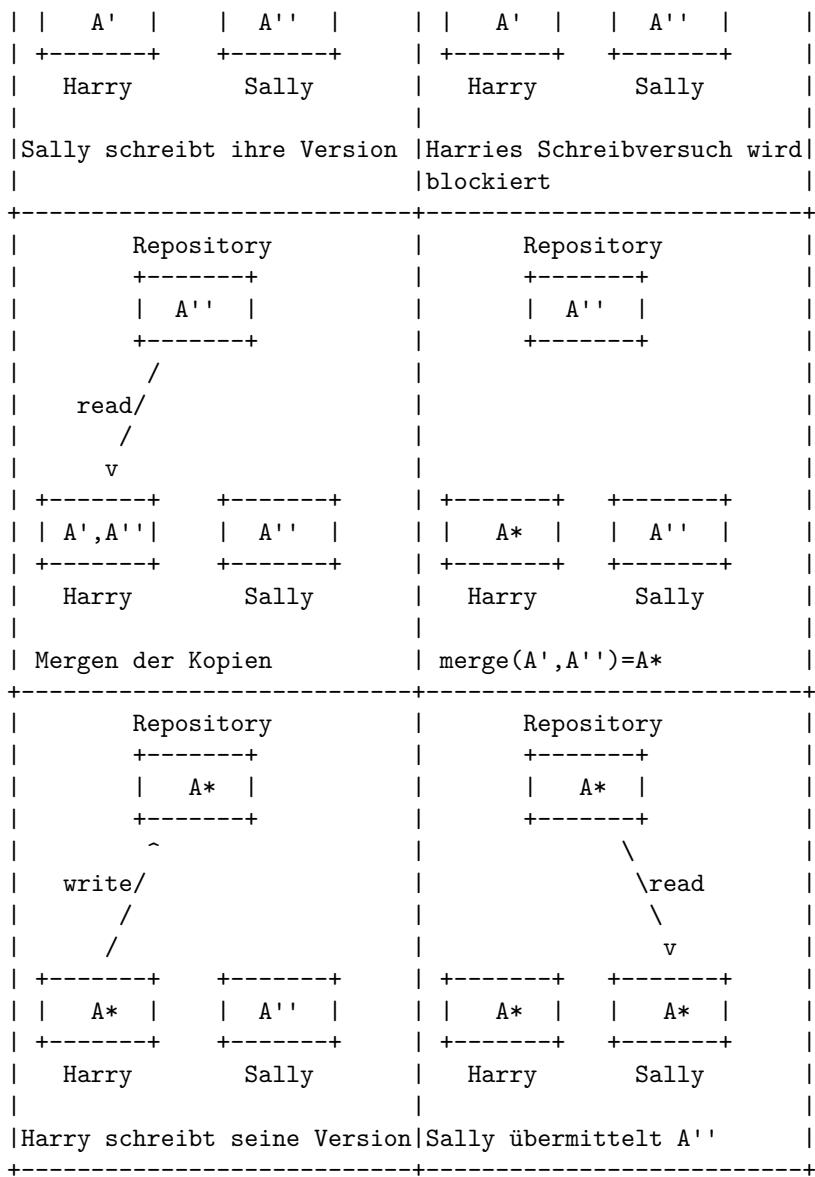
Welche Aspekte sehen Sie an dieser Lösung kritisch?

1. Administrative Probleme ... Gesperrte Dokumente werden vergessen zu entsperren.
  2. Unnötige Sequentialisierung der Arbeit ... Wenn zwei Nutzer ein Dokument an verschiedenen Stellen ändern möchten, könnten sie dies auch gleichzeitig tun.
  3. Keine Abbildung von übergreifenden Abhängigkeiten ... Zwei Nutzer arbeiten getrennt auf den Dokumenten A und B. Was passiert, wenn A von B abhängig ist? A und B passen nicht mehr zusammen. Die Nutzer müssen dieses Problem diskutieren.

Lösung II - Kollaboratives Arbeiten mit Mischen (Mergen)

Optimistische Versionsverwaltungen (*Copy Modify Merge*) versuchen die Schwächen der pessimistischen Versionsverwaltung zu beheben, in dem sie gleichzeitige Änderungen durch mehrere Benutzer an einer Datei zu lassen und anschließend diese Änderungen automatisch oder manuell zusammen führen (Merge).





Ablauf:

- Harry und Sally kopieren das Dokument A in ihre lokalen Ordner.
- Beide arbeiten unabhängig daran und erzeugen die Versionen A' und A''
- Sally schreibt als Erste das Dokument in das Repository zurück.
- Harry kann das Dokument nun nicht mehr zurückschreiben, seine Version ist veraltet
- Harry vergleicht seine lokale Version mit der aktuellen Version im Repository und mischt die Änderungen von Sally mit seinen Anpassungen
- Die neue (gemischte) Version A\* wird zurückgeschrieben.
- Sally muss eine neue Leseoperation realisieren, da Ihre lokale Version veraltet ist.

Welche Konsequenzen ergeben sich daraus?

- Unser Dokument muss überhaupt kombinierbar sein! Auf ein binäres Format ließe sich das Konzept nicht anwenden!
- Das Dokument liegt in zeitgleich in n-Versionen vor, die ggf. überlappende Änderungen umfassen.
- Das zentrale Repository kennt die Version von Harry nur indirekt. Man kann zwar indirekt aus A'' und A\* auf A' schließen, man verliert aber zum Beispiel die Information wann Harry seine Änderungen eingebaut hat.

Die Herausforderung liegt somit im Mischen von Dokumenten!

## Mischen von Dokumenten

### Schritt 1: Identifikation von Unterschieden

Zunächst einmal müssen wir feststellen an welchen Stellen es überhaupt Unterschiede gibt. Welche Differenzen sehen Sie zwischen den beiden Dokumenten:

TU

Bergakademie

Freiberg

Softwareentwicklung

## Online Course

Sommersemester 2020

1

Etiam ipsum ac felis sit amet, semper euismod sapien nisi, sed ultricies nonummy enim tempor invidunt ut luctus.

TU

Bergakademie

Freiberg

Softwareentwicklung

Sommersemester 2019

*Etiam velut in meliore loco, sed non in melius.*

Offenbar wurden sowohl Lehrzeichen, als auch neue Zeilen eingeführt. In anderen Zeilen wurden Inhalte angepasst.

Nutzen wir das Tool `diff` um diese Änderungen automatisiert festzustellen. Die Zeilen, die mit `>` beginnen, sind nur in der ersten Datei vorhanden, diejenigen, die mit `<`, markieren das Vorkommen in der zweiten Datei. Die einzelnen Blöcke werden durch sogenannte change commands („Änderungsbefehle“) getrennt, die angeben, welche Aktion (Zeilen hinzufügen – `a`, ändern – `c` oder entfernen – `d`) in welchen Zeilen ausgeführt wurde.

```
▶diff DokumentV1.md DokumentV2.md
0a1,3
>
>
>
5,7c8,9
< Online Course
< Sommersemester 2020
< Lorem ipsum dolor sit amet, CONSETETUR sadipscing elit, ...
---
> Sommersemester 2019
> Lorem ipsum dolor sit amet, consetetur sadipscing elit, ...
```

**Merke:** Sehr lange Zeilen erschweren die Suche nach wirklichen Änderungen!

Dahinter steht das *Longest Common Subsequence* Problem, dessen Umsetzung kurz dargestellt werden soll.

```
def lcs_algo(S1, S2, m, n):
    L = [[0 for x in range(n+1)] for x in range(m+1)]

    # Building the matrix in bottom-up way
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0:
                L[i][j] = 0
            elif S1[i-1] == S2[j-1]:
                L[i][j] = L[i-1][j-1] + 1
            else:
                L[i][j] = max(L[i-1][j], L[i][j-1])

    index = L[m][n]

    lcs_algo = [""] * (index+1)
    lcs_algo[index] = ""

    i = m
    j = n
    while i >= 0 and j >= 0:
        if S1[i-1] == S2[j-1]:
            lcs_algo[index-1] = S1[i-1]
            i -= 1
            j -= 1
        elif L[i-1][j] > L[i][j-1]:
            lcs_algo[index-1] = S1[i-1]
            i -= 1
        else:
            lcs_algo[index-1] = S2[j-1]
            j -= 1

    print(lcs_algo)
```

```

j = n
while i > 0 and j > 0:

    if S1[i-1] == S2[j-1]:
        lcs_algo[index-1] = S1[i-1]
        i -= 1
        j -= 1
        index -= 1

    elif L[i-1][j] > L[i][j-1]:
        i -= 1
    else:
        j -= 1

# Printing the sub sequences
print("S1 : " + S1 + "\nS2 : " + S2)
print("LCS: " + "".join(lcs_algo))

S1 = "Das ist ein Test"
S2 = "und das ist ein weiterer Test mit mehr Zeichen"
m = len(S1)
n = len(S2)
lcs_algo(S1, S2, m, n)

@Pyodide.eval

```

### Schritt 2: Mischen

In der Praxis wird zwischen zwei Szenarien unterschieden:

1. Mischen unabhängiger Dokumente (2-Wege-Mischen) - Ziel ist die Erzeugung eines neuen Dokumentes, dass die gemeinsamen Komponenten und individuelle Teilmengen vereint.
2. Mischen von Dokumenten mit gemeinsamen Ursprung (3-Wege-Mischen) - Ziel ist die Integration möglichst aller Änderungen der neuen Dokumente in eine weiterentwickelte Version des Ursprungsdocuments

Ein Paar von Änderung aus D1 bzw. D2 gegenüber einem Ausgangsdokument D0 kann unverträglich sein, wenn die Abbildung beider Änderungen in einem gemeinsamen Dokument nicht möglich ist. In diesem Fall spricht man von einem Konflikt.

Bei einem Konflikt muss eine der beiden Änderungen weggelassen werden. Die Entscheidung darüber kann anhand von zwei Vorgehensweisen realisiert werden:

1. Nicht-interaktives Mischen: Es wird zunächst ein Mischergebnis erzeugt, das beide Änderungen umfasst. Über eine entsprechende Semantik werden die notwendigerweise duplizierten Stellen hervorgehoben. Ein Vorteil dieser Vorgehensweise ist, dass ein beliebiges weitergehendes Editieren zur Konfliktauflösung möglich ist.
2. Interaktives Mischen: Ein Entwickler wird unmittelbar in den Mischprozess eingebunden und um "Schritt-für-Schritt" Entscheidungen gebeten. Denkbare Entscheidungen dabei sind:
  - Übernahme der Änderung gemäß D1 oder D2,
  - Übernahme keiner Änderung,
  - Übernahme von modifizierten Änderung

### Revisionen

Bislang haben wir lediglich einzelne Dateien betrachtet. Logischerweise muss ein übergreifender Ansatz auch Ordnerstrukturen integrieren.

Damit werden sowohl die Ordnerstruktur als auch die Dokumente als Struktur, wie auch deren Inhalte, erfasst.

Wichtig für die Nachvollziehbarkeit der Entwicklung ist somit die Kontinuität der Erfassung!

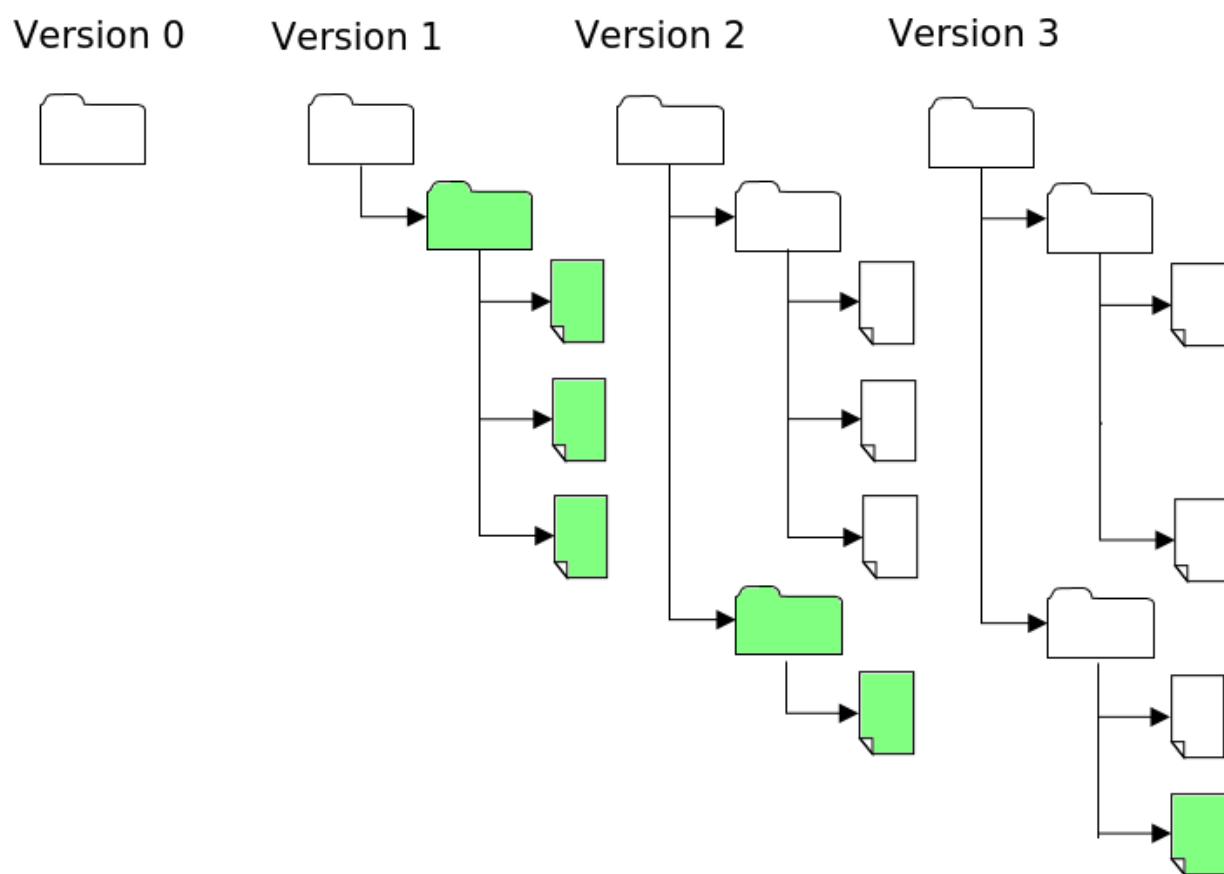


Figure 12.2: ProblemKollaborativesArbeiten

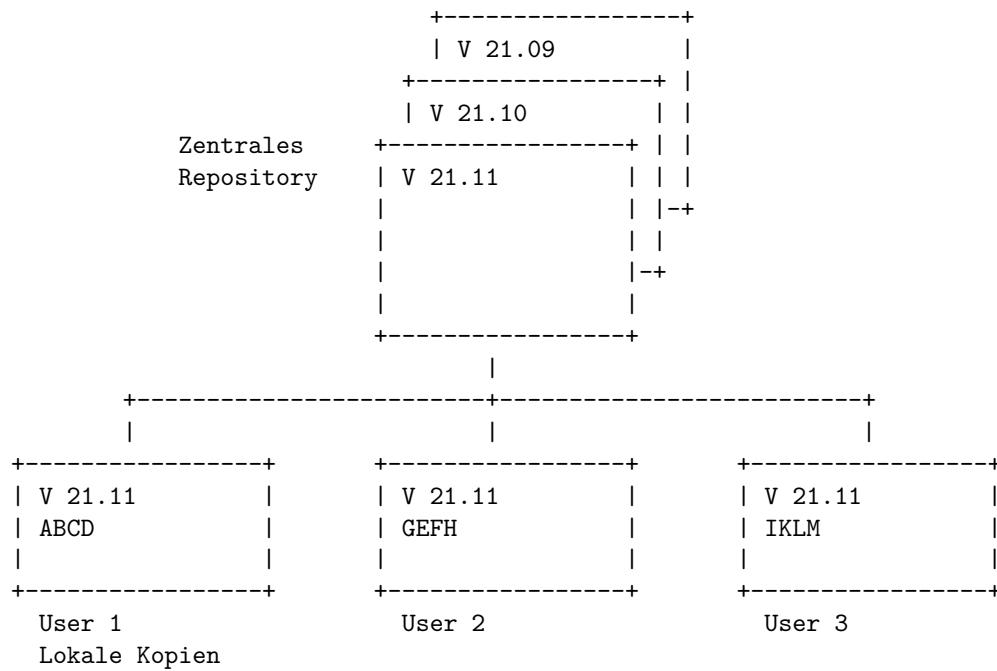
Wenn sich der Ordner- oder Dateiname ändert wollen wir trotzdem noch die gesamte History der Entwicklung innerhalb eines Dokuments kennen. Folglich muss ein Link zwischen altem und neuem Namen gesetzt werden.

## Formen der Versionsverwaltung

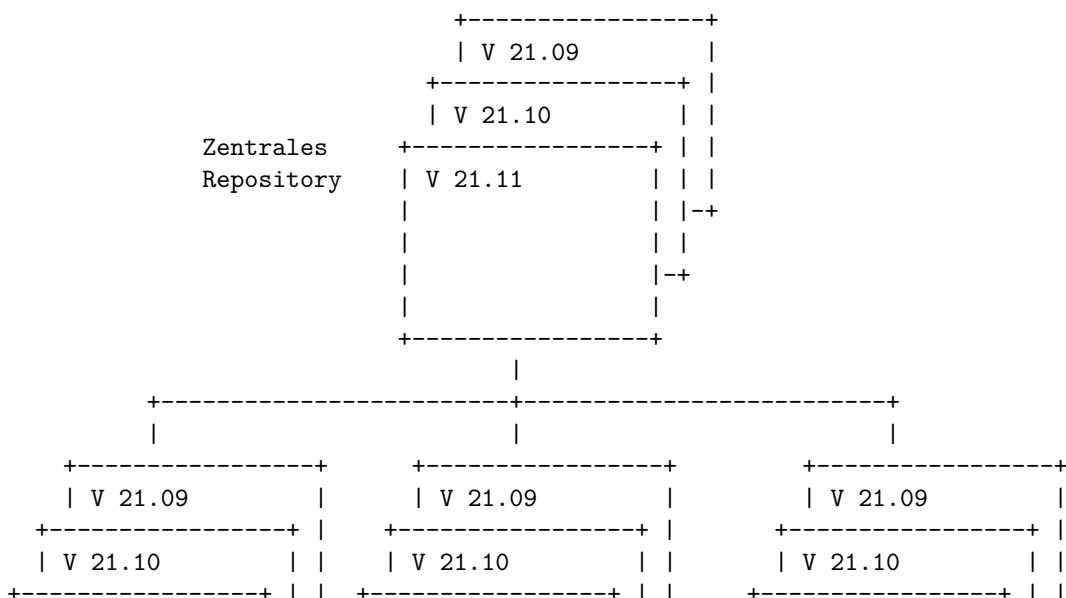
**Lokale Versionsverwaltung** Bei der lokalen Versionsverwaltung wird oft nur eine einzige Datei versioniert, diese Variante wurde mit Werkzeugen wie SCCS und RCS umgesetzt. Sie findet auch heute noch Verwendung in Büroanwendungen, die Versionen eines Dokumentes in der Datei des Dokuments selbst speichern (Word).

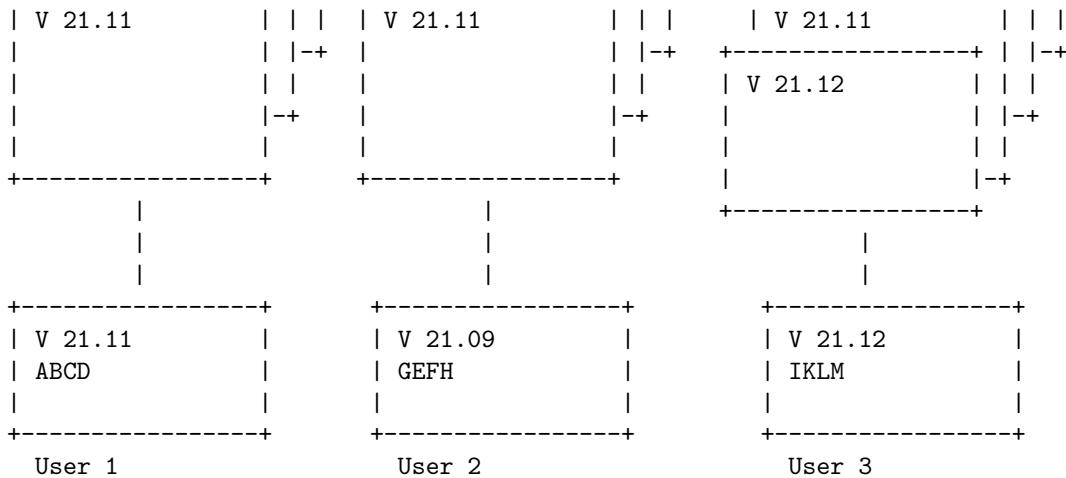
**Zentrale Versionsverwaltung** Diese Art ist als Client-Server-System aufgebaut, sodass der Zugriff auf ein Repository auch über Netzwerk erfolgen kann. Durch eine Rechteverwaltung wird dafür gesorgt, dass nur berechtigte Personen neue Versionen in das Archiv legen können. Die Versionsgeschichte ist hierbei nur im Repository vorhanden.

Dieses Konzept wurde vom Open-Source-Projekt Concurrent Versions System (CVS) populär gemacht, mit Subversion (SVN) neu implementiert und von vielen kommerziellen Anbietern verwendet.



**Verteilte Versionsverwaltung** Die verteilte Versionsverwaltung (DVCS, distributed VCS) verwendet kein zentrales Repository mehr. Jeder, der an dem verwalteten Projekt arbeitet, hat sein eigenes Repository und kann dieses mit jedem beliebigen anderen Repository abgleichen. Die Versionsgeschichte ist dadurch genauso verteilt. Änderungen können lokal verfolgt werden, ohne eine Verbindung zu einem Server aufzubauen zu müssen.





### Zentrale Versionsverwaltung

Historie liegt nur auf dem Server

Zentrales Repository als Verbindungselement  
Konflikte bei Manipulation eines Dokumentes durch  
mehrere Autoren  
Sequenz von Versionen

### Verteilte Versionsverwaltung

gesamte Historie ist den lokalen Repositories  
bekannt

n gleichberechtigte Repositories  
Existenz paralleler Versionen eines Dokumentes  
abgesichert  
gerichteter azyklischer Graph

## Git

### Geschichte und Einsatz

Die Entwicklungsgeschichte von git ist mit der des Linux Kernels verbunden:

Jahr	Methode der Versionsverwaltung
1991	Änderungen am Linux Kernel via patches und archive files
2002	Linux Kernel mit dem Tool BitKeeper verwaltet
2005	Bruch zwischen der vertreibenden Firma und der Linux Community
2021	Die aktuelle Version ist 2.31.1

2005 wurde eine Anforderungsliste für eine Neuentwicklung definiert. Dabei wurde hervorgehoben, dass sie insbesondere sehr große Projekte (Zahl der Entwickler, Features und Codezeilen, Dateien) unterstützen müssen. Daraus entstand **Git** als freie Software zur verteilten Versionsverwaltung von Dateien.

Git dominiert entweder als einzelne Installation oder aber eingebettet in verschiedene Entwicklungsplattform die Softwareentwicklung!

### Wie bekommen sie Git auf Ihren Windows-Rechner?

1. Variante 1: als Integralen Bestandteil in Ihrer Entwicklungsumgebung (Visual Studio Code) Die großen IDEs umfassen einen eigenen Git-Client, für einfache Editoren muss dieser meist nachinstalliert werden.
2. Variante 2: als unabhängige Installation Unter [Link](#) findet sich eine Schritt-für-Schritt Beschreibung für die Installation von Git unter Windows. Dabei wird sowohl ein Shell als auch eine GUI installiert.
3. Variante 3: mittels cygwin Cygwin emuliert Linuxbefehle und Tools der Shell. Neben Compilern und einer Vielzahl von Entwicklertools können auch verschiedene Versionsverwaltungen installiert werden.

Eine übergreifende Erklärung für die Installation gibt zum Beispiel dieses [Tutorial](#).

### Zustandsmodell einer Datei in Git

Dateien können unterschiedliche Zustände haben, mit denen sie in Git-Repositories markiert sind.

```
@startuml
hide empty description
```

```

[*] --> Untracked : Erzeugen einer Datei
Untracked --> Staged : Hinzufügen zum Repository
Unmodified --> Modified : Editierung der Datei
Modified --> Staged : Markiert als neue Version
Staged --> Unmodified : Bestätigt als neue Version
Unmodified --> Untracked : Löschen aus dem Repository
@enduml

@startuml
hide empty description
[*] --> Untracked : Erzeugen einer Datei
Untracked --> Staged : Hinzufügen zum Repository\n <color:Red> ""git add"""
Unmodified --> Modified : Editierung der Datei
Modified --> Staged : Markiert als neue Version\n <color:Red> ""git add"""
Staged --> Unmodified : Bestätigt als neue Version\n <color:Red> ""git commit"""
Unmodified --> Untracked : Löschen aus dem Repository\n <color:Red> ""git remove"""
@enduml

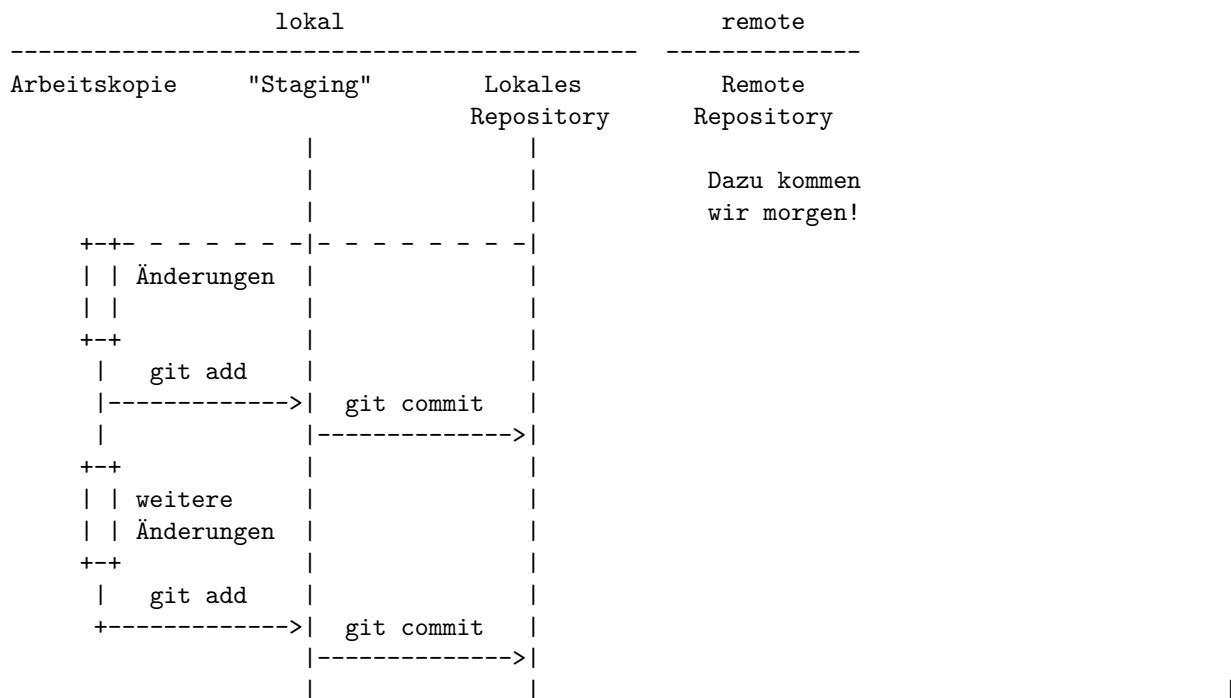
```

## Grundlegende Anwendung (lokal!)

**Merke:** Anders als bei svn können Sie mit git eine völlig autonome Versionierung auf Ihrem Rechner realisieren. Ein Server ist dazu *zunächst* nicht nötig.

Aus dem Zustandmodell einer Datei ergeben sich drei Ebenen auf der wir eine Datei in Git Perspektivisch einordnen können - Arbeitsverzeichnis, Stage und Repository.

**Achtung:** Die folgende Darstellung dient hauptsächlich der didaktischen Hinführung zum Thema!



Phase 1: Anlegen des Projektes und initialer Commit

```

mkdir GitBasicExample
cd GitBasicExample
git init
touch README.md
// Hier kommen jetzt einige Anpassungen in README.md dazu
git add README.md
git commit -m "Add first commit!"
git status

```

Phase 2: Generieren unseres DotNet Projektes

```
cd GitBasicExample
```

```
dotnet new console -o MyApp
cd MyApp
dotnet run    // Nur zur Testzwecken
git status
git add MyApp.csproj
git add Program.cs
git commit -m "Add initial dotnet project!"
git log --oneline
```

**Merke:** Beschränken Sie die versionierten Dateien auf ein Minimum! Der gesamte Umfang des dotnet-Projektes gehört nur in seltenen Fällen ins Repository!

Phase 3: Arbeit im Projekt

```
// Veränderungen im Programmcode
git add Program.cs
git commit -m "Change output of project!"
git log --oneline
```

Bis hierher haben wir lediglich eine Erfassung unserer Aktivitäten umgesetzt. Wir können anhand der Log-Files einen Überblick darüber behalten, wann welche Änderungen in welcher Datei realisiert wurden.

## “Kommando zurück”

Nun wird es aber interessanter! Lassen Sie uns jetzt aber zwischen den Varianten navigieren.

### Variante 1 - Reset

`git reset` löscht die Historie bis zu einem Commit. Adressieren können wir die Commits relativ zum HEAD `git reset HEAD~1` oder mit der jeweiligen ID `git reset <commit-id>`.

Wichtig sind dabei die Parameter des Aufrufes:

Attribut	Bedeutung
<code>git reset --soft</code>	uncommit changes, changes are left staged (index).
<code>git reset --mixed</code>	(default): uncommit + unstaged changes, changes are left in working tree.
<code>git reset --hard</code>	uncommit + unstaged + delete changes, nothing left.

```
text @ExplainGit.eval git commit -m V1 git commit -m V2 git commit -m V3
```

### Variante 2 - Revert

Häufig möchte man nicht die gesamte Historie zurückgehen und alle Änderungen verwerfen, sondern vielmehr eine Anpassung zurücknehmen und deren Auswirkung korrigieren. Eine Anpassung würde aber bedeuten, dass alle nachgeordneten Commits angepasst werden müssten.

```
text @ExplainGit.eval git commit -m V1 git commit -m V2 git commit -m V3
```

### Variante 3 - Rebase

Ein sehr mächtiges Werkzeug ist der interaktive Modus von `git rebase`. Damit kann die Geschichte neugeschrieben werden, wie es die git Dokumentation beschreibt. Im Grund können Sie damit Ihre Versionsgeschichte „aufräumen“. Einzelne Commits umbenennen, löschen oder fusionieren. Dafür besteht ein eigenes Interface, dass Sie mit dem folgenden Befehl aufrufen können:

► `git rebase -i HEAD~5`

```
pick d2a06e4 Update main.yml
pick 78839b0 Reconfigures checkout
pick f70cfcc7 Replaces wildcard by specific filename
pick 05b76f3 New pandoc command line
pick c56a779 Corrects md filename

# Rebase a3b07d4..c56a779 onto a3b07d4 (5 commands)
#
# Commands:
```

```

# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out

```

Als Anwendungsfall habe ich mir meine Aktivitäten im Kontext einiger Experimente mit den GitHub Actions, die im nächsten Abschnitt kurz eingeführt werden, ausgesucht. Schauen wir zunächst auf den ursprünglichen Stand. Alle Experimente drehten sich darum, eine Datei anzupassen und dann auf dem Server die Korrektheit zu testen.



```

c56a779 - Sebastian Zug, 7 hours ago : Corrects md filename
05b76f3 - Sebastian Zug, 7 hours ago : New pandoc command line
f70cfcc7 - Sebastian Zug, 8 hours ago : Replaces wildcard by specific filename
78839b0 - Sebastian Zug, 21 hours ago : Reconfigures checkout
d2a06e4 - Sebastian Zug, 22 hours ago : Update main.yml
...
aa04051 - Sebastian Zug, 23 hours ago : Restart action activities
4b22d12 - Sebastian Zug, 23 hours ago : Deleting old state
64075cc - Sebastian Zug, 24 hours ago : Update main.yml
01f341b - Sebastian Zug, 24 hours ago : Missing links added
...
29c8e68 - Sebastian Zug, 11 days ago : Update README.md

```

Unser lokaler Branch liegt nach dem Löschen aber um einiges hinter dem auf GitHub entsprechend müssen wir mit `git push --force` das Überschreiben erzwingen.

## Was kann schief gehen?

### 1. Ups, die Datei sollte im Commit nicht dabei sein (Unstage)

```

... ein neues Feature wird in sourceA.cs implementiert
... ein Bug in Codedatei sourceB.cs korrigiert
... wir wollen schnell sein und fügen alle Änderungen als staged ein
git add *
... Aber halt! Die beiden Dinge gehören doch nicht zusammen!
git reset

```

```
text @ExplainGit.eval git commit -m V1 git commit -m V2 git commit -m V3
```

### 2. Ups, eine Datei in der Version vergessen! (Unvollständiger Commit)

```

... eine neue Codedatei source.cs anlegen
... zugehörige Anpassungen in der README.md erklären
git add README.md
git commit -m "Hinzugefügt neues Feature"
... Leider wurde die zugehörige Code Datei vergessen!
git add source.cs
git commit --amend --no-edit
... Hinzufügen der Datei ohne die Log Nachricht anzupassen

```

## Ich sehe was, was Du nicht siehst ...

Häufigbettet ein Projekt Dateien ein, die Git nicht automatisch hinzufügen oder schon gar nicht als „nicht versioniert“ anzeigen soll. Beispiele dafür sind automatisch generierte Dateien, wie Log-Dateien oder die Binaries, die von Ihrem Build-System erzeugt wurden. In solchen Fällen können Sie die Datei `.gitignore` erstellen, die eine Liste mit Vergleichsmustern enthält. Hier ist eine `.gitignore` Beispieldatei:

```
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in any directory named build
build/

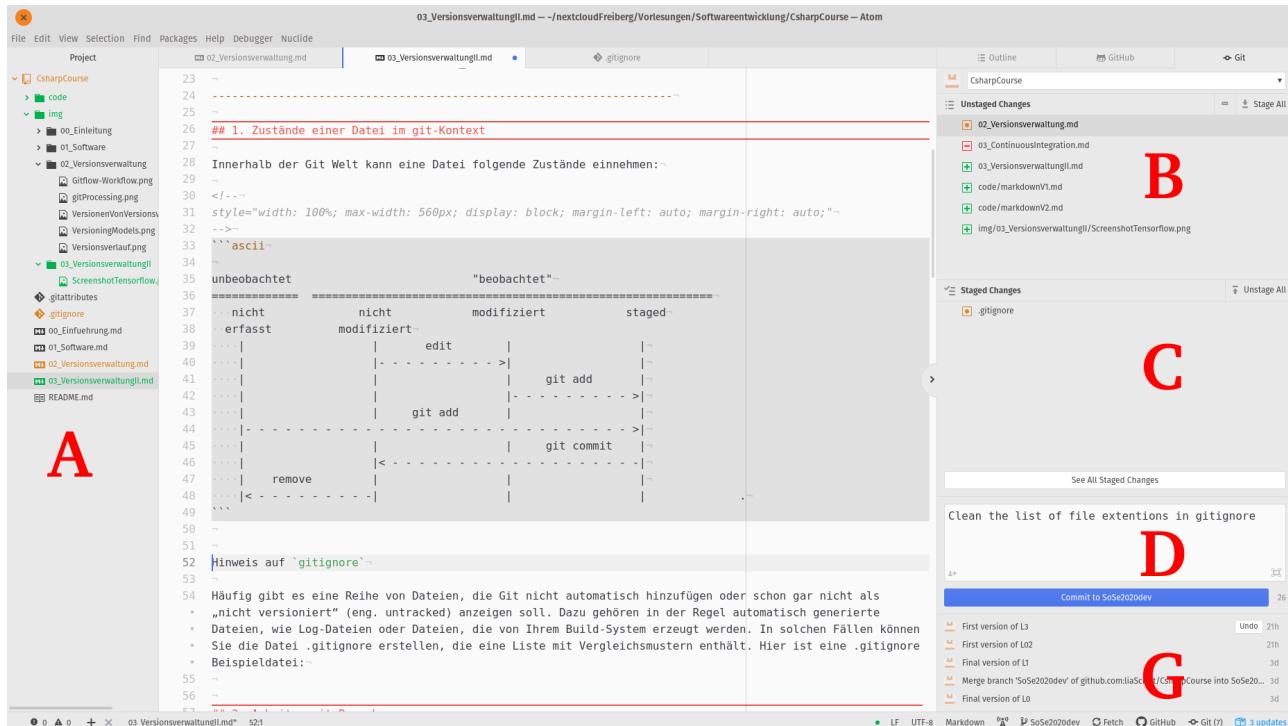
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/**.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf
```

Unter [gitIgnoreBeispiele](#) gibt es eine ganze Sammlung von Konfigurationen für bestimmte Projekttypen.

## Kommandozeile oder keine Kommandozeile

Die Editoren unterstützen den Nutzer bei der Arbeit, in dem Sie die eigentlichen Commandozeilentools rund um die Arbeitsfläche anordnen.



### BereichBedeutung

- A Darstellung der Dateien im Ordner, wobei der Typ über die führenden Symbole und der Zustand über die Farbgebung hervorgehoben wird (grün untracked Files, orange getrackte Dateien mit Änderungen).

---

BereicBedeutung

- B Hier erfolgt die Darstellung der **Staged Changes** also der registrierten Änderungen. Offenbar ist die Datei `02_Versionsverwaltung.md` verändert worden, ohne dass ein entsprechendes `git add 02_Versionsverwaltung.md` ausgeführt wurde.
- C Die Übersicht der erfassten Änderungen mit einem **stage** Status übernimmt die aus der darüber geführten Liste, wenn der Befehl `git add` ausgeführt wurde. Offenbar ist dies fr. `.gitignore` der Fall.
- D Die Übertragung ins lokale Repository wird gerade vorbereitet. Die Commit-Nachricht ist bereits eingegeben. Der Button zeigt den zugehörigen Branch.
- G An dieser Stelle wird ein kurzes Log der letzten Commits ausgegeben. Dies ermöglicht das effiziente Durchsuchen der nach bestimmten Veränderungen.
- 

Die identischen Informationen lassen sich auch auf der Kommandozeile einsehen:

```
►git status
On branch SoSe2020dev
Your branch is up to date with 'origin/SoSe2020dev'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   .gitignore

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   02_Versionsverwaltung.md
    deleted:   03_ContinuousIntegration.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    03_VersionsverwaltungII.md
    code/
    img/03_VersionsverwaltungII/

►git log
commit d7603554c958c478f1ec600bd3cce437d91ae9a (HEAD -> SoSe2020dev, origin/SoSe2020dev)
Author: Sebastian Zug <Sebastian.Zug@informatik.tu-freiberg.de>
Date:   Thu Apr 9 13:16:38 2020 +0200

  First version of L3

commit 39fc168222f4c7a7d062adcaccff17fe34bccbe3
Author: Sebastian Zug <Sebastian.Zug@informatik.tu-freiberg.de>
Date:   Thu Apr 9 13:16:12 2020 +0200

  First version of L02

commit 350c127c7dfbc61a81edc8bd148f605ee681a07a
Author: Sebastian Zug <Sebastian.Zug@informatik.tu-freiberg.de>
Date:   Tue Apr 7 07:01:02 2020 +0200

  Final version of L1
....
```

## Aufgaben

- [ ] Legen Sie sich einen GitHub Account an.
- [ ] Experimentieren Sie mit lokalen Repositories!

# Chapter 13

## Versionsverwaltung II

---

Parameter Kursinformationen

---

**Veranstaltung:** Vorlesung Softwareentwicklung

Semester Sommersemester 2021

Hochschule: Technische Universität Freiberg

Inhalte: Versionsverwaltung mit Git und GitHub

Link auf [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/12\\_VersionsverwaltungII.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/12_VersionsverwaltungII.md)

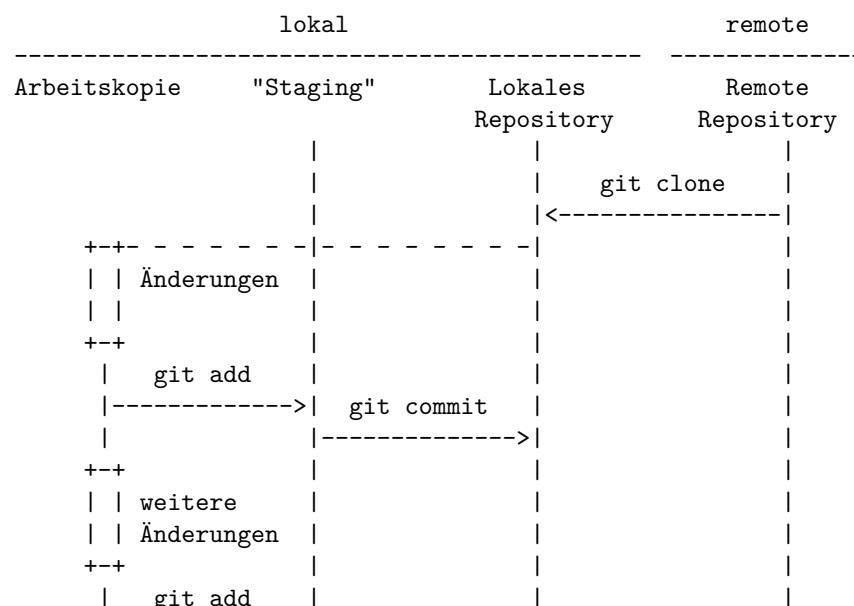
GitHub:

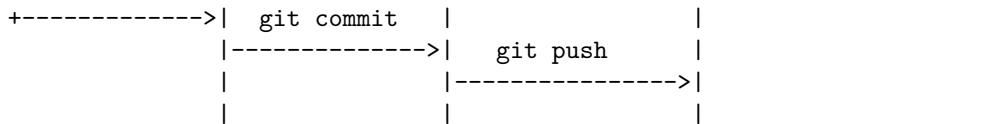
Autoren @author

---

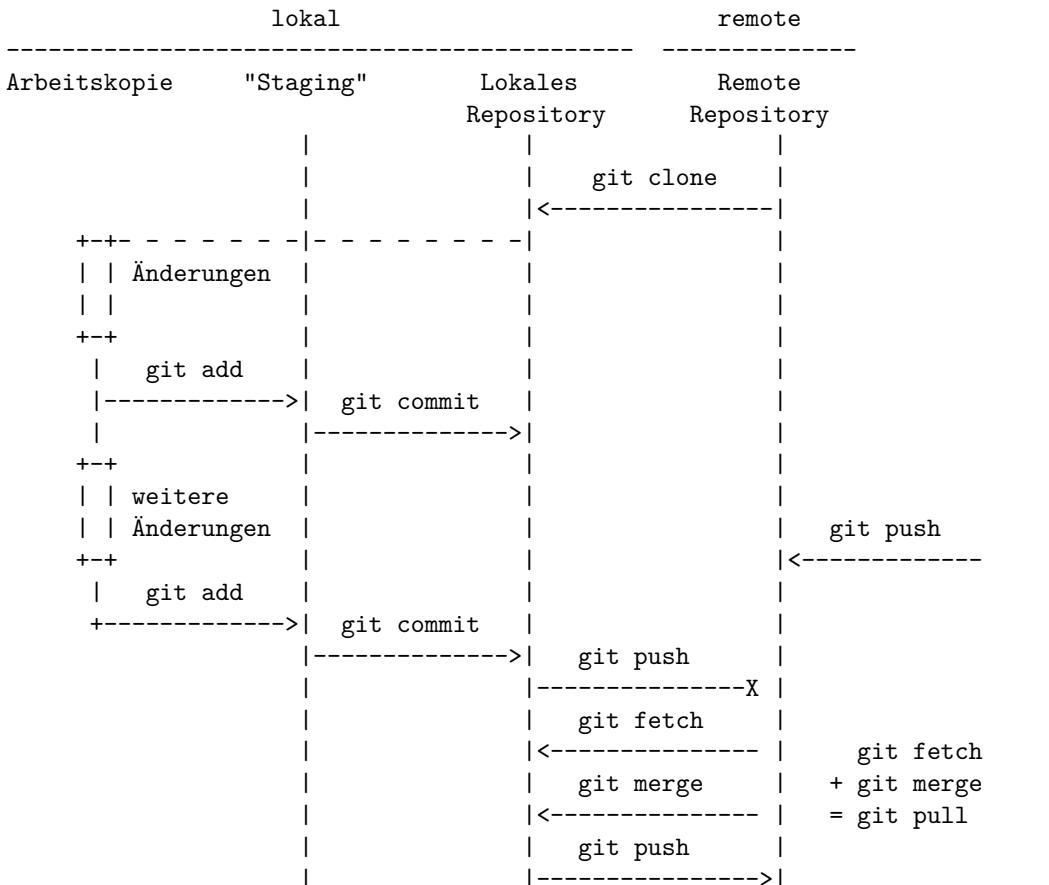
### Verteiltes Versionsmanagement

*Einfaches Editieren:* Sie klonen das gesamte Repository, dass sich auf dem “Server-Rechner” befindet. Damit haben Sie eine vollständige Kopie aller Versionen in Ihrem Working Directory. Wenn wir annehmen, dass keine branches im Repository bestehen, dann können Sie direkt auf der Ihrer Arbeitskopie arbeiten und diese verändern. Danach generieren Sie einen Snappshot des Arbeitsstandes *Staging*. Ihre Version ist als relevant markiert und kann im lokalen Repository als neuer Eintrag abgelegt werden. Vielleicht wollen sie Ihren Algorithmus noch weiterentwickeln und speichern zu einem späteren Zeitpunkt eine weitere Version. All diese Vorgänge betreffen aber zunächst nur Ihre Kopie, ein anderer Mitstreiter in diesem Repository kann darauf erst zurückgreifen, wenn Sie das Ganze an den Server übermittelt haben.





*Kooperatives Arbeiten:* Nehmen wir nun an, dass Ihr Kollege in dieser Zeit selbst das Remote Repository fortgeschrieben hat. In diesem Fall bekommen Sie bei Ihrem push eine Fehlermeldung, die sie auf die neuere Version hinweist. Nun “ziehen” Sie sich den aktuellen Stand in Ihr Repository und kombinieren die Änderungen. Sofern keine Konflikte entstehen, wird daraus ein neuer Commit generiert, den Sie dann mit Ihren Anpassungen an das Remote-Repository senden.



Versuchen wir das ganze noch mal etwas plastischer nachzuvollziehen. Rechts oben sehen Sie unser Remote-Repository auf dem Server. Im mittleren Bereich den Status unseres lokalen Repositories.

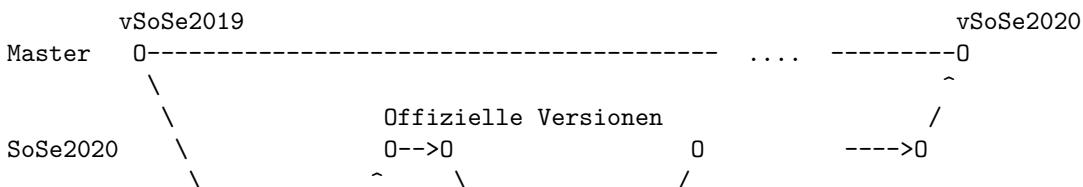
text @ExplainGit.eval create origin

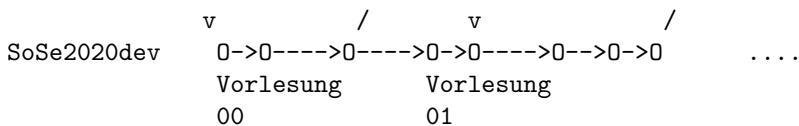
## Arbeiten mit Branches

Die Organisation von Versionen in unterschiedlichen Branches ist ein zentrales Element der Arbeit mit git. Branches sind Verzweigungen des Codes, die bestimmte Entwicklungsziele kapseln.

Der größte Nachteil bei der Arbeit mit nur einem Branch liegt darin, dass bei einem defekten Master(-Branch) die Arbeit sämtlicher Beteiligter unterbrochen wird. Branches schaffen einen eignen (temporären) Raum für die Entwicklung neuer Features, ohne die Stabilität des Gesamtsystems zu gefährden. Gleichzeitig haben die Entwickler den gesamten Verlauf eines Projekts in strukturierter Art zur Hand.

Wie sieht das zum Beispiel für unsere Kursmaterialien aus?





Ein Branch in Git ist einfach ein Zeiger auf einen Commit zeigt. Der zentrale Branch wird zumeist als `master` bezeichnet.

## Generieren und Navigation über Branches

Wie navigieren wir nun konkret über den verschiedenen Entwicklungszweigen.

```
text @ExplainGit.eval git commit -m V1 git commit -m V2 git commit -m V3
```

## Mergoperationen über Branches

Nehmen wir folgendes Scenario an. Sie arbeiten an einem Issue, dafür haben Sie einen separaten Branch (`newFeature`) erzeugt und haben bereits einige Commits realisiert. Beim Kaffeetrinken denken Sie über den Code von letzter Woche nach und Ihnen fällt ein Bug ein, den Sie noch nicht behoben haben. Jetzt aber schnell!

Legen Sie dafür einen neuen Branch an, committen Sie eine Version und mergen Sie diese mit dem Master. Kehren Sie dann in den Feature-Branch zurück und beenden Sie die Arbeit. Mergen Sie auch diesen Branch mit dem Master. Worin unterscheiden sich beide Vorgänge?

```
text @ExplainGit.eval git branch newFeature git checkout newFeature git commit -m FeatureV1
git commit -m FeatureV2
```

Mergen ist eine nicht-destruktive Operation. Die bestehenden Branches werden auf keine Weise verändert. Das Ganze „bläht“ aber den Entwicklungsbau auf.

## Rebase mit einem Branch

Zum `merge` existiert auch noch eine alternative Operation. Mit `rebase` werden die Änderungen eines branches in einem Patch zusammengefasst. Dieser wird dann auf `head` angewandt.

```
text @ExplainGit.eval git branch newFeature git checkout newFeature git commit -m FeatureV1
git commit -m FeatureV2 git checkout master git commit -m V1
```

## Arbeit mit GitHub

### Issues

Issues dienen dem Sammeln von Benutzer-Feedback, dem Melden von Software-Bugs und der Strukturierung von Aufgaben, die Sie erledigen möchten. Dabei sind Issues unmittelbar mit Versionen verknüpft, diese können dem Issue zugeordnet werden.

Sie können eine Pull-Anfrage mit einer Ausgabe verknüpfen, um zu zeigen, dass eine Korrektur in Arbeit ist und um die Ausgabe automatisch zu schließen, wenn jemand die Pull-Anfrage zusammenführt.

Um über die neuesten Kommentare in einer Ausgabe auf dem Laufenden zu bleiben, können Sie eine Ausgabe beobachten, um Benachrichtigungen über die neuesten Kommentare zu erhalten.

<https://guides.github.com/features/issues/>

### Pull requests und Reviews

Natürlich wollen wir nicht, dass „jeder“ Änderungen ungesehen in unseren Code einspeist. Entsprechend kapseln wir ja auch unseren Master-Branch. Den Zugriff regeln sowohl die Rechte der einzelnen Mitstreiter als auch die Pull-Request und Review Konfiguration.

---

Status  
des  
Beitragenden  
Einreichen des Codes

---

Collaborator  Auschecken des aktuellen Repositories und Arbeit. Wenn die Arbeit in einem eigenen Branch erfolgt, wird diese mit einem Pull-Request angezeigt und gemerged.

---

Status des Beitragenden	Einreichen des Codes
non Collaborator	Keine Möglichkeit seine Änderungen unmittelbar ins Repository zu schreiben. Der Entwickler erzeugt eine eigene Remote Kopie ( <i>Fork</i> ) des Repositories und dortige Realisierung der Änderungen. Danach werden diese als Pull-Request eingereicht.

---

Wird ein Pull Request akzeptiert, so spricht man von einem *Merge*, wird er geschlossen, so spricht man von einem *Close*. Vor dem Merge sollte eine gründliche Prüfung sichergestellt werden, diese kann in Teilen automatisch erfolgen oder durch Reviews [Doku](#)

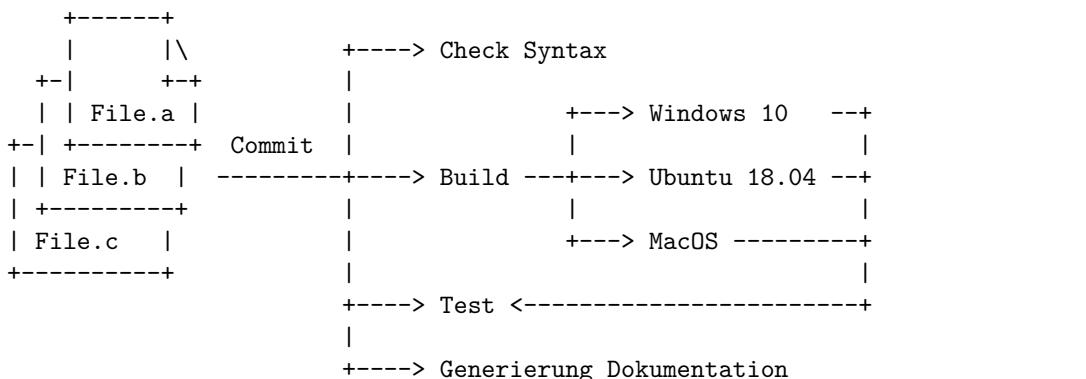
## Automatisierung der Arbeit

Hervorragend! Wir sind nun in der Lage die Entwicklung unseres Codes zu “verwalten”. Allerdings sagt noch niemand, dass ein eingereichter Code auch lauffähig ist. Wie können wir aber möglichst schnell realisieren, dass etwas schief geht? Es wäre wünschenswert, dass wir unmittelbar mit den Aktivitäten unserer Entwickler entsprechende Tests durchführen und zum Beispiel deren Commits zurückweisen.

An dieser Stelle wollen wir zunächst die Möglichkeiten des *Continuous Integration* aufzeigen, die differenzierte Diskussion einer Folge von Build und Test-Schritten folgt später. Wir werden diese Möglichkeit im Rahmen der Übungsaufgaben nutzen, um Ihre Lösungen zu testen.

GitHub stellt dafür die sogenannten *Actions* zur Verfügung. Dies sind Verarbeitungsketten, die auf verschiedensten Architekturen, Betriebssystemen, Konfigurationen usw. laufen können. Damit haben wir die Möglichkeit einen Quellcode für verschiedene Plattformen zu bauen und zu testen oder eine Dokumentation zu erstellen.

Ein *Workflow* wird durch vordefinierten *Trigger* ausgelöst. Dies können das Anlegen einer Datei, ein Commit oder ein Pull Request sein. Danach wird das System konfiguriert und die Folge der Verarbeitungsschritte gestartet.



GitHub gliedert diese Punkte in *Workflows* und *Jobs* in einer hierarchischen Struktur, die über `yml` Files beschrieben werden. Eine kurze Einführung zur Syntax findet sich unter [Wikipedia](#).

```

name: Hello World
on: [push]

jobs:
  build-and-run:
    name: Print Hello World
    runs-on: ubuntu-latest
    steps:
      - name: Checkout files (master branch)
        uses: actions/checkout@v2
      - name: Show all files
        run: pwd && whoami && ls -all

```

Spannend wird die Sache nun dadurch, dass es eine breite Community rund um die Actions gibt. Diese stellen häufig benötigte *Steps* bereits zur Verfügung, fertige Tools für das Bauen und Testen von .NET Code.

Die Dokumentation zu den GitHub-Actions findet sich unter <https://github.com/features/actions>. Ein umfangreicheres Beispiel finden Sie in unserem Projektordner im aktuellen Branch **SoSe2020**. Hier werden alle LiaScript-Dateien in ein pdf umgewandelt.

**Merke** Workflow files müssen unter `.github\workflows\*.yml` abgelegt werden.

## Ein Wort zur Zusammenarbeit

Bitte haben Sie immer den spezifischen Kontext Ihres Projektes vor Augen. Üblicherweise legt man am Anfang, bei einem "kleinen Hack" keinen Wert auf formelle Abläufe und Strukturen. Diese sind aber in großen Projekten unablässig.

Ein neues Feature wird in einem Issue beschrieben, in einem eigenen Branch implementiert, mit Commits beschrieben, auf den master branch abgebildet und das Issue mit Referenz auf den commit geschlossen.

Entsprechend ist die Dokumentation in Form der Issues und Commit-Messages der zentrale Schlüssel für die Interaktion im Softwareentwicklerteam. Entsprechend hoch ist Ihre Bedeutung anzusetzen.

## Commit Messages

Stöbern Sie dafür mal durch anderen Projekte (zum Beispiel [GitHub Tensorflow](#)) und informieren Sie sich über deren Policies.

Author	Message	Time
tensorflower-gardener	Fix input size used for batch normalization.	1 hour ago
.github/ISSUE_TEMPLATE	Fix GitHub issue templates.	last month
tensorflow	Fix input size used for batch normalization.	1 hour ago
third_party	Internal change	3 hours ago
tools	Merge pull request #25673 from Ryan-Qiyu-Jiang:env_capture_script_mor...	10 months ago
.bazelrc	Lower case c++1z config	16 days ago
.bazelversion	Upgrading bazel version to 2.0.0	2 months ago
.gitignore	Ignore CoreML BUILD files which are generated by the configure script	7 days ago
.pylintrc	Add soft-link to pylintrc to project root	11 months ago

Kurze (72 Zeichen oder weniger) Zusammenfassung

Ausführlicherer erklärender Text. Umfassen Sie ihn auf 72 Zeichen. Die Leerzeile Zeile, die die Zusammenfassung vom Textkörper trennt, ist entscheidend (es sei denn, Sie lassen den den Textkörper ganz weg).

Schreiben Sie Ihre Commit-Nachricht im Imperativ: "Fix bug" und nicht "Fixed Fehler" oder "Behebt Fehler". Diese Konvention stimmt mit den Commit-Nachrichten überein die von Befehlen wie "git merge" und "git revert" erzeugt werden.

Weitere Absätze kommen nach Leerzeilen.

- Aufzählungspunkte sind für eine Liste von Anpassungen in Ordnung.
- ... und noch einer

Folgende Regeln sollte man für die Beschreibung eines Commits berücksichtigen:

- Trennen Sie den Betreff durch eine Leerzeile vom folgenden Text
- Beschränkt Sie sich bei der Betreffzeile auf maximal 50 Zeichen
- Beginnen Sie die Betreffzeile mit einem Großbuchstaben
- Schreiben Sie die Betreffzeile im Imperativ
- Brechen Sie den Text der Message 72 Zeichen um

**Merke:** Beschreiben Sie in der Commit-Nachricht das was und warum, aber nicht das wie.

Das *Semantic Versioning* geht einen Schritt weiter und gibt den Commit-Messages eine feste Satzstruktur, vgl. [entwickler.de](http://entwickler.de)

## Generelles Vorgehen

Lassen Sie uns einen Blick auf das Aufgabenblatt der kommenden Woche werfen. Ihre Aufgabe besteht darin, in einem zweier Team verschiedene Rollen einzunehmen.

```
@startuml
actor Maintainer
actor Developer
== Vorbereitung ==
Maintainer --> Maintainer: Einfügen der Rolleninformation\n und des Fragebogenschlüssels\n in [[https://
Developer --> Developer: Einfügen der Rolleninformation\n und des Fragebogenschlüssels\n in [[https://g
== Projekt Initialisierung ==
Maintainer --> Maintainer: Konfiguration [[https://docs.github.com/en/organizations/organizing-members-
Maintainer --> Maintainer: Anlegen [[https://guides.github.com/features/issues/{Mastering Issues} Issue
Maintainer --> Developer: Zuweisung Issue
== Implementierung ==
activate Maintainer
Maintainer --> Maintainer: Monitoring Projekt
Developer --> Maintainer: //Issue in progress//
activate Developer
Developer --> Developer: Anlegen eines Branches
Developer -> Developer: Implementieren 1
note right
    * Anlegen neue Datei
    * Kopieren der txt Inhalte
    * Formatieren als md
    * Einbau einiger Typos, die
        der Maintainer finden soll
end note
Developer --> Developer: Commiten
Developer -> Developer: Implementieren 2
note right
    * Ergänzen eines Hello-World Codebeispiels
end note
Developer -> Developer: Commiten
== Review ==
Developer --> Developer: Starte [[https://docs.github.com/en/github/collaborating-with-issues-and-pu
Developer --> Maintainer : Anforderung [[https://docs.github.com/en/github/collaborating-with-issues-an
Maintainer --> Maintainer: Code Review / Kommentare
Maintainer --> Developer : Anforderung Nachbesserungen
note left
Bitte um zusätzliche
    * Bildunterschrift
    * Rechtschreibkorrektur
end note
Developer -> Developer: Implementieren 3
note right
    * Korrektur der Fehler
    * Einfügen einer Beschreibung der Grafik
end note
Developer --> Maintainer : Bitte um erneutes Rereview
```

```
Maintainer --> Developer : Review abgeschlossen
Maintainer --> Maintainer: Abschluss Pull request
deactivate Developer
== Deploy ==
Maintainer --> Maintainer: Abschluss des Pullrequests
Maintainer --> Maintainer: Generierung Release
deactivate Maintainer
@enduml
```

**Wichtig:** Schließen Sie Ihre Arbeit mit einem Release ab! Damit ist für uns erkennbar, dass Sie die Aufgabe erfolgreich umgesetzt haben.

## Aufgaben

!?alt-text

- [ ] Recherchieren Sie die Methode des “Myers-diff-Algorithmus” <https://blog.jcoglan.com/2017/02/12/the-myers-diff-algorithm-part-1/>
- [ ] Legen Sie sich ein lokales Repository mit git an und experimentieren Sie damit.
- [ ] Richten Sie sich für Ihren GitHub-Account einen SSH basierten Zugriff ein (erspart einem das fortwährende Eingeben eines Passwortes).
- [ ] Sie erhalten im Laufe der Woche Ihre erste Einladung für einen GitHub Classroom. Ausgehend davon werden Sie aufgefordert sich in Zweierteams zu organisieren und werden dann gemeinsam erste Gehversuche unter git zu unternehmen.



# Chapter 14

## Modellierung von Software

---

Parameter Kursinformationen

---

**Veranstaltung:** Vorlesung Softwareentwicklung

Semester Sommersemester 2021

**Hochschule:** Technische Universität Freiberg

**Inhalte:** Motivation der Modellierung von Software

**Link auf den GitHub:** [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)

**Autoren** @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);
```

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

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.*<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)

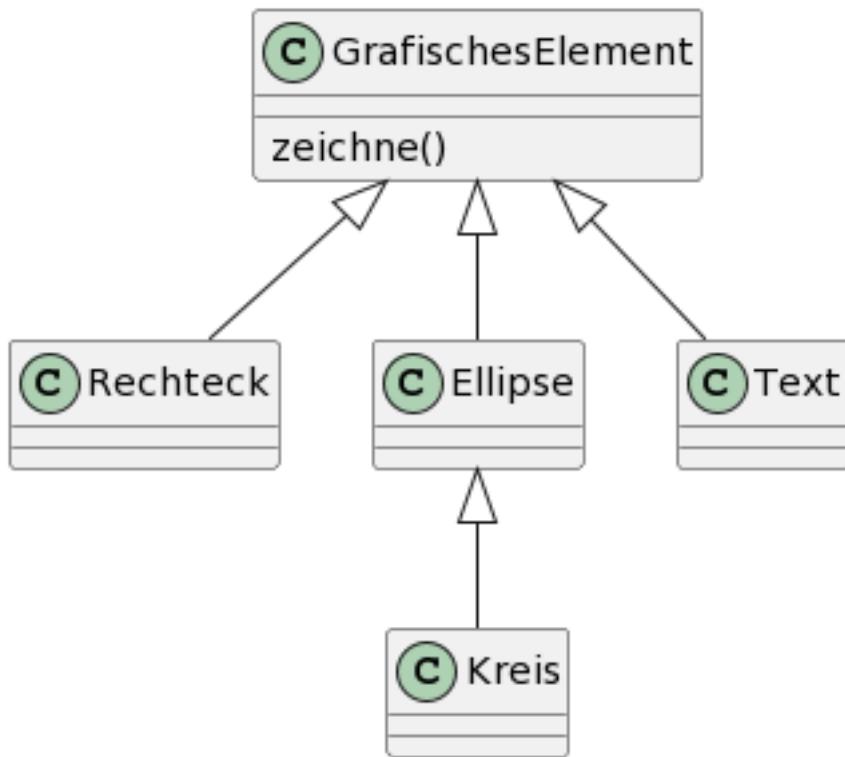
Das Liskovsche Substitutionsprinzip (LSP) oder Ersetzbarkeitsprinzip besagt, dass ein Programm, das Objekte einer Basisklasse T verwendet, auch mit Objekten der davon abgeleiteten Klasse S korrekt funktionieren muss, ohne dabei das Programm zu verändern:

*"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."*

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

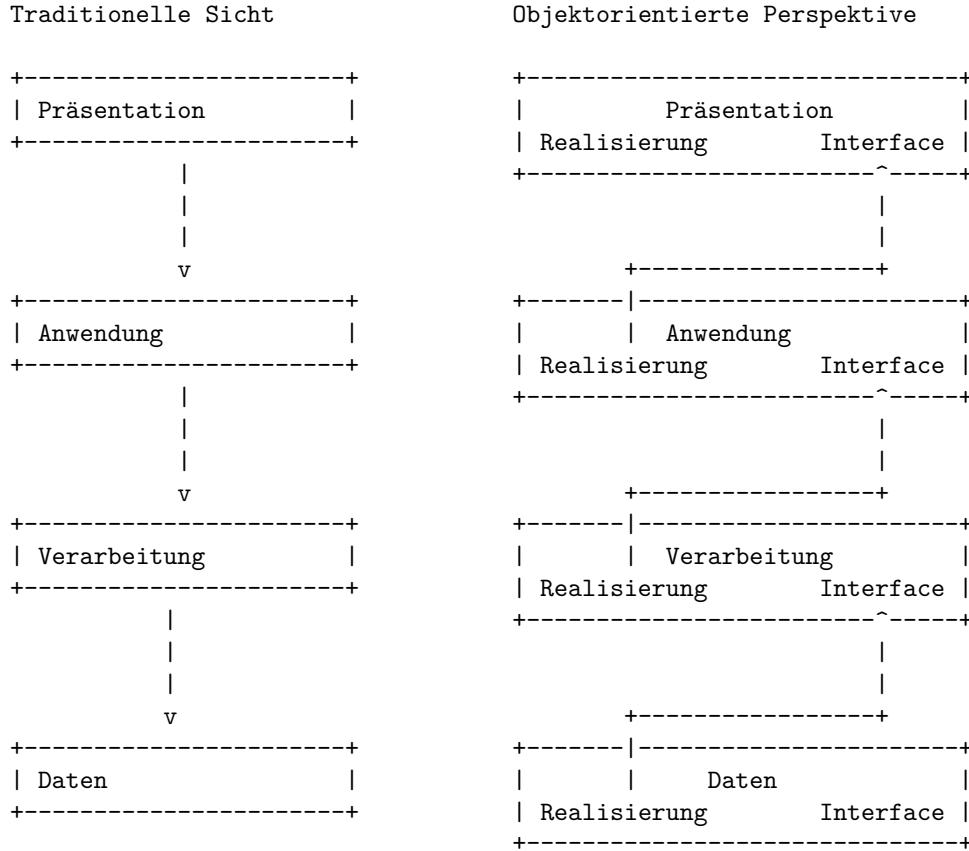
- übersichtlichere kleinere Schnittstellen, die flexibler kombiniert werden können
- Klassen umfassen keine Methoden, die sie nicht benötigen

-> 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 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, verstößen 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 niedrigerwertigeren Klassen implementieren 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/>

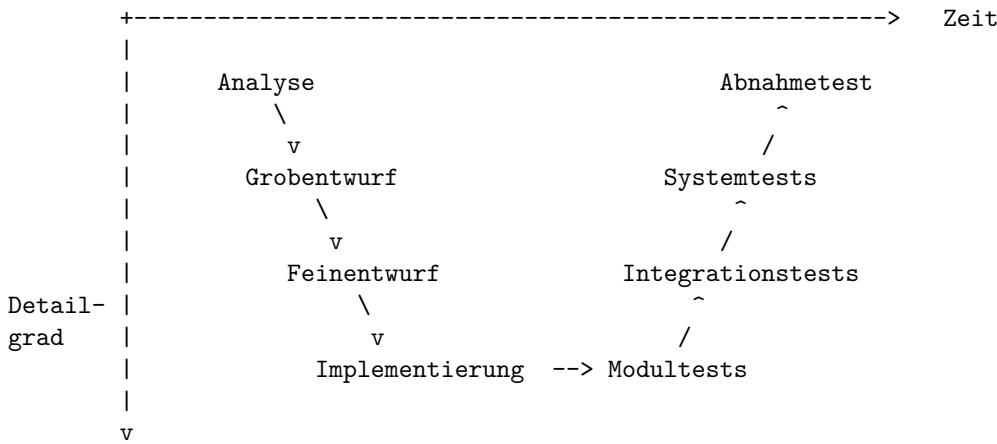
## Herausforderungen bei der Umsetzung der Prinzipien

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 [Website](#). Welche hier scherhaft beschriebenen Herausforderungen sehen Sie im Projekt?

Wie verzahnhen 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

- Quellcodezeugung - 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/>

---

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

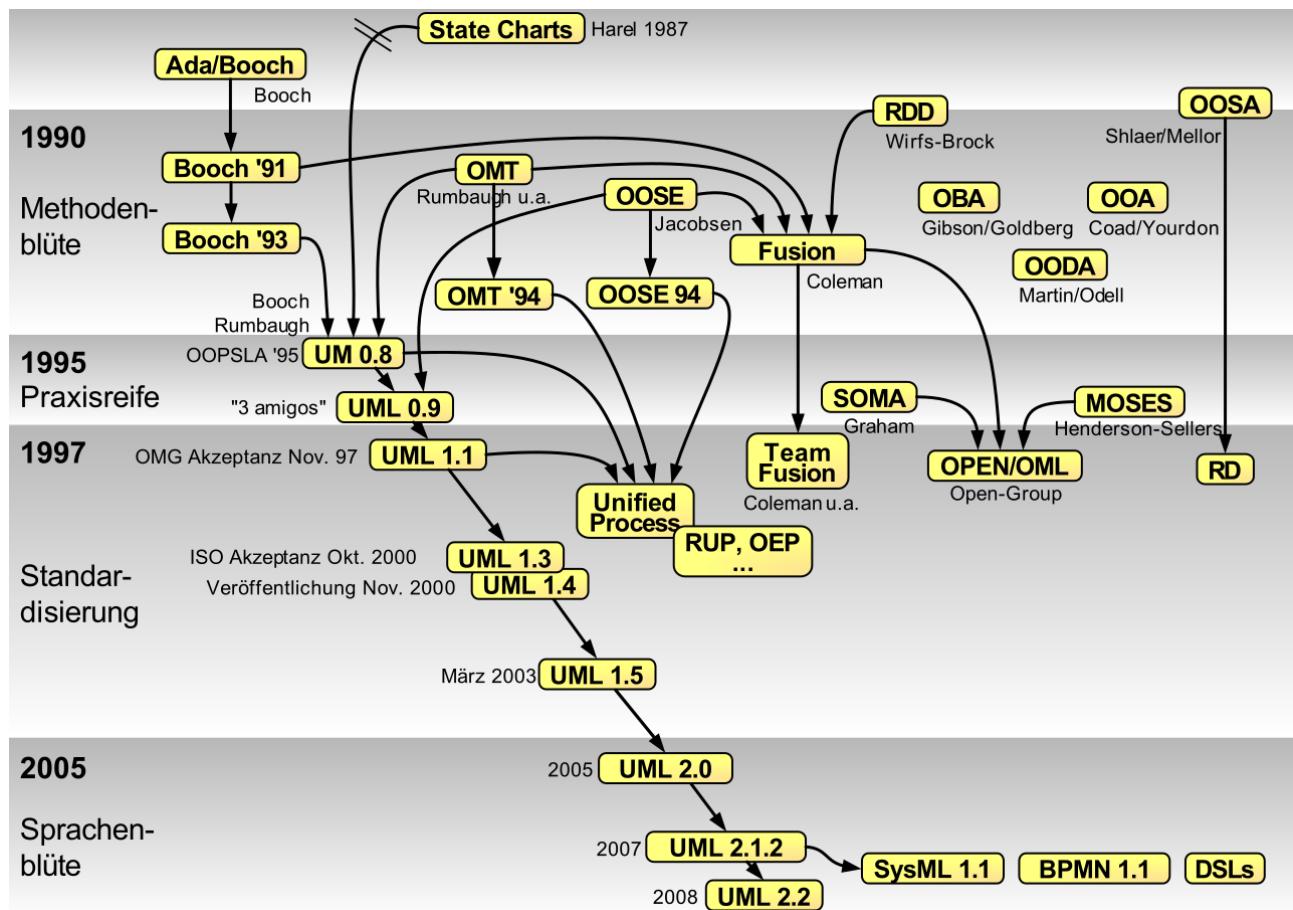


Figure 14.1: OOPGeschichte

Main Page   Namespaces   Classes   Files   [Search](#)

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[int] --> sfd[sfd]
    socket[libsocket::socket] --> keys[keys]
    vector[std::vector<int>] --> elements[elements]
    map[std::map<int, socket*>] --> fdsockmap[fdsockmap]
    fd_set[fd_set] --> writeset[writeset]
    selectset[libsocket::selectset] --> int
    selectset --> socket
    selectset --> vector
    selectset --> map
    selectset --> fd_set
    selectset --> bool
    
```

[\[legend\]](#)

### Public Member Functions

---

```

void add_fd(socket &sock, int method)
std::pair< std::vector< socket * >,
          std::vector< socket * > > wait(long long microsecs=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 file:

Figure 14.2: OOPGeschichte

```

@startuml
class Car

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

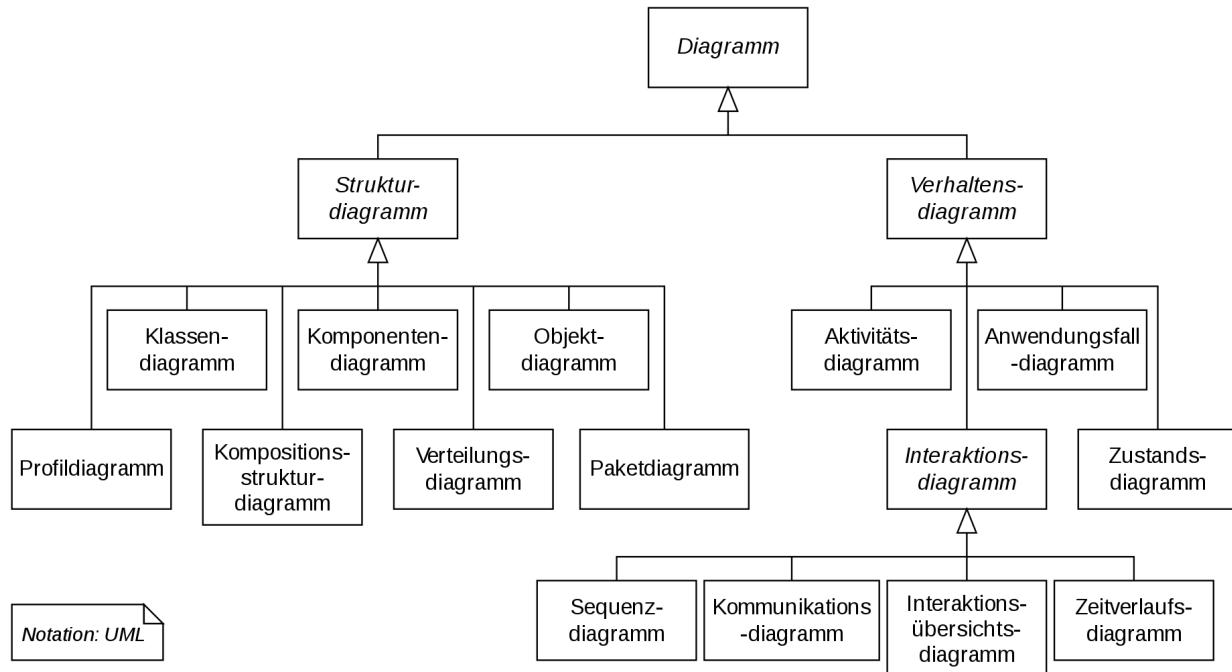


Figure 14.3: OOPGeschichte

## 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?
Kompositionssstrukturdiagramm	Mögliche 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 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.

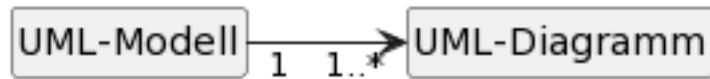


Figure 14.4: Modelle

## Aufgaben

- [ ] Bearbeiten Sie die Aufgabe 3 im GitHub Classroom
- [ ] Experimentieren Sie mit [Umbrello](#)



# Chapter 15

## Modellierung von Software

---

Parameter Kursinformationen

---

**Veranstaltung:** Vorlesung Softwareentwicklung

Semester Sommersemester 2021

Hochschule: Technische Universität Freiberg

Inhalte: Ausgewählte UML Diagrammtypen

Link auf [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/14\\_UML\\_ModellierungII.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/14_UML_ModellierungII.md)

GitHub:

Autoren @author

---

## Neues aus Github

Wann wird gearbeitet?

Stunde	0	1	2	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	nan	2	nan	3	2	nan	nan	1	nan	nan	nan	nan						
2	nan	nan	nan	nan	2	nan	nan	nan	2	8	nan	10	24	3	nan	nan	nan	nan
3	nan	1	4	6	7	25	22	nan	nan	nan	nan	nan						
4	2	nan	nan	2	2	nan	5	8	7	nan	nan	29	16	14	12	nan	2	1
5	nan	nan	1	nan	nan	2	nan	12	10	3	1	2	2	nan	4	3	nan	nan
6	nan	nan	nan	nan	nan	1	nan	nan	nan	1	2	4	7	7	2	nan	nan	4
7	nan	2	4	3	8	5	4	2	4	nan	nan	nan						

Was ist ein guter Commit?

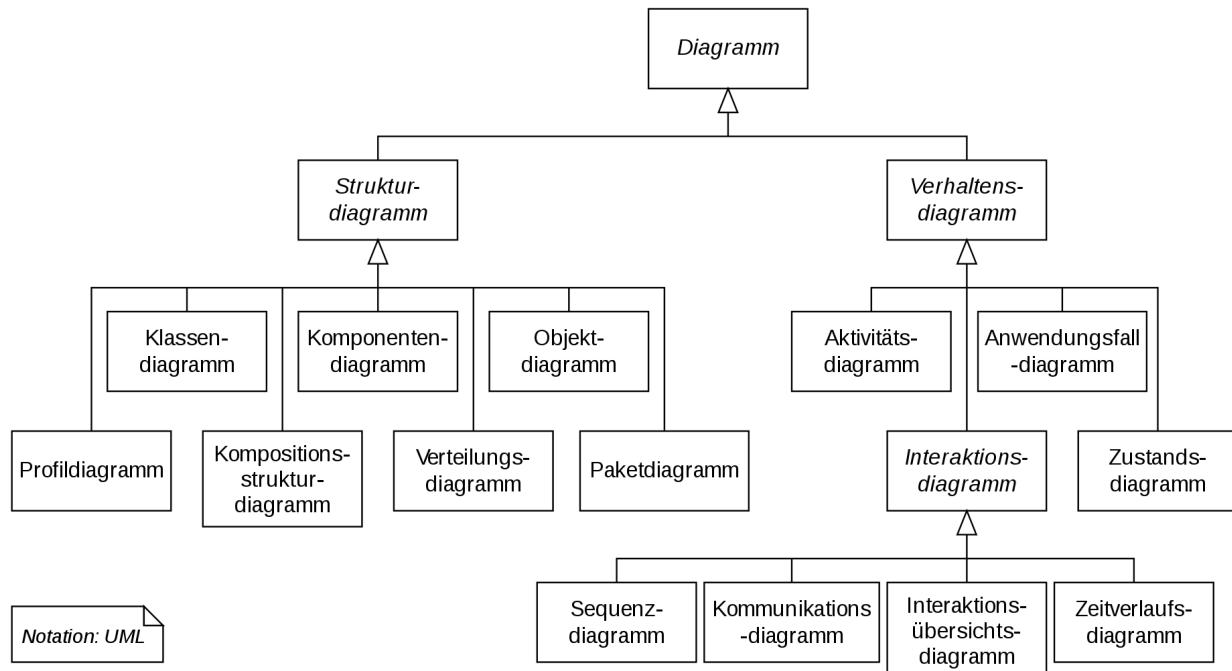
Das folgende Diagramm zeigt die Commits pro Aufgabe

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
3	7	15	12	8	18	10	5	8	45	21	5	8	18	8	6	6	3
4	nan	nan	nan	15	13	18	3	5	5	9	4	3	19	2	3	24	nan

... und wie viele unterschiedliche Dateien wurden dabei editiert?

Task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
3	5	3	2	15	4	5	2	6	3	16	2	2	7	2	2	4	2
4	nan	nan	nan	26	11	7	5	6	8	6	5	7	7	1	2	6	nan

## UML Diagrammtypen



1

Im folgenden werden wir uns aus den beiden Hauptkategorien jeweils folgende Diagrammtypen genauer anschauen:

- Verhaltensdiagramme
  - Anwendungsfall Diagramm
  - Aktivitätsdiagramm
  - Sequenzdiagramm
- Strukturdiagramm
  - Klassendiagramm
  - Objektdiagramm

### Anwendungsfall Diagramm

Das Anwendungsfalldiagramm (Use-Case Diagramm) abstrahiert das erwartete Verhalten eines Systems und wird dafür eingesetzt, die Anforderungen an ein System zu spezifizieren.

Ein Anwendungsfalldiagramm stellt keine Ablaufbeschreibung dar! Diese kann stattdessen mit einem Aktivitäts-, einem Sequenz- oder einem Kollaborationsdiagramm (ab UML 2.x Kommunikationsdiagramm) dargestellt werden.

#### Basiskonzepte

Elemente:

- Systemgrenzen werden durch Rechtecke gekennzeichnet.
- Akteure werden als „Strichmännchen“ dargestellt, dies können sowohl Personen (Kunden, Administratoren) als auch technische Systeme sein (manchmal auch ein Bandsymbol verwendet). Sie ordnen den Symbolen Rollen zu
- Anwendungsfälle werden in Ellipsen dargestellt. Üblich ist die Kombination aus Verb und ein Substantiv **Kundendaten Ändern**.
- Beziehungen zwischen Akteuren und Anwendungsfällen müssen durch Linien gekennzeichnet werden. Man unterscheidet „Association“, „Include“, „Extend“ und „Generalization“.

<sup>1</sup><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

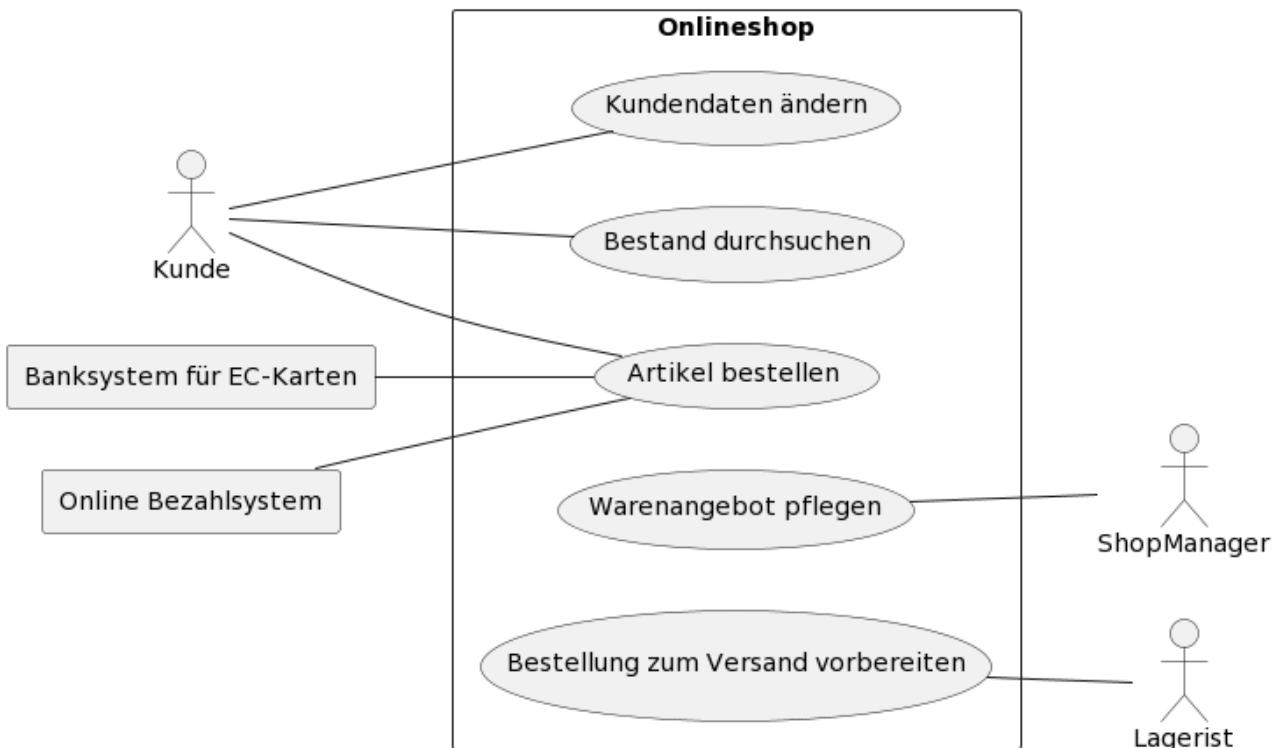


Figure 15.1: Modelle

### Verfeinerung

Use-Case Diagramme erlauben die Abstraktion von Elementen auf der Basis von Generalisierungen. So können Akteure von einander erben und redundante Beschreibungen von Verhalten über **<<extend>>** oder **<<include>>** (unter bestimmten Bedingungen) erweitert werden.

	<b>&lt;&lt;include&gt;&gt;</b> Beziehung	<b>&lt;&lt;extend&gt;&gt;</b> Beziehung
Bedeutung	Ablauf von A schließt den Ablauf von B immer ein	Ablauf von A kann optional um B erweitert werden
Anwendung	Hierachische Zerlegung	Abbildung von Sonderfällen
Abhängigkeiten	A muss B bei der Modellierung berücksichtigen	Unabhängige Modellierung möglich

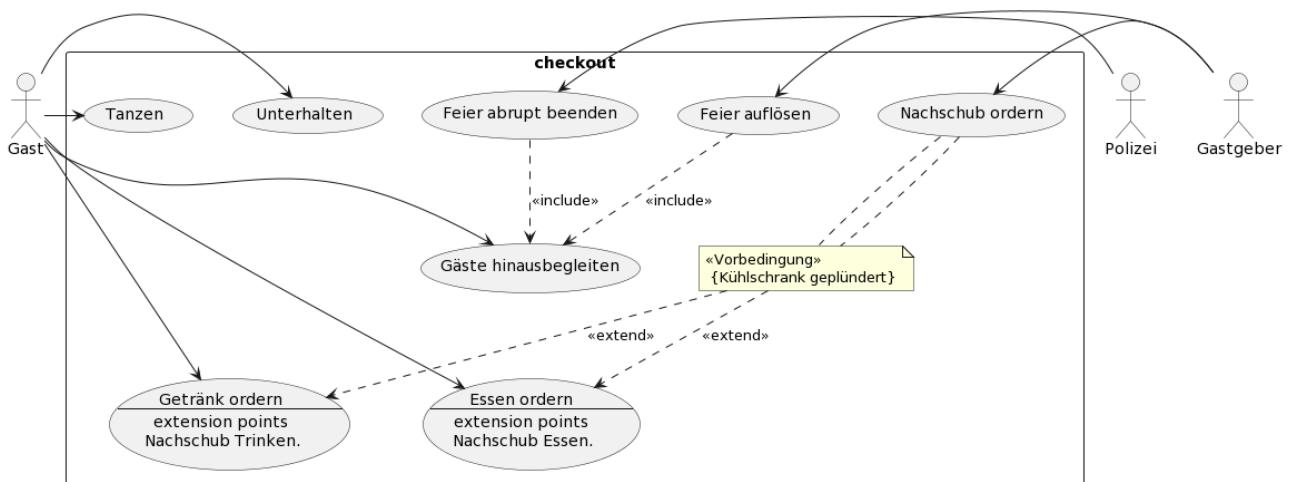


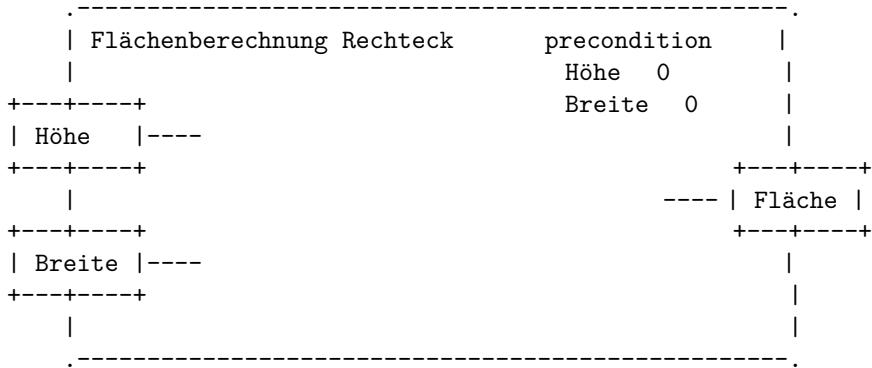
Figure 15.2: PartyUCD

### Anwendungsfälle

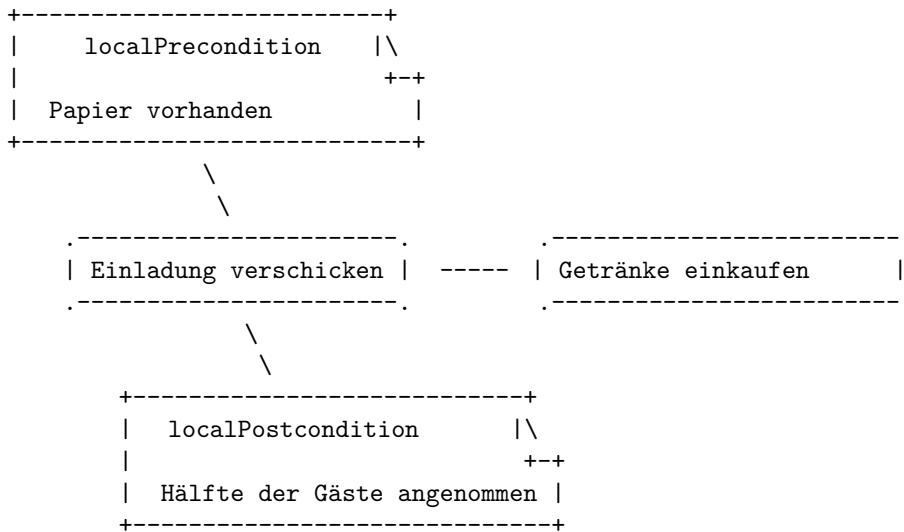
- Darstellung der wichtigsten Systemfunktionen
- Austausch mit dem Anwender und dem Management auf der Basis logischer, handhabbarer Teile
- Dokumentation des Systemüberblicks und der Außenschnittstellen
- Identifikation von Anwendungsfällen

UML2 strukturiert das Konzept der Aktivitätsmodellierung neu und führt als übergeordnete Gliederungsebene Aktivitäten ein, die Aktionen, Objektknoten sowie Kontrollelemente der Ablaufsteuerung und verbindende Kanten umfasst. Die Grundidee ist dabei, dass neben dem Kontrollfluss auch der Objektfluss modelliert wird.

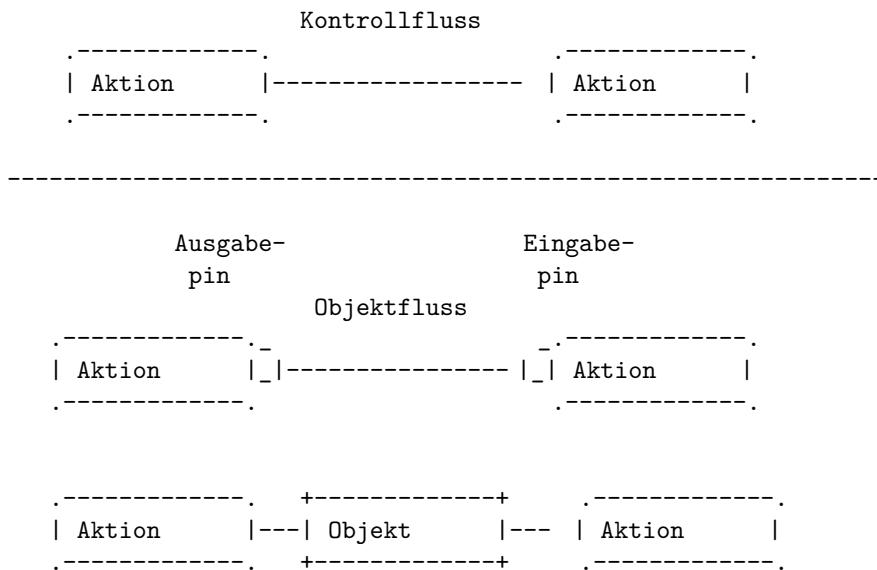
- Aktivitäten definieren Strukturierungselemente für Aktionen, die durch Ein- und Ausgangsparameter, Bedingungen, zugehörige Aktionen und Objekte sowie einen Bezeichner gekennzeichnet sind.



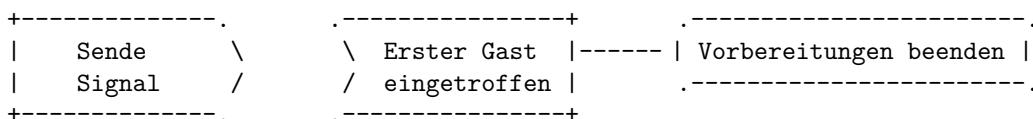
- Aktionen stehen für den Aufruf eines Verhaltens oder die Bearbeitung von Daten, die innerhalb einer Aktivität nicht weiter zerlegt wird.



- Objekte repräsentieren Daten und Werte, die innerhalb der Aktivität manipuliert werden. Damit ergibt sich ein nebeneinander von Kontroll- und Objektfluss.



- Signale und Ereignisse sind die Schnittstellen für das Auslösen einer Aktion



### Beispiel

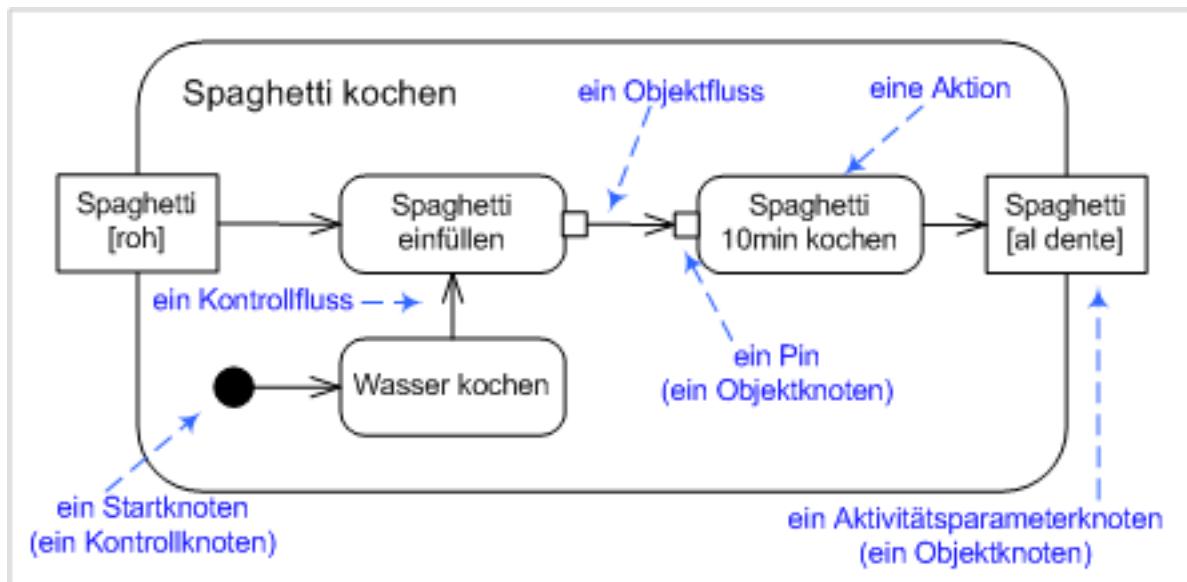


Figure 15.3: Aktivitätsdiagramme

### Anwendungsfälle

- Verfeinerung von Anwendungsfällen (aus den Use Case Diagrammen)
- Darstellung von Abläufen mit fachlichen Ausführungsbedingungen
- Darstellung für Aktionen im Fehlerfall oder Ausnahmesituationen

!?

### Sequenzdiagramm

Sequenzdiagramme beschreiben den Austausch von Nachrichten zwischen Objekten mittels Lebenslinien.

Ein Sequenzdiagramms besteht aus einem Kopf- und einem Inhaltsbereich. Von jedem Kommunikationspartner geht eine Lebenslinie (gestrichelt) aus. Es sind zwei synchrone Operationsaufrufe, erkennbar an den Pfeilen mit ausgefüllter Pfeilspitze, dargestellt. Notationsvarianten für synchrone und asynchrone Nachrichten

Eine Nachricht wird in einem Sequenzdiagramm durch einen Pfeil dargestellt, wobei der Name der Nachricht über den Pfeil geschrieben wird. Nachrichten können:

- Operationsaufrufe einer Klasse sein
- Ergebnisse einer Operation
- Signale
- Interaktionen mit dem Nutzern
- das Setzen einer Variablen

Synchrone Nachrichten werden mit einer gefüllten Pfeilspitze, asynchrone Nachrichten mit einer offenen Pfeilspitze gezeichnet.

Die schmalen Rechtecke, die auf den Lebenslinien liegen, sind Aktivierungsbalken, die den Focus of Control anzeigen, also jenen Bereich, in dem ein Objekt über den Kontrollfluss verfügt, und aktiv an Interaktionen beteiligt ist.

### Beispiel

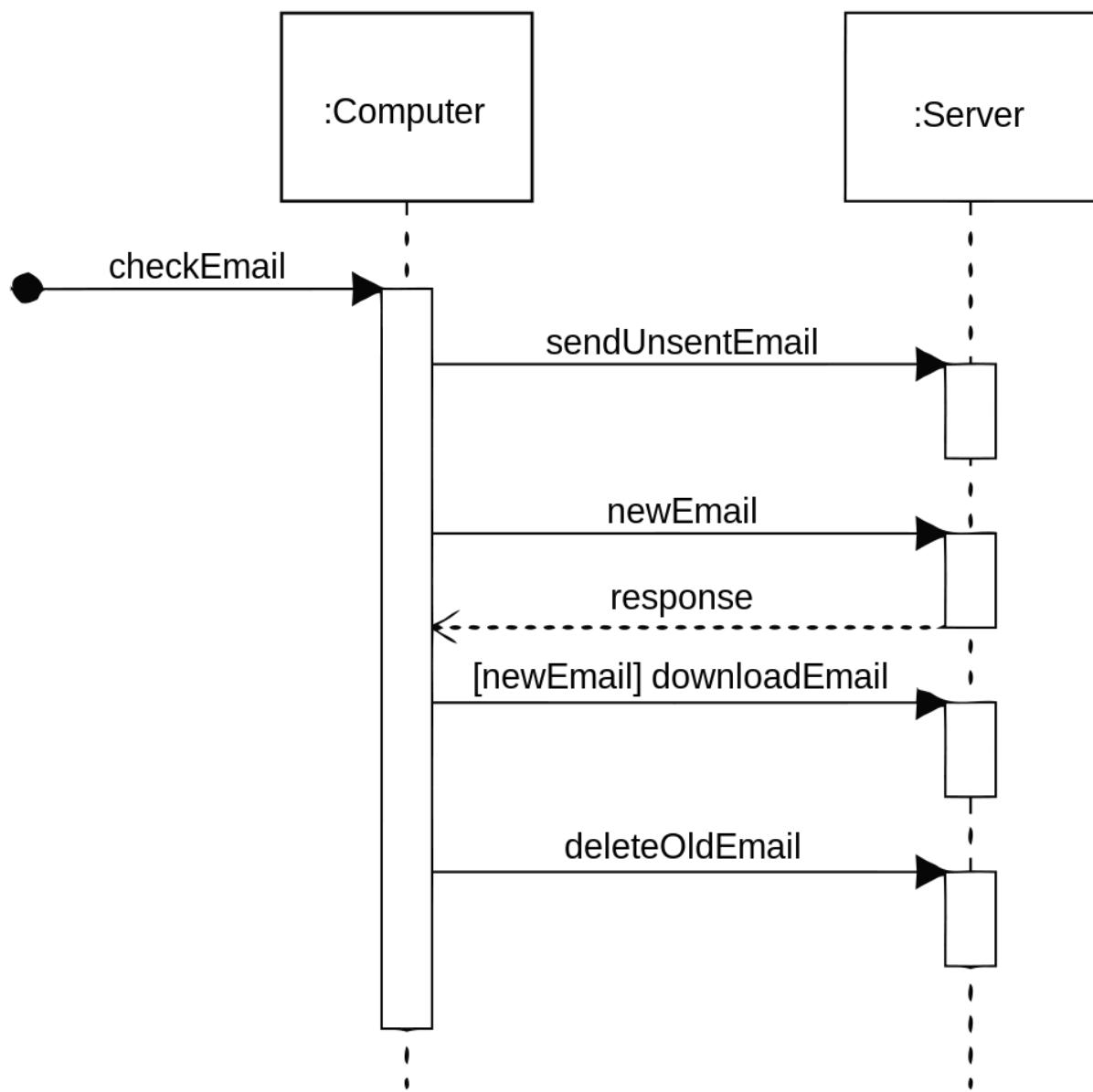


Figure 15.4: Aktivitätsdiagramme

## Bestandteile

Name	Beschreibung
Objekt	Dient zur Darstellung einer Klasse oder eines Objekts im Kopfbereich.
Nachrichtensequenz	Modelliert den Informationsfluss zwischen den Objekten
Aktivitätsbalken	Repräsentiert die Zeit, die ein Objekt zum Abschließen einer Aufgabe benötigt.
Paket	Strukturiert das Sequenzdiagramm
Lebenslinien-	Stellt durch die Ausdehnung nach unten den Zeitverlauf dar.
Symbol	
Fragmente	Kapseln Sequenzen in „Wenn-dann“-Szenarien, optionalen Interaktionen, Schleifen, etc.
Alternativen-	Stellt eine Auswahl zwischen zwei oder mehr Nachrichtensequenzen (die sich in der
Symbol	Regel gegenseitig ausschließen) dar
Interaktionsreferenz	Binden Submodelle und deren Ergebnisse ein deren

## Beispiel

|

|Alkoholkontrolle.plantUML |

## Klassendiagramme

Ein Klassendiagramm ist eine grafische Darstellung (Modellierung) von Klassen, Schnittstellen sowie deren Beziehungen.

## Beispiel

Nehmen wir an, sie planen die Software für ein Online-Handel System. Es soll sowohl verschiedene Nutzertypen (*Customer* und *Administrator*) als auch die Objekt *ShoppingCart* und *Order* umfassen.

```
@startuml
left to right direction
skinparam classAttributeIconSize 0
abstract class User{
    -userId: string
    -password: string
    -email: string
    -loginStatus: string
    +verifyLogin():bool
    +login()
}

class Customer{
    -customerName: string
    +register()
    +updateProfile()
}

class Administrator{
    -adminName: string
    +updateCatlog(): bool
}

class ShoppingCart{
    -cartId: int
    +addCartItem()
    +updateQuantity()
    +checkOut()
}

class Order{
    -orderId: int
}
```

```

customerId: int
shippingId: int
dateCreated: date
dateShipped: date
status: string
+updateQuantity()
+checkOut()

}

class ShippingInfo{
    -shipingId: int
    -shipingType: string
    +updateShipingInfo()
}

User <|-- Customer
User <|-- Administrator
Customer "1" *-- "0..*" ShoppingCart
Customer "1" *-- "0..*" Order
Order "1" *-- "1" ShippingInfo

@enduml

```

## Klassen

Klassen werden durch Rechtecke dargestellt, die entweder nur den Namen der Klasse (fett gedruckt) tragen oder zusätzlich auch Attribute, Operationen und Eigenschaften spezifiziert haben. Oberhalb des Klassennamens können Schlüsselwörter in Guillemets und unterhalb des Klassennamens in geschweiften Klammern zusätzliche Eigenschaften (wie {abstrakt}) stehen.

Elemente der Darstellung :

Eigenschaften	Bedeutung
Attribute	beschreiben die Struktur der Objekte: Bestandteile und darin enthalten Daten
Operationen	Beschreiben das Verhalten der Objekte (Methoden)
Zusicherungen	Bedingungen, Voraussetzungen und Regeln, die die Objekte erfüllen müssen
Beziehungen	Beziehungen einer Klasse zu anderen Klassen

Wenn die Klasse keine Eigenschaften oder Operationen besitzt, kann die unterste horizontale Linie entfallen.

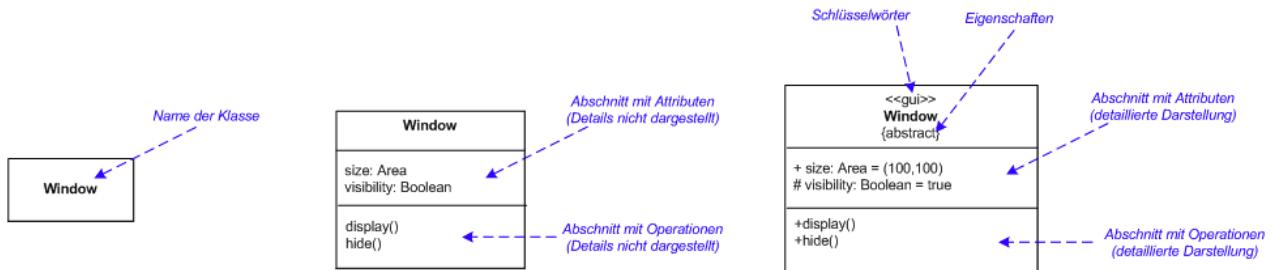


Figure 15.5: OOPGeschichte

## Objekte vs. Klassen

---

Klassendiagramm	Beispielhaftes Objektdiagramm
-----------------	-------------------------------

---

<modeling-language/what-is-object-diagram/>, Autor unbekannt

**Merke:** Vermeiden Sie bei der Benennung von Klassen, Attributen, Operationen usw. sprachspezifische Zeichen

Modellierung in UML

```
@startuml
@enduml
@startuml
skinparam classAttributeIconSize 0

class Zähler{
    +i: int = 12345
}
@enduml
```

Ausführbarer Code in Python 2

```
class Zähler:
    """A simple example class"""
    i = 12345
```

```
A = Zähler()
print(A.i)
```

Ausführbarer Code in C++ 20

```
#include <iostream>
```

```
class Zähler{
public:
    int i = 12345;
};

int main()
{
    Zähler A = Zähler();
    std::cout << A.i;
    return 0;
}
```

## Sichtbarkeitsattribute

Zugriffsmodifizierer	Innerhalb eines Assemblys	Vererbung	Instanzierung	Outerhalb eines Assemblys	Vererbung	Instanzierung
-----	-----	-----	-----	-----	-----	-----
`public`	ja	ja	ja	ja	ja	ja
`private`	nein	nein	nein	nein	nein	nein
`protected`	ja	nein	ja	ja	nein	nein
`internal`	ja	ja	ja	nein	nein	nein
`internal protected`	ja	ja	ja	ja	ja	nein

### public, private

Die Sichtbarkeitsattribute `public` und `private` sind unabhängig vom Vererbungs-, Instanzierungs- oder Paketstatus einer Klasse. Im Beispiel kann der `TrafficOperator` nicht auf die Geschwindigkeiten der Instanzen von `Car` zurückgreifen.

### protected

Die abgeleitete Klassen `Bus` und `PassengerCar` erben von `Car` und übernehmen damit deren Methoden. Die Zahl der Sitze wird beispielsweise mit ihrem Initialisierungswert von 5 auf 40 gesetzt. Zudem muss die Methode `StopAtStation` auch auf die Geschwindigkeit zurückgreifen können.

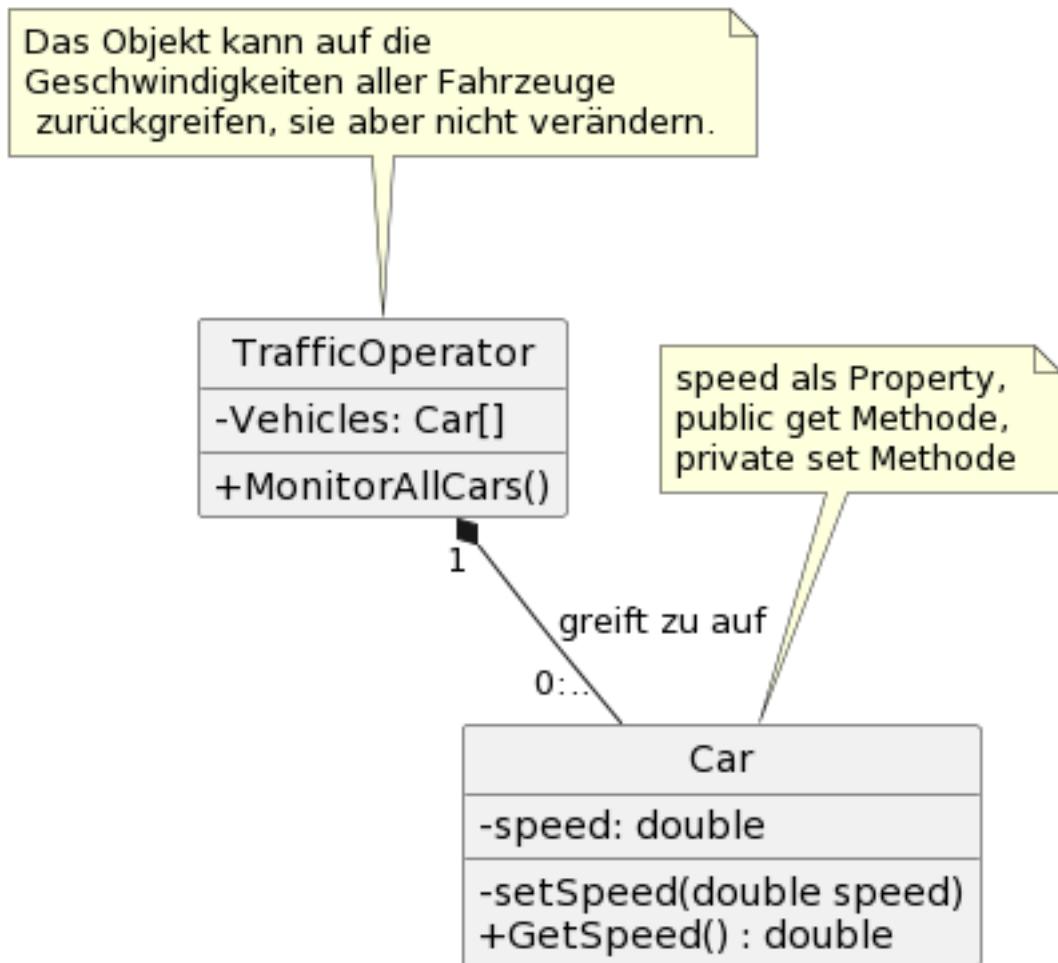


Figure 15.6: PublicPrivate

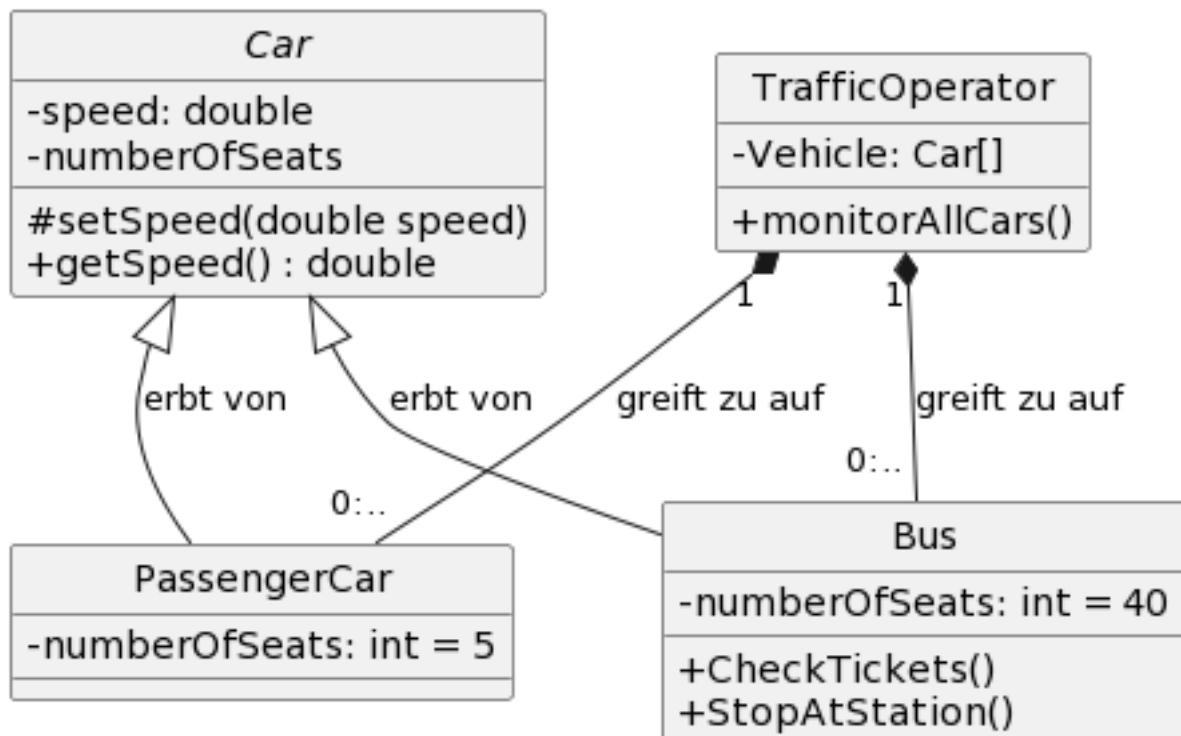


Figure 15.7: Protected

### internal

Ein Member vom Typ `protected internal` einer Basisklasse kann von jedem Typ innerhalb seiner enthaltenden Assembly aus zugegriffen werden.

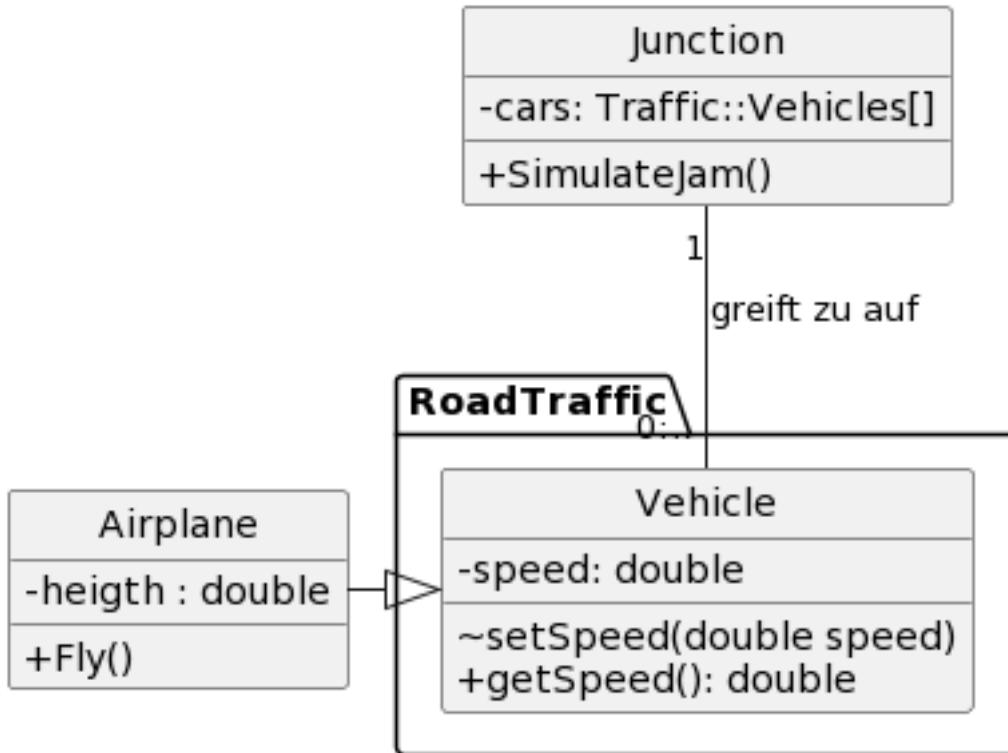


Figure 15.8: Protected

**Merke:** Der UML Standard kennt nur `+ public`, `- private`, `# protected` und `~ internal`. Das C# spezifische `internal protected` ist als weitere Differenzierungsmöglichkeit nicht vorgesehen.

### Attribute

**Merke:** In der C# Welt sprechen wir bei Attributen von Membervariablen und Feldern.

Im einfachsten Fall wird ein Attribut durch einen Namen repräsentiert, der in der Klasse eindeutig sein muss - Die Klasse bildet damit den Namensraum der enthaltenen Attribute.

Entsprechend der Spezifikation sind folgende Elemente eines Attributes definierbar:

[Sichtbarkeit] [/] Name [: Typ] [ Multiplizität ] [= Vorgabewert] [{Eigenschaftswert}]

- *Sichtbarkeit* ... vgl. vorheriger Absatz Das “/” bedeutet, dass es sich um ein abgeleitetes Attribut handelt, dessen Daten von anderen Attributen abhängt
- *Name* ... des Attributes, Leer und Sonderzeichen sollten weggelassen werden, um zu vermeiden, dass Sie bei der Implementierung Probleme generieren.
- *Typ* ... UML verwendet zwar einige vordefinierte Typen (Integer, String, Boolean) beinhaltet aber keine Einschränkungen zu deren Wertebereich!
- *Multiplizität* ... die Zahlenwerte in der rechteckigen Klammer legen eine Ober- und Untergrenze der Anzahl (Kardinalitäten) von Instanzen eines Datentyps fest.

Beispiel	Bedeutung
0..1	optionales Attribut, das aber höchstens in einer Instanz zugeordnet wird
1..1	zwingendes Attribut
0..n	optionales Attribut mit beliebiger Anzahl
1..*	zwingend mit beliebiger Anzahl größer Null
n..m	allgemein beschränkte Anzahl größer 0

- *Vorgabewerte* ... definieren die automatische Festlegung des Attributes auf einen bestimmten Wert
- *Eigenschaftswerte* ... bestimmen die besondere Charakteristik des Attributes

Eigenschaft	Bedeutung
<code>readOnly</code>	unveränderlicher Wert
<code>subsets</code>	definiert die zugelassen Belegung als Untermenge eines anderen Attributs
<code>redefines</code>	überschreiben eines ererbten Attributes
<code>ordered</code>	Inhaltes eines Attributes treten in geordneter Reihenfolge ohne Dublikate auf
<code>bag</code>	Attribute dürfen ungeordnet und mit Dublikaten versehen enthalten sein
<code>sequence</code>	legt fest, dass der Inhalt sortiert, aber ohne Dublikate ist
<code>composite</code>	

Daraus ergeben sich UML-korrekte Darstellungen

Attributdeklaration	Korrekt	Bemerkung
<code>public zähler:int</code>	ja	Umlaute sind nicht verboten
<code>/ alter</code>	ja	Datentypen müssen nicht zwingend angegeben werden
<code>privat adressen:</code>	ja	Menge der Zeichenketten
<code>String [1..*]</code>		
<code>protected bruder Person</code>	ja	Datentyp kann neben den Basistypen jede andere Klasse oder eine Schnittstelle sein
<code>String</code>	nein	Name des Attributes fehlt
<code>privat, public name: String</code>	nein	Fehler wegen mehrfachen Zugriffsattributen

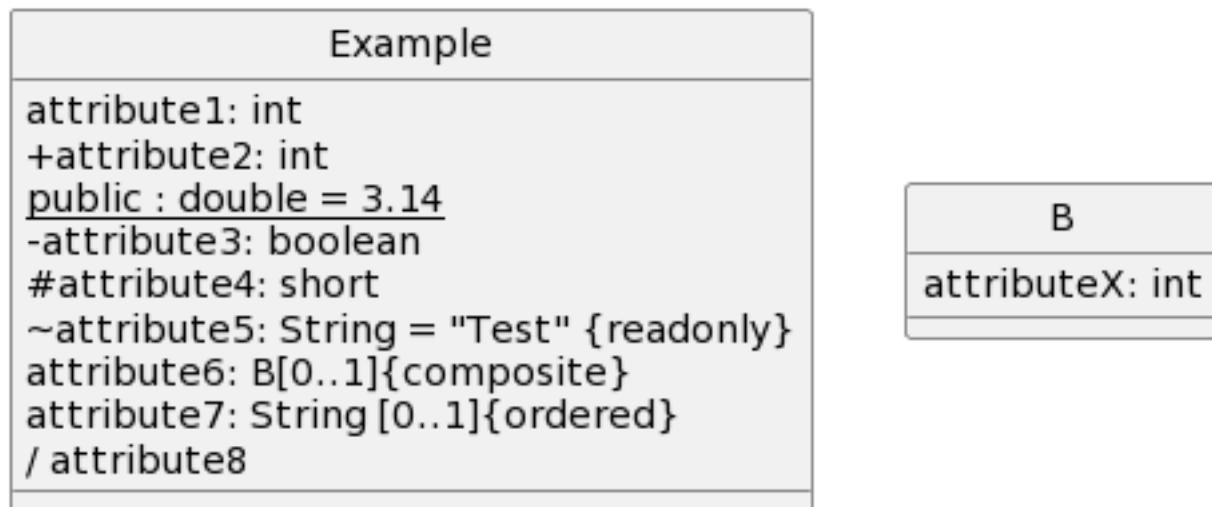


Figure 15.9: Protected

```

using System;

namespace Rextester
{
    class Example
    {
        int attribute1;
        public int attribute2;
        public static double pi = 3.14;
        private bool attribute3;
        protected short attribute4;
        internal const string attribute5 = "Test";
    }
}

```

```
B attribute6;
System.Collections.Specialized.StringCollection attribute7;
private int wert;
Object attribute8{
    get{return wert * 10;}
}
}
```

## Operationen

**Merke:** In der C# Welt sprechen wir bei Operationen von Methoden.

Operationen werden durch mindestens ihren Namen sowie wahlfrei weitere Angaben definiert. Folgende Aspekte können entsprechend der UML Spezifikation beschrieben werden:

[Sichtbarkeit] Name (Parameterliste) [: Rückgabetyp] [{Eigenschaftswert}]

Dabei ist die Parameterliste durch folgende Elemente definiert:

[Übergaberichtung] Name [: Typ] [Multiplizität] [= Vorgabewert] [{Eigenschaftswert}]

- *Sichtbarkeit* ... analog Attribute
  - *Name* ... analog Attribute
  - *Parameterliste* ... Aufzählung der durch die aufrufende Methode übergebenden Parameter, die im folgenden nicht benannten Elementen folgend den Vorgaben, die bereits für die Attribute erfasst wurden:
    - *Übergaberichtung* ... Spezifiziert die Form des Zugriffes (*in* = nur lesender Zugriff, *out* = nur schreibend (reiner Rückgabewert), *inout* = lesender und schreibender Zugriff)
    - *Vorgabewert* ... default-Wert einer Übergabevariablen
  - *Rückgabetyp* ... Datentyp oder Klasse, der nach der Operationsausführung zurückgegeben wird.
  - *Eigenschaftswert* ... Angaben zu besonderen Charakteristika der Operation

### Example

```
+operation1()
-operation2(in param1: int = 5): int {readonly}
#operation3(inout param2 : C)
~operation4(out param3: String [1..*] {ordered}): B
```

Figure 15.10: Protected

```
using System;

class Example
{
    public static void operation1(){
        // Implementierung
    }

    private int operation2 (int param1 = 5)
    {
        // Implementierung
        return value;
    }

    protected void operation3 (ref C param3)
    {
    }
}
```

```

// Implementierung
param3 = ...

internal B operation4 (out StringCollection param3)
{
    // Implementierung
    return value;
}
}
}

```

## Schnittstellen

Eine Schnittstelle wird ähnlich wie eine Klasse mit einem Rechteck dargestellt, zur Unterscheidung aber mit dem Schlüsselwort `interface` gekennzeichnet.

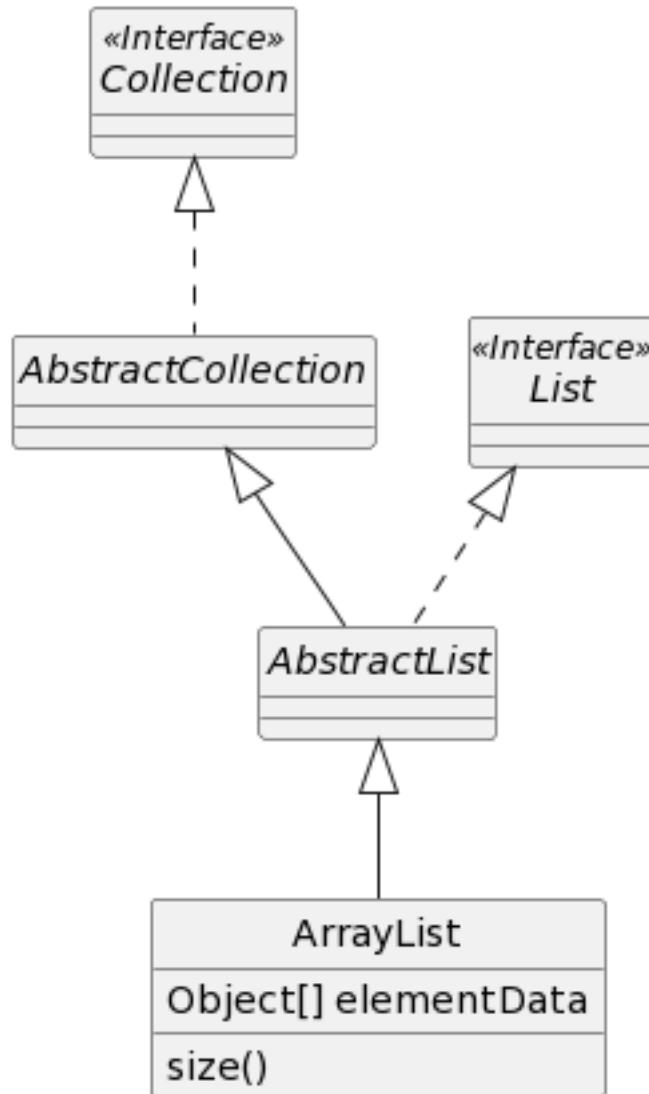


Figure 15.11: Protected

Eine alternative Darstellung erfolgt in der Lollipop Notation, die die grafische Darstellung etwas entkoppelt.

```

using System;

interface Sortierliste{
    void einfuegen (Eintrag e);
    void loeschen (Eintrag e);
}

```

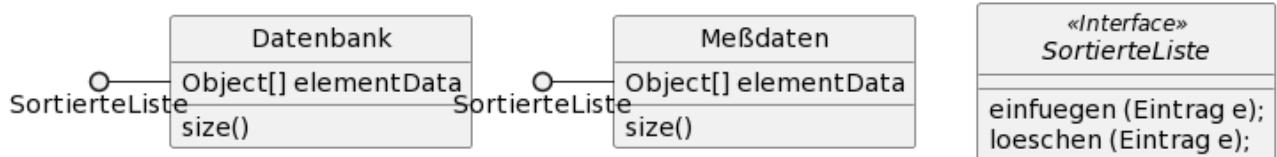


Figure 15.12: Lollipop

```

}
class Datenbank : SortierteListe
{
    void einfuegen (Eintrag e) {//Implementierung};
    void loeschen (Eintrag e) {//Implementierung};
}

```

### Beziehungen

Die Möglichkeiten der Verknüpfung zwischen Klassen und Interfaces lassen sich wie folgt gliedern:

Beziehung	Darstellung	Bedeutung
Generalisierung		gerichtete Beziehung zwischen einer generelleren und einer spezielleren Klasse (Vererbung)
Assoziationen (ohne Anpassung)		beschreiben die Verknüpfung allgemein
Assoziation (Komposition/Aggregation)		Bildet Beziehungen von einem Ganzen und seinen Teilen ab

## Verwendung von UML Tools

Verwendung von Klassendiagrammen

- ... unter Umbrello (UML Diagramm Generierung / Code Generierung)
- ... unter Microsoft Studio [Link](#))

!?[VisualStudio](#)

- ... unter Visual Studio Code mit PlantUML

**Aufgabe:** Wer findet eine “automatisierbare” Lösung?

## Aufgaben

- [ ] Experimentieren Sie mit der automatischen Extraktion von UML Diagrammen für Ihre Computer-Simulation aus den Übungen
- [ ] Evaluieren Sie das Add-On “Class Designer” für die Visual Studio Umgebung

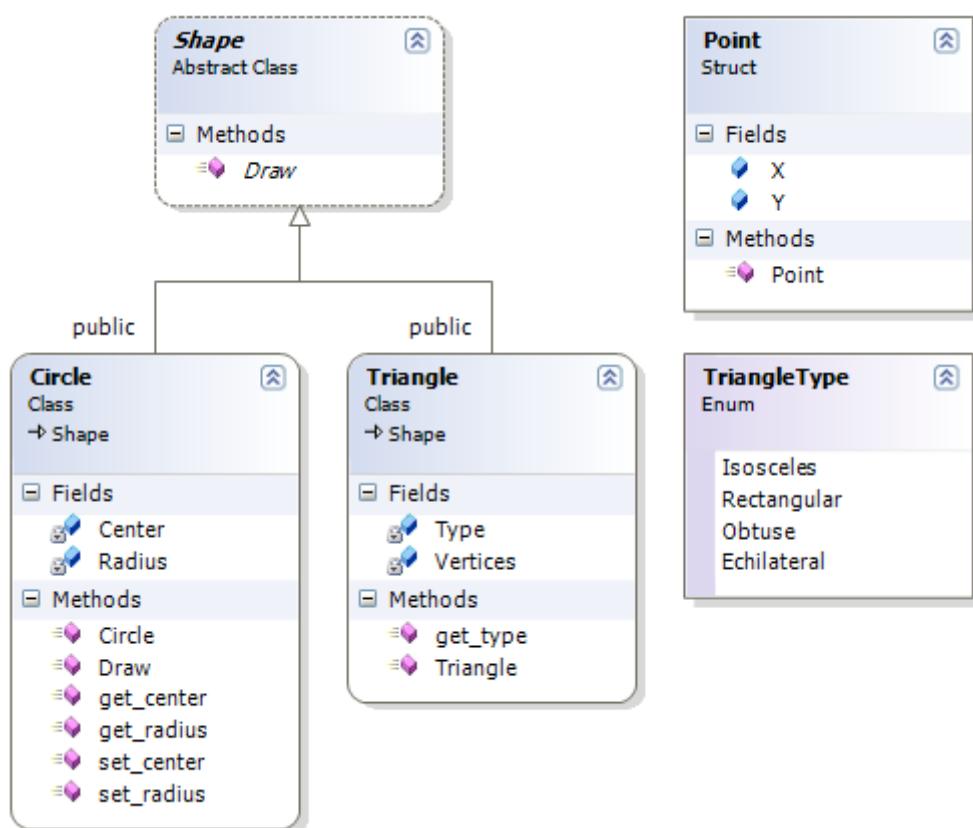


Figure 15.13: ClassDesigner

# Chapter 16

## Modellierung von Software

Parameter	Kursinformationen
<b>Veranstaltung</b>	Vorlesung Softwareentwicklung
Semester	Sommersemester 2021
<b>Hochschule:</b>	Technische Universität Freiberg
<b>Inhalte:</b>	Fallbeispiel UML Modellierung
<b>Link auf den</b>	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/15_UML_ModellierungIII.md">https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/15_UML_ModellierungIII.md</a>
<b>GitHub:</b>	
<b>Autoren</b>	@author

### Beispieldaten UML-Modellierung

Das folgende Beispiel entspricht der Analyse, die auf der Webseite [Link](#) von Boris Schäling beschrieben wird (CC BY-NC-ND 3.0 DE)<sup>1</sup>.

Hier sind auch weitergehende Informationen und Designdiskussionen zu finden. Die entsprechenden Texte sind jeweils in kursiver Schrift gehalten, Ergänzungen für die Einbettung in dieser Vorlesung entsprechend hervorgehoben. Ein zusätzliches Durcharbeiten der Originalunterlagen wird empfohlen.

Der Anforderungskatalog an den beispielhaft zur realisierenden Onlineshop gliedert sich wie folgt:

- Alle im Online-Shop erhältlichen Artikel besitzen eine eindeutige Artikelnummer, einen Namen und einen Preis. Außerdem sollen Artikel aus dem Angebot entfernt werden können, ohne ihre Daten aus dem System zu löschen.
- Während des Bestellvorgangs geben Kunden eine Rechnungs- und Lieferanschrift ein. Grundsätzlich wird zwischen diesen beiden Anschriften nicht unterschieden. Wenn ein Kunde es verlangt, kann er jedoch unterschiedliche Adressen für Rechnung und Lieferung angeben.
- Die Rechnungs- und Lieferanschrift bestehen aus der Anrede, die Kunden aus einer Liste wählen, dem Vor- und Nachnamen, der Straße, Hausnummer, Postleitzahl und dem Ort. Da Bestellungen europaweit ausgeliefert werden, wählen Kunden außerdem ihr Land aus einer Liste aus. Für die Zustellung über einen Paketdienst ins Ausland ist es notwendig, dass Kunden auch ihre Telefonnummer angeben. Darüber hinaus müssen sie ihre Email-Adresse angeben, an die am Ende des Bestellvorgangs eine Bestätigungsmail geschickt wird.
- Kunden dürfen wählen, auf welche Weise sie ihre Bestellung bezahlen möchten. Als Zahlungsmethoden stehen Nachnahme, Bankeinzug und Vorauskasse zur Verfügung. Je nach gewählter Zahlungsmethode fallen zusätzliche Kosten an. Außerdem muss beachtet werden, dass je nach Land, in das die Bestellung ausgeliefert werden soll, unterschiedlich hohe Kosten für Zahlungsmethoden anfallen: Zum Beispiel ist die Nachnahmegebühr im Inland geringer als die Nachnahmegebühr bei einem Versand ins Ausland.

<sup>1</sup>Boris Schäling, "Der moderne Softwareentwicklungsprozess mit UML, Kapitel 3: Das Aktivitätsdiagramm"  
<http://www.highscore.de/uml/titelseite.html>

- Was für Kosten für Zahlungsmethoden gilt, trifft auch auf Verpackungs- und Versandkosten zu: Diese fallen je nach Land unterschiedlich hoch aus.
- Es dürfen nur Bestellungen aus Ländern entgegengenommen werden, bei denen die Kosten von mindestens einer Zahlungsmethode bekannt sind. Ist also für ein Land nicht bekannt, was ein Versand per Nachnahme, Bankeinzug oder Vorauskasse an Kosten verursacht, darf für dieses Land keine Bestellung entgegengenommen werden. Dieses Land darf also nicht bei Eingabe einer Lieferanschrift ausgewählt werden können.
- Es dürfen für eine Bestellung in ein Land nur die Zahlungsmethoden angeboten werden, für die die Kosten der Zahlungsmethoden für dieses Land bekannt sind. Ist zum Beispiel nicht bekannt, was ein Versand per Nachnahme nach Italien kosten würde, darf die Nachnahme als Zahlungsmethode für Bestellungen aus Italien natürlich nicht angeboten werden. Sind lediglich die Kosten für eine Zahlungsmethode bekannt, hat der Kunde keine Wahl und wird automatisch zu dieser Zahlungsmethode weitergeleitet.
- Wenn der Wert einer Bestellung eine festgelegte Grenze überschreitet, werden keine Verpackungs- und Versandkosten in Rechnung gestellt.
- Kundendaten müssen bei der Eingabe auf Vollständigkeit und Richtigkeit überprüft werden, soweit dies technisch möglich ist. Fehlen Daten oder sind Daten falsch angegeben, müssen verständliche Fehlermeldungen ausgegeben werden, die gleichzeitig erklären, wie der Fehler behoben werden kann.
- In vielen Ländern, in die geliefert wird, gibt es einen Distributor, an den Bestellungen im Online-Shop weitergeleitet werden. Das muss auf sicherem Weg geschehen, da mit Kunden- und Bestelldaten natürlich verantwortungsvoll umgegangen wird. Gleichzeitig soll diese Weitergabe jedoch automatisiert sein, um den Aufwand zur Auslieferung von Bestellungen niedrig zu halten. Da diese Distributoren mit unterschiedlichen Softwaresystemen arbeiten, müssen Kunden- und Bestelldaten im jeweils richtigen Format weitergegeben werden.
- Da es nicht für jedes Land, in das geliefert wird, einen dort ansässigen Distributor gibt, sind einige Distributoren für mehrere Länder verantwortlich.

Nach welchen Aspekten lassen die Anforderungen des Kunden strukturieren:

- Datenformate und -inhalte “Die Rechnungsadresse besteht aus Anrede ...”
- Verhalten
  - Kundensicht - “Kunden dürfen wählen”
  - Betreibersicht - “Diese [Versandkosten] fallen ja nach Land unterschiedlich hoch aus”
- Systemparameter

## Use-Case Diagramm

### Basisabläufe

Welche Elemente unserer Anforderungsliste werden in Bezug auf Anwendungsfälle mit dieser Darstellung nicht abgedeckt?

### Verfeinerung des Anwendungsfalldiagramms

- systeminterne Prüfung der Anschriften
- Abbildung von Alternativen und unterschiedlichen Konsequenzen bei der Wahl der Zahlungsmethode

Was fehlt noch?

### Anwendungsfälle des Distributors

Welche Zusammenhänge konnten wir bisher nicht abbilden?

## Aktivitätsdiagramme

Die Aktivitäten rund um die Eingabe der Anschrift lassen sich im einfachsten Fall als Sequenz Eingaben verstehen. Unter UML 1 könnte man das folgendermaßen darstellen:

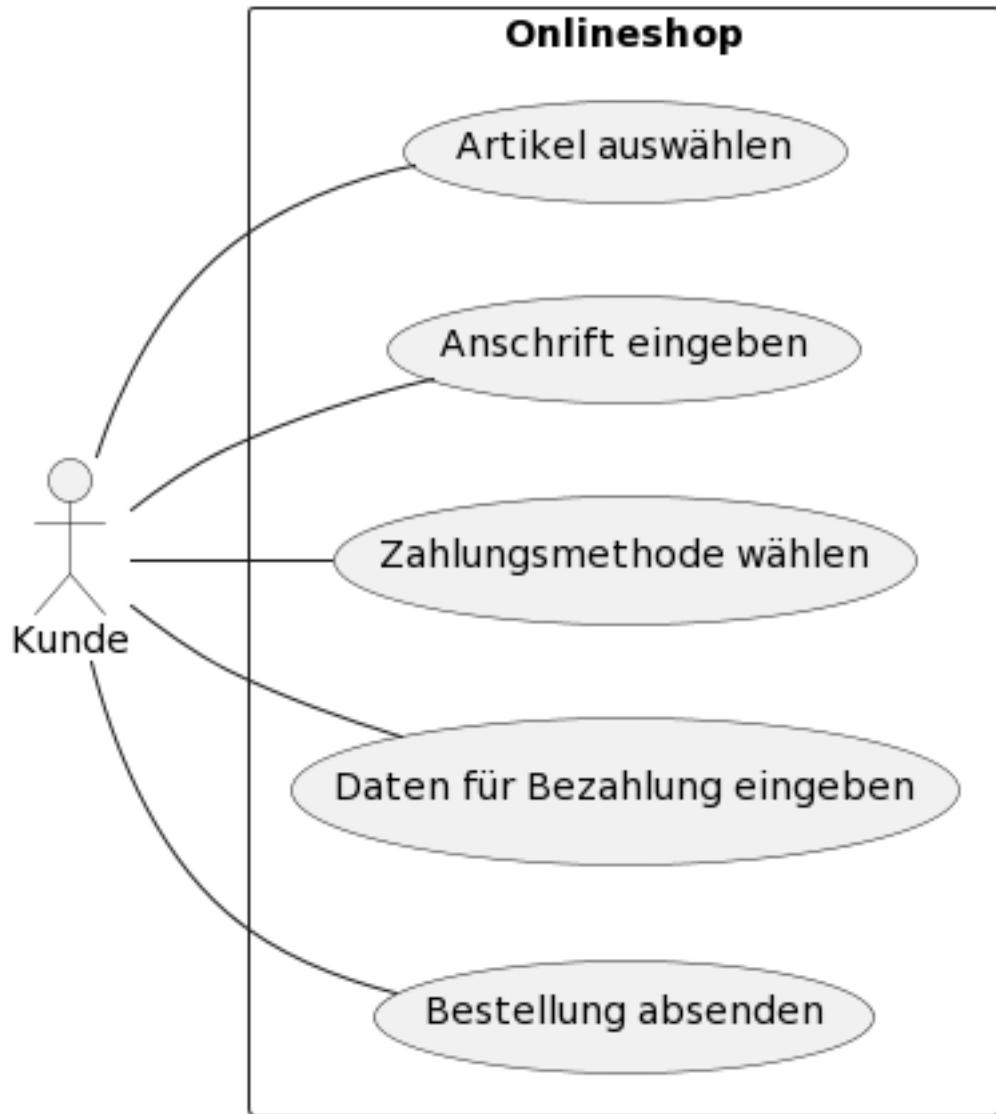


Figure 16.1: UseCaseOnlineShopII

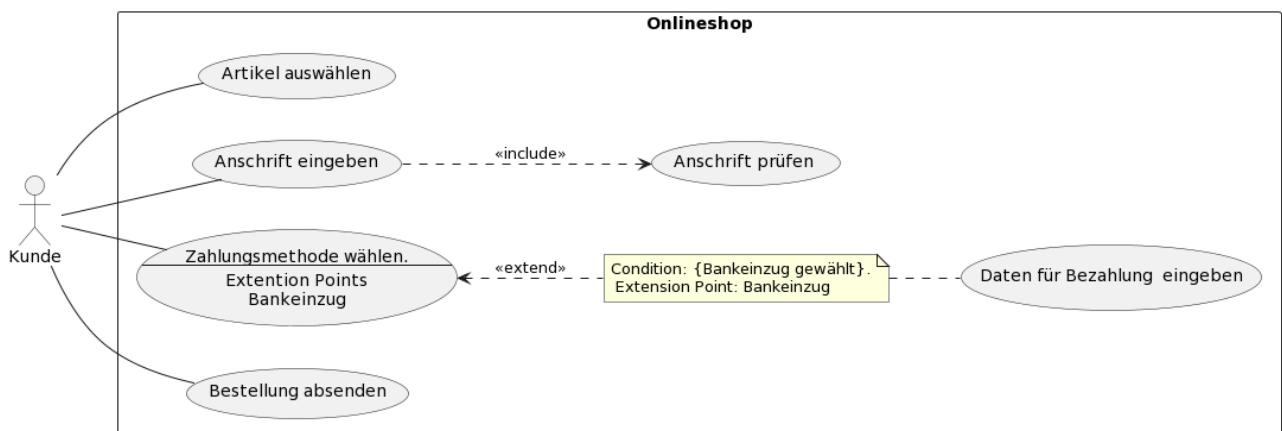


Figure 16.2: UseCaseOnlineShopII

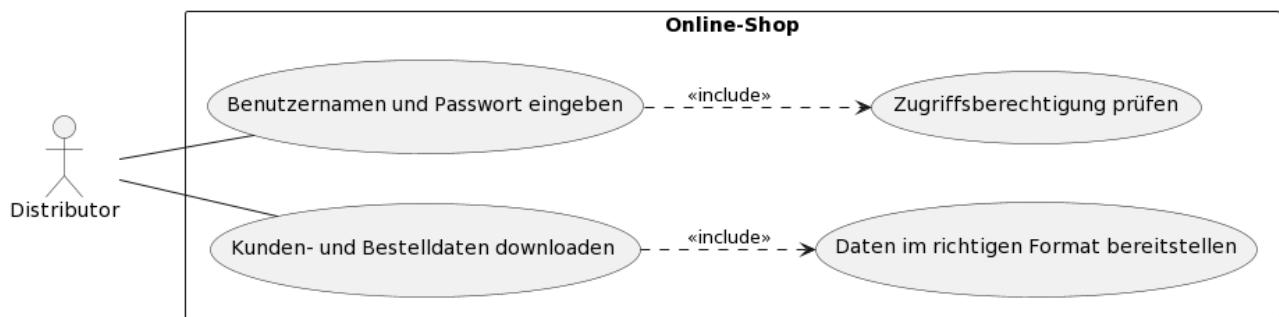
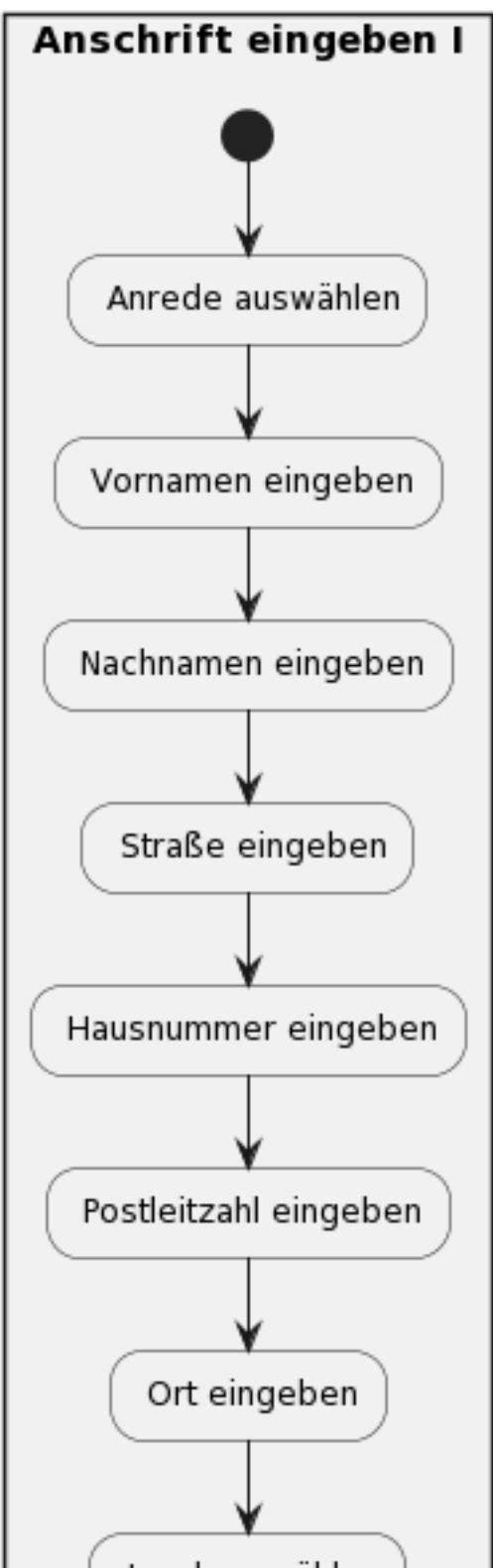


Figure 16.3: UseCaseOnlineShopIII

Variante I

Variante II



Variante I

Variante II

UML 2 für eine stärkere Differenzierung in die Aktivitätsdiagramme ein. Was sind die Hauptmerkmale?

Leider Unterstützt *plantUML* die Features diese Feature noch nicht, so dass hier auf eine Grafik des Original-tutorials zurückgegriffen wird:

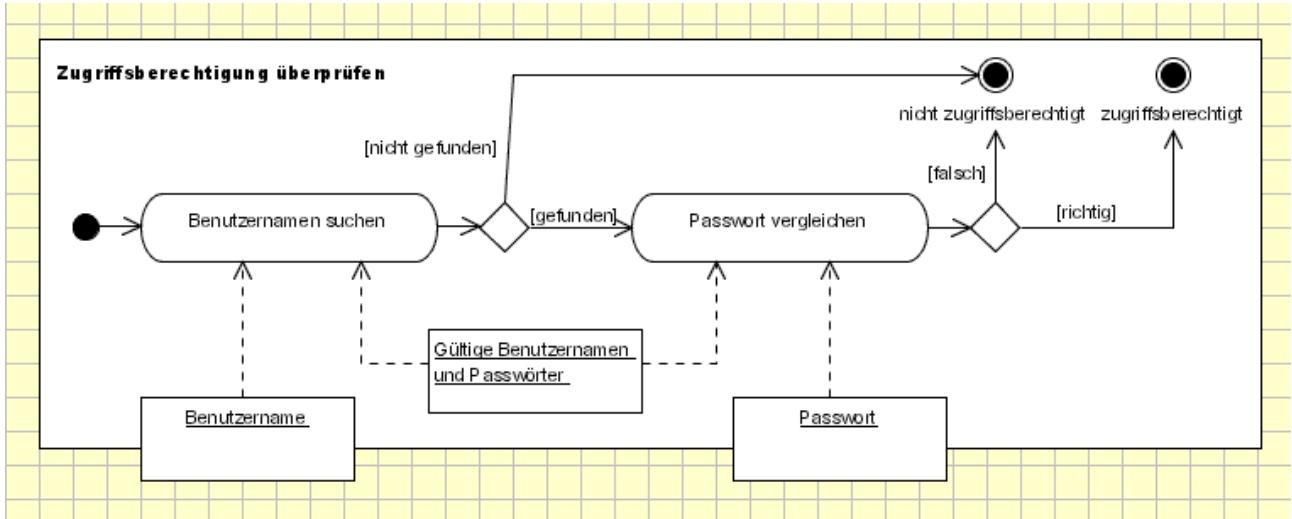


Figure 16.4: ActivityDiagram

## Klassendiagramme

Konzentrieren wir uns zunächst auf einzelne Aspekte der Modellierung, um darauf aufbauend die das gesamte Diagramm zu entwerfen.

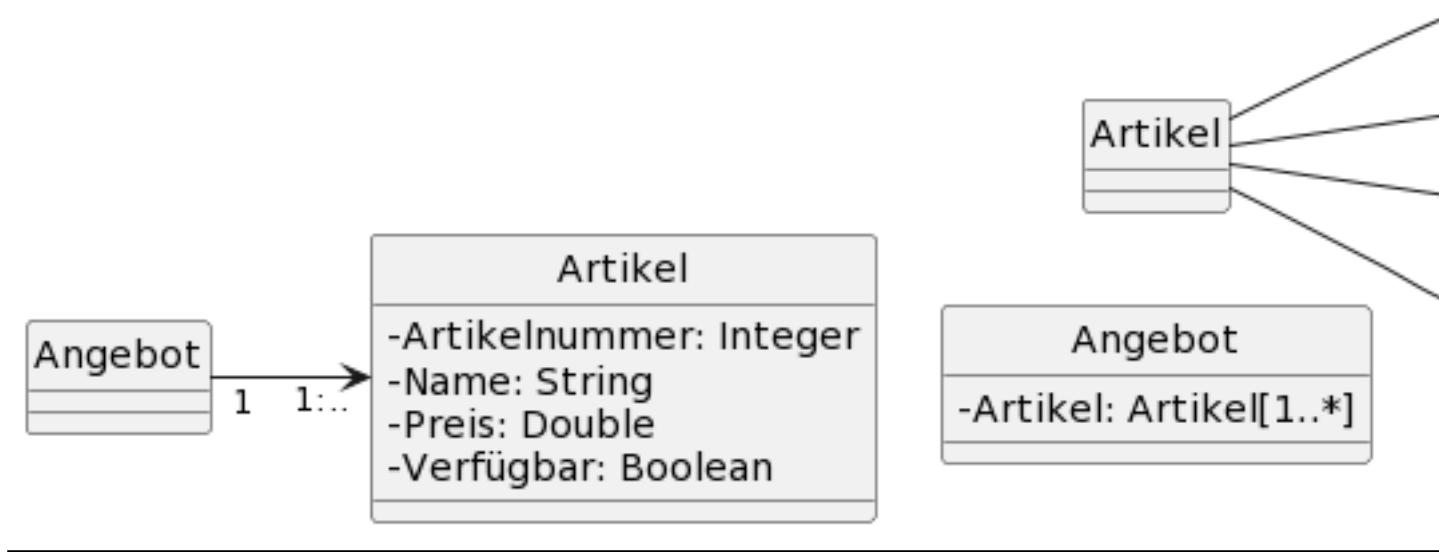
### Ausgangspunkt Angebot / Artikel

*Angebote* sind Sammlungen von *Artikeln* die in einer Kategorie (Sportgerät, Kleidung, usw.) liegen. Wie fassen wir also die Datenanforderungen aus unserer Spezifikation zusammen?

Assoziationen und Eigenschaften sind äquivalent in der Darstellung, entsprechend repräsentieren die folgenden beiden Diagramme eine analoge Aussage.

Variante I

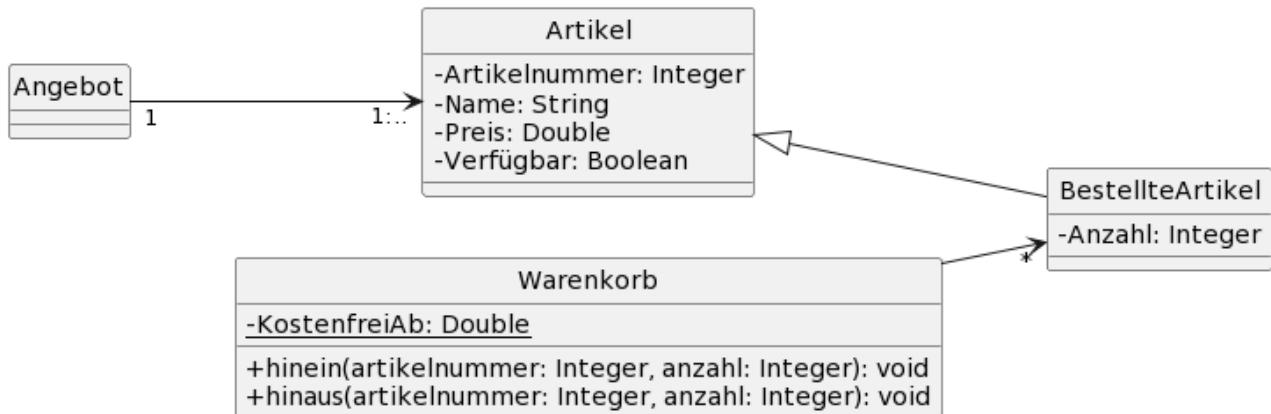
Variante II



### Integration des Warenkorbes

Der Warenkorb bildet die Menge der bestellten Artikel ab, entsprechend wird hier eine 1:n-Beziehung zwischen der Klasse Warenkorb und der Klasse BestellteArtikel definiert. *BestellteArtikel* erbt dabei von *Artikel* und erweitert die Felder um den Eintrag Anzahl.

Zudem bilden wir in der Klasse Warenkorb unsere Anforderung nach einer Bestellumfang gebundenen Versandkostenhöhe ab. Es wird davon ausgegangen, dass der Kunde hier eine Landes- und Versandart und Zahlungsmethodenunabhängige Lösung wünscht. Daher besitzt die Klasse Warenkorb neben der Assoziation zur Klasse Artikel eine statische Eigenschaft KostenfreiAb (diese ist unterstrichen). Die Membervariable gibt einen Euro-Betrag an, ab dem für die Lieferung der bestellten Ware keine Verpackungs- und Versandkosten in Rechnung gestellt werden.

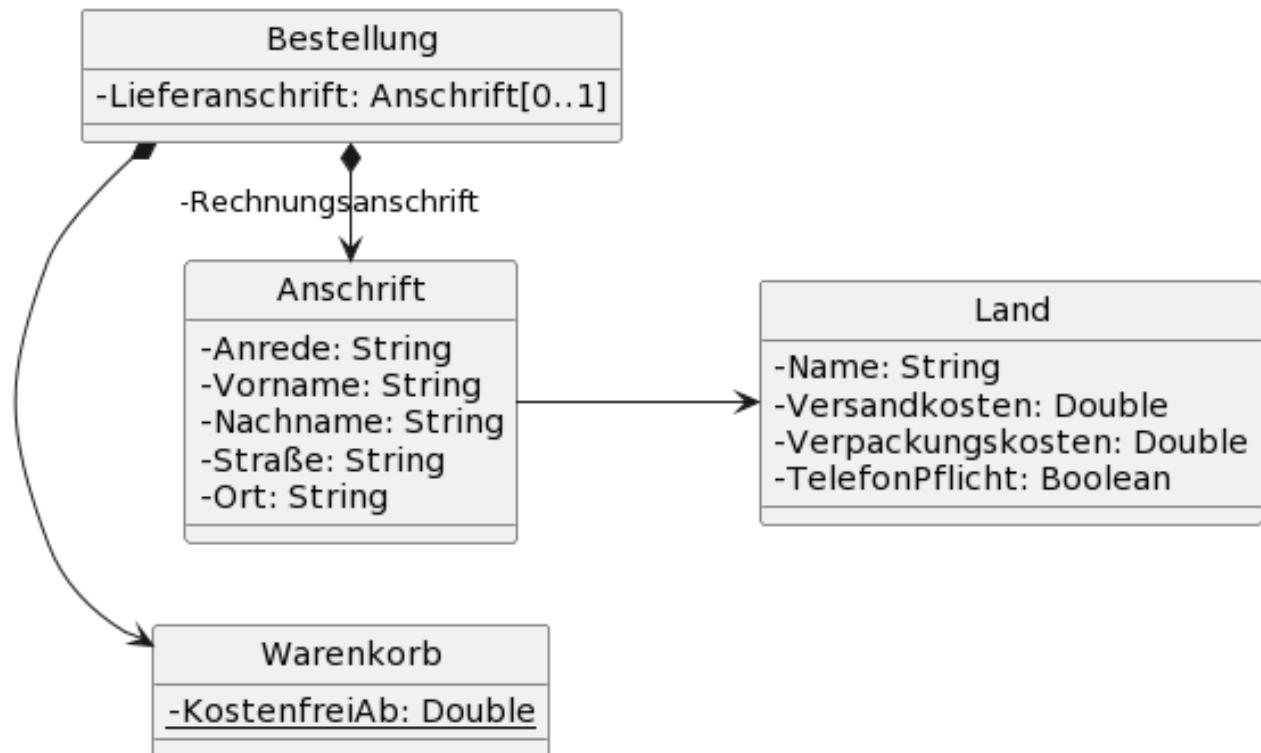


Die Klasse Bestellung umfasst zwei private Eigenschaften vom Typ Anschrift: Lieferanschrift und Rechnungsanschrift. Dabei gibt es genau eine Rechnungsanschrift, eine Lieferanschrift ist aber nicht zwingend ist. Ist keine Lieferanschrift angegeben, wird der Warenkorb an die Rechnungsanschrift ausgeliefert.

Bestellung fasst jeweils ein Objekt vom Typ Warenkorb und eines vom Typ Anschrift zusammen. Anhand der Komposition wird deutlich, dass eine Bestellung ohne die anderen beiden Klassen nicht existieren kann.

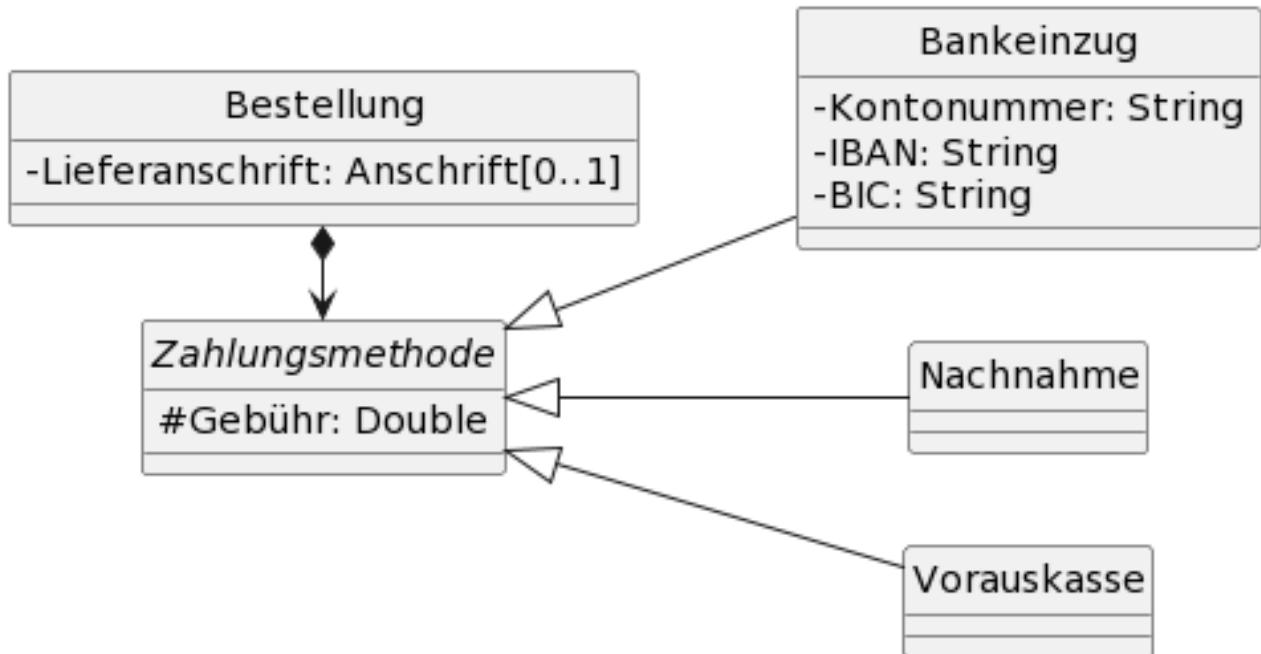
Die Klasse Anschrift enthält zahlreiche private Eigenschaften vom Typ **string**. Bis auf die Telefonnummer sind

alle Angaben Pflicht. Hinzu kommt ein Member vom Objekttyp Land, das neben der nationalen Zuordnung auch die Höhe der Versand- und Verpackungskosten umfasst.



### Zahlungsmethoden

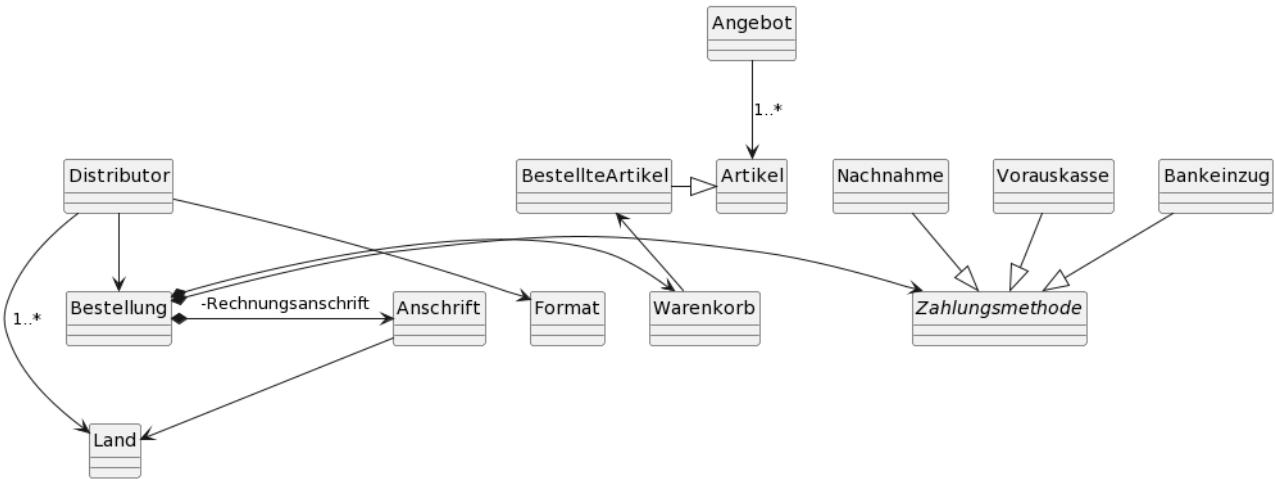
Das Anforderungsset des Kunden beschreibt 3 Zahlungsmethoden für die Abrechnung der Bestellung: Bankeinzug, Nachnahme, Vorauskasse. Lediglich im Falle des Bankeinzuges müssen weitere Daten erhoben werden. Bitte beachten Sie, dass die Basisklasse Zahlungsmethode als abstrakte Klasse definiert wurde, um zu vermeiden, dass davon (sinnlose) Instanzen gebildet werden können.



### Und nun alles zusammen

Was bisher fehlte, war der Distributor, der, wie in unserem Use-Case Diagramm modelliert, per Benutzername und Kennwort auf den Online-Shop zugreifen möchte, um Bestell- und Kundendaten herunterzuladen. Die Klasse Distributor ist entsprechend mit den Klassen Bestellung und Land verknüpft. Diese Assoziationen

drücken aus, dass der Distributor auf beliebig viele Bestellungen zugreifen kann und für mindestens ein Land verantwortlich ist, in das er Bestellungen auszuliefern hat. Die Klasse Format umfasst die Methoden zur Generierung der Ausgabeformate für die Daten, da davon ausgegangen wird, die dass die weitere Bearbeitung mit alternativen Programmen realisiert wird.



# Chapter 17

## Modellierung von Software

---

Parameter Kursinformationen

---

**Veranstaltung:** Vorlesung Softwareentwicklung

Semester Sommersemester 2021

Hochschule: Technische Universität Freiberg

Inhalte: Einbettung des Testens in den Softwareentwicklungsprozess

Link auf [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/16\\_Testen.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/16_Testen.md)

GitHub:

Autoren @author

---

### Organisatorisches

- Alternative Prüfungsleistung für die Veranstaltung Softwareentwicklung
- Prüfungsvorleistung für Einführung in die Softwareentwicklung

### Neues aus der GitHub Woche

Wie stabil sind die Teams?

teamKey	3	4	5
0	[2 3]		
1	[4 5]	[4 5]	
2	[6 7]	[6 7]	
3	[8 9]	[31 8 9]	[8 9]
4	[10 11]	[10 11]	[10 11]
5	[12 13]	[12 13]	
6	[14]	[14]	
7	[15 16]	[15]	
8	[17 18]	[17]	
9	[19 20]	[19 20]	[19]
10	[21 22]	[21 22]	
11	[23 24]	[23 24]	
12	[25 26]	[26 25]	[25 26]
13	[27 28]	[27 28]	
14	[29 30 31]	[29 30]	
15	[32 33]	[32 33]	[33 32]
16	[34]	[34]	[34]
17		[ 2 15]	

## Softwarefehler

Bekannte Softwarefehler und deren Folgen:

- Beim Kampfflugzeug F-16 brachte der Autopilot das Flugzeug in Rückenlage, wenn der Äquator überflogen wurde. Dies kam daher, dass man keine „negativen“ Breitengrade als Eingabedaten bedacht hatte. Dieser Fehler wurde sehr spät während der Entwicklung der F-16 anhand eines Simulators entdeckt und beseitigt.
- 1999 verpasste die NASA-Sonde Mars Climate Orbiter den Landeanflug auf den Mars, weil die Programmierer unterschiedliche Maßsysteme verwendeten (ein Team verwendete das metrische und das andere das angloamerikanische) und beim Datenaustausch es so zu falschen Berechnungen kam. Eine Software wurde so programmiert, dass sie sich nicht an die vereinbarte Schnittstelle hielt, in der die metrische Einheit Newton × Sekunde festgelegt war. Die NASA verlor dadurch die Sonde.
- Zwischen 1985 und 1987 gab es mehrere Unfälle mit dem medizinischen Bestrahlungsgerät Therac-25. Infolge einer Überdosis, die durch fehlerhafte Programmierung und fehlende Sicherungsmaßnahmen verursacht wurde, mussten Organe entfernt werden, und es verstarben drei Patienten.
- Das Jahr-2000-Problem, auch als Millennium-Bug (zu deutsch „Millennium-Fehler“) oder Y2K-Bug bezeichnet, ist ein Computerproblem, das im Wesentlichen durch die Behandlung von Jahreszahlen als zweistellige Angabe innerhalb von Computersystemen entstanden ist.

Softwarefehler sind sowohl sicherheitstechnisch wie ökonomisch ein erhebliches Risiko. Eine Studie der Zeitschrift iX ermittelte 2013 für Deutschland folgende Werte:

- Ca. 84,4 Mrd. Euro betragen die jährlichen Verluste durch Softwarefehler in Mittelstands- und Großunternehmen
- Ca. 14,4 Mrd. Euro jährlich (35,9 % des IT-Budgets) werden für die Beseitigung von Programmfehlern verwendet;
- Ca. 70 Mrd. Euro betragen die Produktivitätsverluste durch Computerausfälle aufgrund fehlerhafter Software

### Was sind Softwarefehler eigentlich?

Ein Programm- oder Softwarefehler ist, angelehnt an die allgemeine Definition für „Fehler“

„Nichterfüllung einer Anforderung“ [EN ISO 9000:2005]

Konkret definiert sich der Fehler danach als

„Abweichung des IST (beobachtete, ermittelte, berechnete Zustände oder Vorgänge) vom SOLL (festgelegte, korrekte Zustände und Vorgänge), wenn sie die vordefinierte Toleranzgrenze [die auch 0 sein kann] überschreitet.“

Im Rahmen dieser Veranstaltung lassen wir Lexikalische Fehler und Syntaxfehler außen vor. Diese sind in der Regel über den Compiler identifizierbar. Darüber hinaus existieren aber :

Fehlertyp	Folgen
Logisch/semantischer Fehler	Ableitung ist zwar syntaktisch fehlerfrei, aber inhaltlich trotzdem fehlerhaft (plus statt minus, kleiner statt kleiner gleich, fehlende Synchronisation, usw.)
Designfehler	Strukturelle Mängel auf der Modul oder Systemebene, die das Zusammenspiel der Komponenten, deren Erweiterung, usw. verhindern.
Fehler im Bedienkonzept	Unintuitive Benutzung, das Programm „fühlt sich komisch an“

Darüber hinaus ist es wichtig zwischen Laufzeit- und Designzeitfehlern zu unterscheiden.

### Wann entstehen Fehler im Projekt?

Problem- und Systemanalyse:

- Die Anforderungen und Qualitätsmerkmale werden nicht festgelegt.
- Es fehlen eindeutige Begriffsdefinitionen.

Systementwurf:

- Die Systemarchitektur ist gar nicht oder nur mit großem Aufwand erweiterbar.
- Das System ist nicht modular aufgebaut, die Daten sind nicht gekapselt.

Feinentwurf:

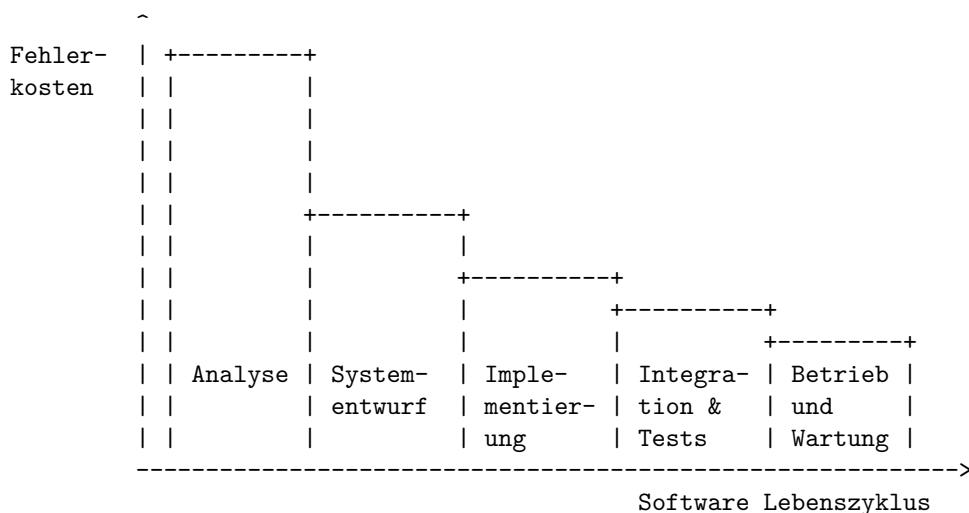
- Schnittstellen sind nicht hinreichend spezifiziert
- Interaktionsmodelle weisen Lücken auf

Codierung

- Programmier-Standards bzw. -Richtlinien werden nicht beachtet.
- Die Namensvergabe ist ungünstig.

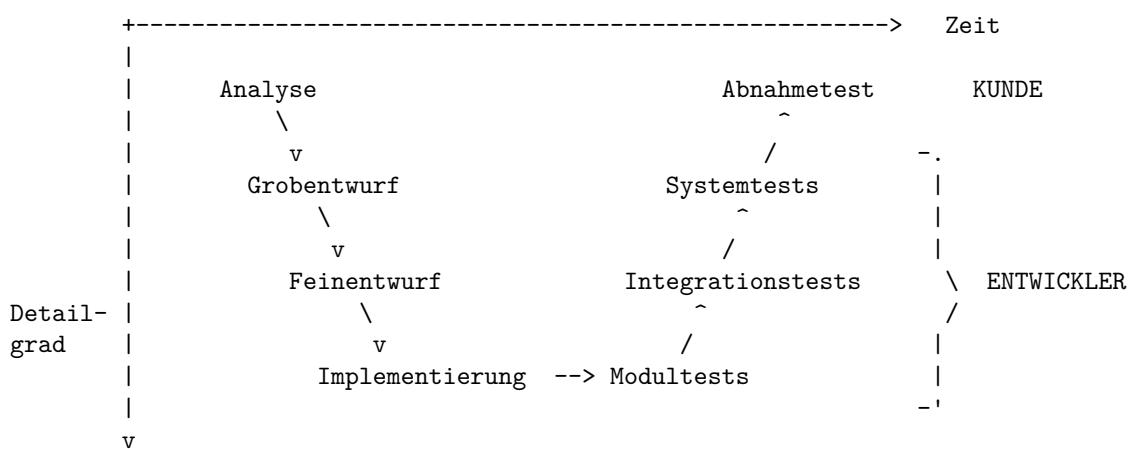
Betrieb und Wartung:

- Die Dokumentation fehlt ganz, ist veraltet oder nicht adäquat.
- Die Schulung der Anwender wird vernachlässigt.
- Das Konfigurationsmanagement ist unzureichend.



## Testen als Teil der Qualitätssicherung

Welche Tests werden in das Projekt integriert?



Der Modultest, auch Komponententest oder Unitest genannt, ist ein Test auf der Ebene der einzelnen Module der Software. Testgegenstand ist die Funktionalität innerhalb einzelner abgrenzbarer Teile der Software (Module, Programme oder Unterprogramme, Units oder Klassen). Testziel dieser häufig durchgeführten Tests ist der Nachweis der technischen Lauffähigkeit und korrekter fachlicher (Teil-) Ergebnisse.

Der Integrationstest bzw. Interaktionstest testet die Zusammenarbeit voneinander abhängiger Komponenten. Der Testschwerpunkt liegt auf den Schnittstellen der beteiligten Komponenten und soll korrekte Ergebnisse über komplexe Abläufe hinweg nachweisen.

Der Systemtest ist die Teststufe, bei der das gesamte System gegen die gesamten Anforderungen (funktionale und nicht-funktionale Anforderungen) getestet wird. Gewöhnlich findet der Test auf einer Testumgebung statt und wird mit Testdaten durchgeführt. Die Testumgebung soll die Produktivumgebung des Kunden simulieren, d. h. ihr möglichst ähnlich sein. In der Regel wird der Systemtest durch die realisierende Organisation durchgeführt.

Ein Abnahmetest, Verfahrenstest, Akzeptanztest oder auch User Acceptance Test (UAT) ist das Testen der gelieferten Software durch den Kunden. Der erfolgreiche Abschluss dieser Teststufe ist meist Voraussetzung für die rechtswirksame Übernahme der Software und deren Bezahlung. Dieser Test kann unter Umständen (z. B. bei neuen Anwendungen) bereits auf der Produktionsumgebung mit Kopien aus Echtdaten durchgeführt werden.

### Warum geht es dann trotzdem schief?

- Es ist angeblich keine Zeit für systematische Tests vorhanden (Termindruck).
- Die Notwendigkeit für systematische Tests wird nicht erkannt.
- Die Tests werden manuell realisiert.
- Die Erstellung von Testspezifikationen für systematische Tests wird nicht entwicklungsbegleitend durchgeführt.
- Die Testebenen weisen eine unterschiedliche Realisierung auf (Modultests top, Systemtests flop)

## Definition

Es gibt unterschiedliche Definitionen für den Softwaretest:

„the process of operating a system or component under specified conditions, observing or recording the results and making an evaluation of some aspects of the system or component.“ [ANSI/IEEE Std. 610.12-1990 ]

„Test [...] der überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen“ <sup>1</sup> ist.

„Unter Testen versteht man den Prozess des Planens, der Vorbereitung und der Messung, mit dem Ziel, die Eigenschaften eines IT-Systems festzustellen und den Unterschied zwischen dem tatsächlichen und dem erforderlichen Zustand aufzuzeigen.“ <sup>2</sup>

Welche Unterschiede sehen Sie in den Definitionen?

## Ablauf beim Testen

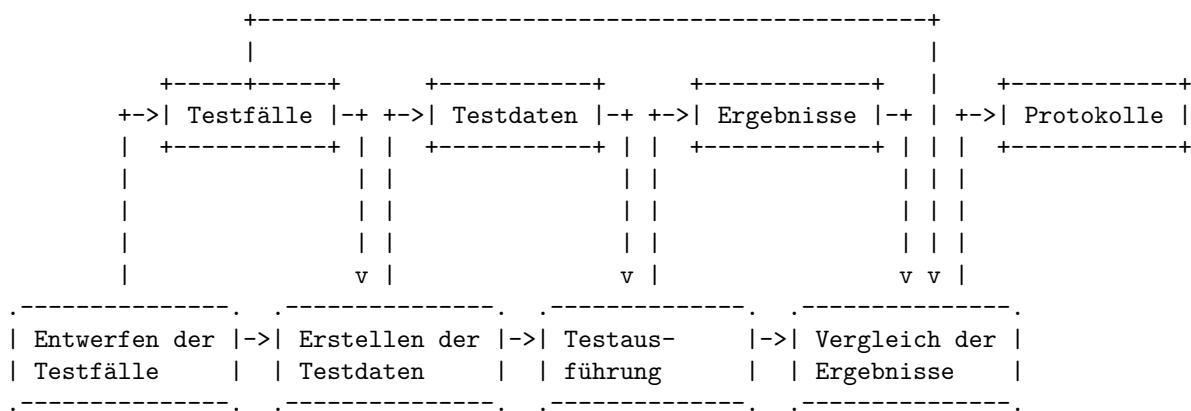
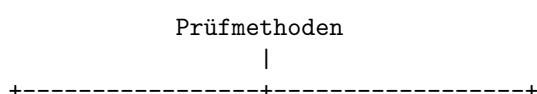


Abbildung motiviert durch <sup>3</sup>

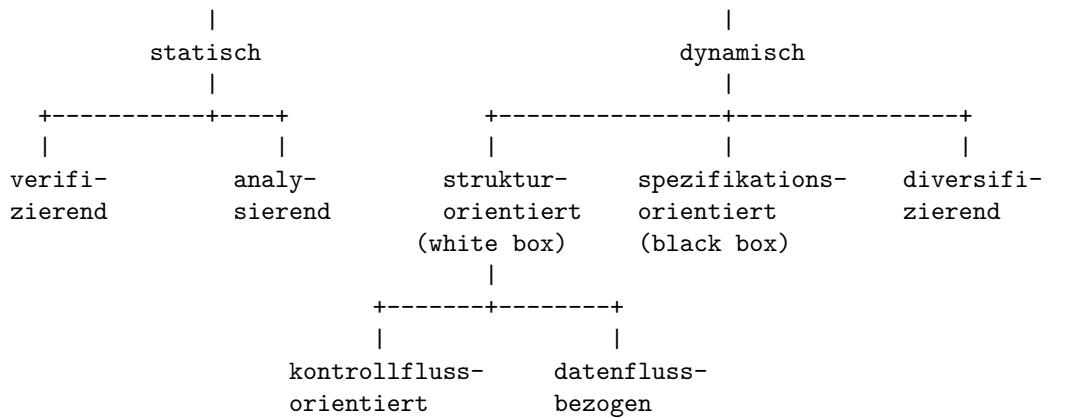
## Klassifikation Testmethoden



<sup>1</sup>Ernst Denert: Software-Engineering. Methodische Projektabwicklung. Springer, Berlin u. a. 1991, ISBN 3-540-53404-0.

<sup>2</sup>Martin Pol, Tim Koomen, Andreas Spillner: Management und Optimierung des Testprozesses. Ein praktischer Leitfaden für erfolgreiches Testen von Software mit TPI und TMap. 2., aktualisierte Auflage. dpunkt.Verlag, Heidelberg 2002, ISBN 3-89864-156-2.

<sup>3</sup>Ian Sommerville: Software Engineering, Pearson Education, 6. Auflage, 2001

Abbildung motivierte aus <sup>4</sup>

### Statische Code Analysen

Statische Code-Analyse ist ein statisches Software-Testverfahren, das zur Übersetzungszeit durchgeführt wird. Der Quelltext wird hierbei einer Reihe formaler Prüfungen unterzogen, bei denen bestimmte Sorten von Fehlern entdeckt werden können, noch bevor die entsprechende Software (z. B. im Modultest) ausgeführt wird. Die Methodik gehört zu den falsifizierenden Verfahren, d. h., es wird die Anwesenheit von Fehlern bestimmt.

- **Codeanalyse** ... In Anlehnung an das klassische Programm Lint wird der Vorgang der Analyse eines Codefragments auch als linten (englisch linting) bezeichnet.

Das folgende Beispiel zeigt die Ausgabe des Tools SonarLinter angewendet auf die initiale Implementierung einer Konsolenanwendung unter Visual Studio 2017. Welche Fehler können Sie ausmachen?

Eine Übersicht zu den Standard-Regeln findet sich unter [Link](#).

- Codereviews ... Reviews sind manuelle Überprüfungen der Arbeitsergebnisse der Softwareentwicklung. Jedes Arbeitsergebnis kann einer Durchsicht durch eine andere Person unterzogen werden.  
Der untersuchte Gegenstand eines Reviews kann verschieden sein. Es wird vor allem zwischen einem Code-Review (Quelltext) und einem Architektur-Review (Softwarearchitektur, insbesondere Design-Dokumente) unterschieden.

### Dynamische Code Analysen

Dynamische Software-Testverfahren sind bestimmte Prüfmethoden, um mit Softwaretests Fehler in der Software aufzudecken. Besonders sollen Programmfehler erkannt werden, die in Abhängigkeit von dynamischen Laufzeitparametern auftreten, wie variierende Eingabeparameter, Laufzeitumgebung oder Nutzer-Interaktion. Wesentliche Aufgabe der einzelnen Verfahren ist die Bestimmung geeigneter Testfälle für den Test der Software.

- strukturorientiert ... Strukturorientierte Verfahren bestimmen Testfälle auf Basis des Softwarequellcodes (Whiteboxtest). Dabei steht entweder die enthaltenen Daten oder aber die Kontrollstruktur, die die Verarbeitung der Daten steuert, im Fokus.
- spezifikationsorientiert ... die sogenannten Black-Box Verfahren werden zum Abgleich des vorgegebenen, spezifizierten und des realen Verhaltens einer Methode genutzt. Beim Modultest wird z. B. gegen die Modulspezifikation getestet, beim Schnittstellentest gegen die Schnittstellenspezifikation und beim Abnahmetest gegen die fachlichen Anforderungen, wie sie etwa in einem Pflichtenheft niedergelegt sind.
- diversifizierend ... Diese Tests analysieren die Ergebnisse verschiedener Versionen einer Software gegeneinander. Es findet entsprechend kein Vergleich zwischen den Testergebnissen und der Spezifikation statt! Zudem kann im Gegensatz zu den funktions- und strukturorientierten Testmethoden kein Vollständigkeitskriterium definiert werden. Die notwendigen Testdaten werden mittels einer der anderen Techniken, per Zufall oder Aufzeichnung einer Benutzer-Session erstellt.

## Planung von Tests

Im folgenden sollen unterschiedliche Black- und White-Box-Tests angewandt werden um eine Klasse MyMath-Functions, die zwei Methoden implementiert, zu testen:

<sup>4</sup>Peter Liggesmeyer, "Software-Qualität - Testen, Analysieren und Verifizieren von Software", Springer, 2002

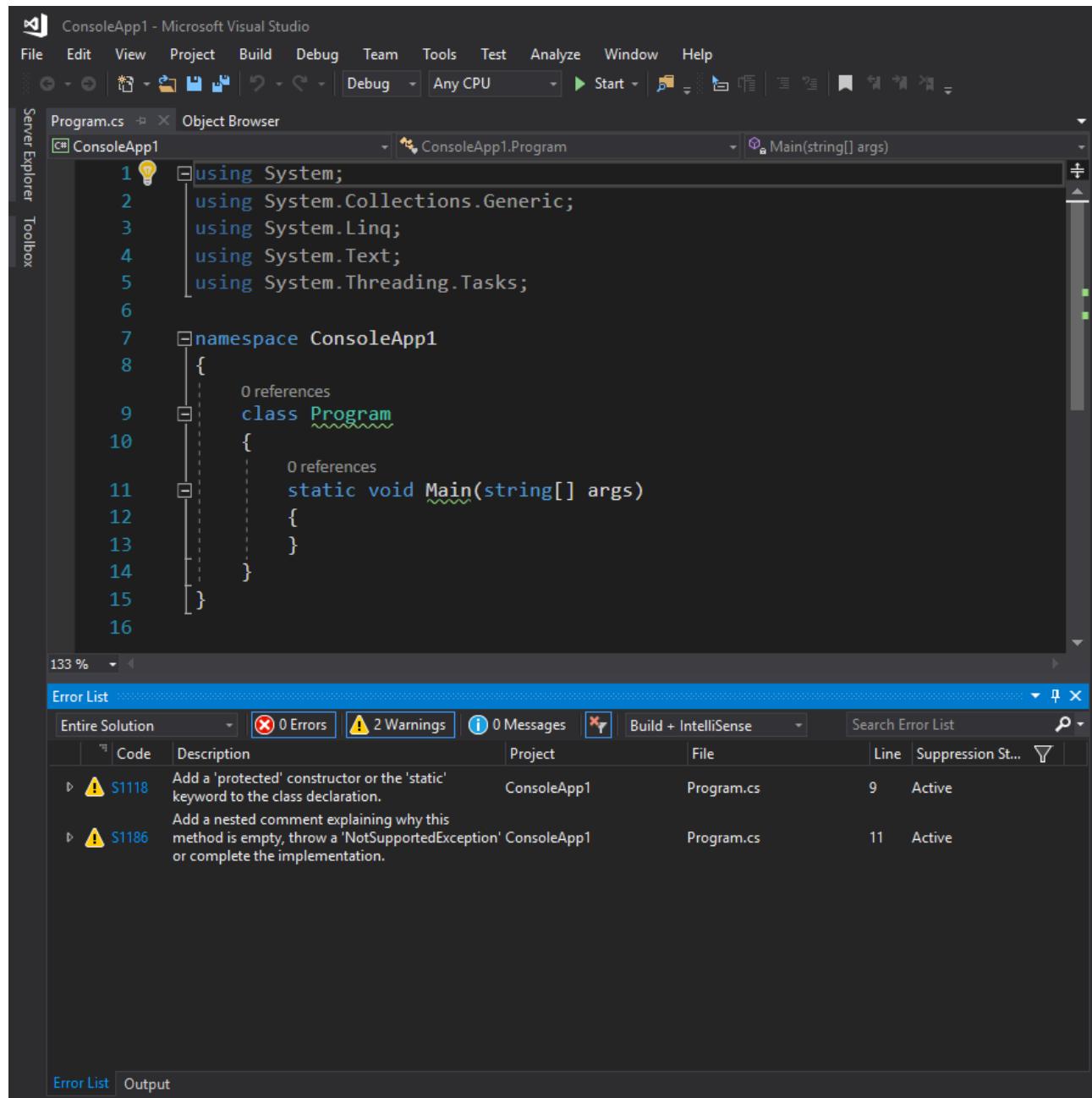


Figure 17.1: instruction-set

```
static class MyMathFunctions{
    //Fakultät der Zahl i
    public int fak(int i) {...}
    // Größter gemeinsamer Teiler von i, j und k
    public int ggt(int i, int j, int k) {...}
}
```

Ein vollständiges Testen aller `int` Werte ( $2^{31}$  bis  $2^{31} - 1$ ) bedeutet für die Funktion `fak()`  $2^{32}$  und für `ggt()`  $2^{32} \cdot 2^{32} \cdot 2^{32}$  Kombinationen. Testen aller möglichen Eingaben ist damit nicht möglich. Für Variablen mit unbestimmtem Wertebereich (`string`) lässt sich nicht einmal die Menge der möglichen Kombinationen darstellen.

## Black-Box-Testing / Spezifikationsorientiert

Black-Box-Testing ... Grundlage der Testfallentwicklung ist die Spezifikation des Moduls. Die Interna des Softwareelements sind nicht bekannt.

Die Güte der Testfälle ist definiert über die Abdeckung möglicher Kombinationen der Eingangsparameter.

Für Black-Box-Testing existieren unterschiedliche Ausprägungen:

- Äquivalenzklassenanalyse
- Grenzwertanalyse [Link](#)
- Ursache Wirkungsgraphen
- Zustandsbasierte Testmethoden

Problematisch ist dabei, dass spezifische Lösungen, wie zum Beispiel in folgendem Fall. Der Entwickler hat hier beschlossen die Performance der Berechnung der Fakultät zu steigern, um die Performance des Algorithmus für Werte kleiner 5 zu verbessern (hypothetisches Beispiel!).

```
static class MyMathFunctions{
    public int fak (int i){
        if ( i==1 ) return 0;
        elseif ( i == 2) return 1;
        elseif ... Ergebnisse für 3 und 4 ...
        elseif (i == 5) return 120;
        else return i * fak(i-1);
    }
}
```

Mit den alleinigen Testfällen `fak(5)==120`, `fak(6)==720` und `fak(10)==3628800` bleiben mögliche Fehler für `fak(1)` und `fak(2)` verborgen.

## White-Box-Testing / Strukturorientiert

White-Box-Testing ... beim „quelltextbasierten Testen“ sind die Interna des getesteten Softwareelements bekannt und werden zur Bestimmung der Testfälle verwendet

White-Box-Testing-Verfahren zerlegen das Programm (statisch oder dynamisch) entsprechend dem Kontrollfluss. Die Güte der Testfälle wird danach beurteilt, wie groß der Anteil der abgedeckten Programmpfade ist. Die Bewertung kann dabei anhand differenzierter Metriken erfolgen:

- Zeilenabdeckung
- Anweisungsüberdeckung
- Zweigüberdeckung
- Pfadüberdeckung
- ...

### C\_0 Anweisungsüberdeckung

Anweisungsüberdeckung (auch  $C_0$ -Test genannt) zerlegt das Programm statisch in seine Anweisungen und bestimmt den Anteil der in den Testfällen berücksichtigten Anweisungen. Üblich ist eine Prüfung von 95%-100% aller Anweisungen durch als  $C - 0$ -Kriterium anzustreben:

$$C_0 = \frac{\text{AnzahlberdeckteAnzahl}}{\text{Gesamtanzahl der Anweisungen}}$$

```

static class MyMathFunctions{
    public int fak (int i){                      // Anweisung
        if ( i==1 ) return 0;                      // 1
        elseif ( i == 2) return 1;                  // 2
        elseif ... Ergebnisse für 3 und 4 ...   // 3 - 4
        elseif ( i == 5) return 120;                // 5
        else return i * fak(i-1);                  // 6
    }
}

```

Der oben genannten Black-Box-Test  $i = \{5, 6, 10\}$  adressierte lediglich 2 der Anweisungen und generiert damit ein  $C_0 = \frac{2}{6} = 0.33$ . Mit dem Wissen um die Codestruktur, kann der White-Box-Test sehr schnell den Nachweis erbringen, dass das gezeigte Black-Box-Vorgehen nur unzureichend die Qualität des Codes abprüft.

```

static class MyMathFunctions{
    public int fak (int i){                      // Anweisung
        int [] facArray = new int [10];           // 1
        facArray[0] = 1;                          // 2
        facArray[1] = 1;
        ...
        facArray[9] = 1;                         // 9
        // besser:
        // int [] facArray = new int[] { 1, 3, 5, 7, 9 };
        if ( i<10 ) return facArray[i];          // 10 + 11
        else return i * fak(i-1);                // 12
    }
}

```

Mit dem Testfall  $i = 1$  lassen sich hingegen vermeindlich  $11/12 = 0.91$  der Anweisungen abdecken, die Fehleinschätzung ist aber offensichtlich. Gleichwohl sind die fest hinterlegten Werte aus Erfahrung heraus auch besonders anfällig für Copy-&-Paste-Fehler.

### C\_1 Zweigüberdeckungstest

Der Zweigüberdeckungstest umfasst den Anweisungsüberdeckungstest vollständig. Für den C1-Test müssen strengere Kriterien erfüllt werden als beim Anweisungsüberdeckungstest. Im Bereich des kontrollflussorientierten Testens wird der Zweigüberdeckungstest als Minimalkriterium angewendet. Mit Hilfe des Zweigüberdeckungstests lassen sich nicht ausführbare Programmzweige aufspüren. Anhand dessen kann man dann Softwareteile, die oft durchlaufen werden, gezielt optimieren.

Die [Zyklomatische Komplexität](#) gibt an, wie viele Testfälle höchstens nötig sind, um eine Zweigüberdeckung zu erreichen.

$$C_1 = \frac{\text{AnzahlberdecktenZweige}}{\text{Gesamtanzahl der Zweige}}$$

```

static class MyMathFunctions{
    public int fak (int i){                      // Verzweigungen
        int [] facArray = new int [10];           //
        facArray[0] = 1;                          //
        facArray[1] = 1;
        ...
        facArray[9] = 1;                         //
        // besser:
        // int [] facArray = new int[] { 1, 3, 5, 7, 9 };
        if ( i<10 ) return facArray[i];          // 1. Verzweigung
        else return i * fak(i-1);                //
    }
}

```

Mit dem Testfall  $i = 1$  ergibt sich eine  $C_1$ -Abdeckung von 0.5.

### C\_2 Pfadüberdeckung

Das  $C_1$  Kriterium berücksichtigt keine Schleifen im zu untersuchenden Code. Der ‘Pfad’ beschreibt gegenüber dem ‘Zweig’ aber eben auch die mehrfache Ausführung ein und des selben Zweiges. Diese Untersuchung muss

entsprechend Schleifen in variabler Durchlaufzahl umsetzten.

### C\_3 Bedingungsüberdeckungstest

$C_3$  Tests extrahieren die Bedingungen die zum Eintritt in die Schleifen führen und generieren Testfälle, die alle Kombinationen abdecken.

```
static class MyMathFunctions{
    public int fak (int i){                                // Verzweigungen
        boolean a, b;
        if (a || b) { ... }
        else { ... }
    }
}
```

Test	Testfälle im Beispiel
C_3a Einfachbedingungsüberdeckungstest	$2 (a = b = \text{true} \text{ sowie } a = b = \text{false})$
C_3b Mehrfachbedingungsüberdeckungstest	$2^n$
C_3c minimaler Mehrfachbedingungsüberdeckungstest	$\leq 2^n$

## Und jetzt konkret!



Figure 17.2: alt-text

**Zu Erinnerung:** Testen ist der Vergleich eines Ergebnisses mit einem erwarteten Resultat.

### Exkurs: Attribute in C

Im Folgenden werden wir Attribute als Hilfsmittel verwenden. Entsprechend soll an dieser Stelle ein kurzer Einschub die Möglichkeiten dieser Zuordnung von Metainformationen zum C# Code verdeutlichen.

Attribute erlaubt es Zusatzinformationen oder Bedingungen in Code (Assemblies, Typen, Methoden, Eigenschaften usw.) einzubinden. Nach dem Zuordnen eines Attributs zu einer Programmheit kann das Attribut zur Laufzeit mit einer Technik namens Reflektion abgefragt werden.

In C# sind Attribute Klassen, die von der Attribute-Basisklasse erben. Alle Klassen, die von Attribute erben, können als eine Art von „Tag“ für andere Codeelemente verwendet werden. Beispielsweise gibt es das Attribut ObsoleteAttribute. Mit diesem Attribut wird gekennzeichnet, dass der Code veraltet ist und nicht mehr verwendet werden sollte.

Beispiele für Standardattribute sind:

Name	Bedeutung
[Obsolete], [Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")] [Conditional("Test")]	Wenn die Zeichenfolge nicht einer #define-Anweisung entspricht, werden alle Aufrufe dieser Methode (aber nicht die Methode selbst) durch den C#-Compiler entfernt.

Attribute werden in rechteckigen Klammern den jeweiligen Codeelementen vorangestellt. Es können mehrere davon kombiniert werden.

```
#define CONDITION1
#define CONDITION2
using System;
using System.Diagnostics;

class Test
{
    static void Main()
    {
        Console.WriteLine("Standard Code ");
        Method0(0);
        Console.WriteLine("Calling Method1");
        Method1(3);
        Console.WriteLine("Calling Method2");
        Method2();
    }

    public static void Method0(int x)
    {
        Console.WriteLine("Here we run actual algorithm.");
    }

    [Conditional("CONDITION1")]
    public static void Method1(int x)
    {
        Console.WriteLine("CONDITION1 is defined");
    }

    [Conditional("CONDITION1"), Conditional("CONDITION2")]
    public static void Method2()
    {
        Console.WriteLine("CONDITION1 or CONDITION2 is defined");
    }
}
```

Die Festlegung der Kompilierungsvorgänge anhand von Inhalten der eigentlichen Code Dateien scheint “unglücklich”. Es bietet sich natürlich an, die zugehörigen Konfigurationen in unsere Projektdateien auszulagern.

```
using System;
using System.Diagnostics;

class Test
{
    static void Main()
    {
        Console.WriteLine("Standard Code ");
        Method0(0);
        Console.WriteLine("Calling Method1");
```

```

Method1(3);
Console.WriteLine("Calling Method2");
Method2();
}

public static void Method0(int x)
{
    Console.WriteLine("Here we run actual algorithm.");
}

[Conditional("CONDITION1")]
public static void Method1(int x)
{
    Console.WriteLine("CONDITION1 is defined");
}

[Conditional("CONDITION1"), Conditional("CONDITION2")]
public static void Method2()
{
    Console.WriteLine("CONDITION1 or CONDITION2 is defined");
}
}

<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
<OutputType>Exe</OutputType>
<TargetFramework>net5.0</TargetFramework>
<DefineConstants>CONDITION2;</DefineConstants>
</PropertyGroup>
</Project>

```

## Idee 1: Eigenen Testmethoden

```

using System;

// Zu testende Klasse
public class Calculator
{
    public static int DivideTwoValues(double x, double y, ref double result){
        if (y != 0){
            result = x / y;
            return 0;
        }
        else return -1;
    }
}

// Testklasse
public class TestCalculator{
    public static void Test_DivideMethod(){
        double result = 0;
        int state = Calculator.DivideTwoValues(3,4, ref result);
        if ((state == 0) & (result == 0.75))
        {
            Console.WriteLine("Test bestanden !");
        }
        else{
            Console.WriteLine("Test fehlgeschlagen");
        }
    }
}

```

```
// Anwendungsprogramm
public class Program
{
    public static void Main(string[] args)
    {
        //double result = 0;
        //int state = Calculator.DivideTwoValues(3,4, ref result);
        //Console.WriteLine($"Das Ergebnis lautet {result}, der State {state}.");
        TestCalculator.Test_DivideMethod();
    }
}
```

Welche Funktionalität fehlt Ihnen in diesem Setup? Welche weitergehenden Features würden Sie für unsere Testmethoden vorschlagen.

## Idee 2: Test-Frameworks

```
[TestClass]    // <-- Framework spezifisch
public class CalculatorTests
{
    [TestMethod]    // <-- Framework spezifisch
    public void TestMethod1()
    {
        // Arrange
        double result;
        double x = 3, y = 4;
        int state;
        double expected = 0.75;

        // Act
        int state = Calculator.DivideTwoValues(x, y, ref result);

        // Assert
        Assert.AreEqual(result, expected);
        //           ^---- Framework spezifisch
    }
}
```

Vorteile:

- Leistungsfähige API (automatisierte Tests, variable Input-Parameter, Berücksichtigung von Exceptions)
- “Standardisiertes” Nutzungskonzept
- Integration in die Entwicklungsumgebungen

Nachteil:

- verschiedene Interpretationen und Performance der Frameworks

Die wichtigsten Tools unter C# sind [xUnit](#), [nunit](#), [MSTest](#). Einen guten Überblick zum Vergleich der Schlüsselworte liefert [Link](#)

Hierzu nutzen wir das xunit Framework. Eine Folge von Tests für unsere DivideTwoValues() Methode könnte dann wie folgt aussehen.

```
using Xunit;

namespace MyMathMethods.Test
{
    public class Test_DivideTwoValues
    {
        [Fact]
        public void Check_StateEqualPositiveInputs()
        {
            // Arrange
```

```

        double result = 0;
        double dividend = 5;
        double divisor = dividend;
        int expected = 0;

        // Act
        var state = Calculator.DivideTwoValues(dividend, divisor, ref result);

        // Assert
        Assert.Equal(expected, state);
    }

    [Fact]
    public void Check_StateZeroAsDivended()
    {
        // Arrange
        double result = 0;
        double dividend = 5;
        double divisor = 0;
        int expected = -1;

        // Act
        var state = Calculator.DivideTwoValues(dividend, divisor, ref result);

        // Assert
        Assert.True(expected == state);
    }

    [Theory]                                     // Übergabe von variablen Parametersets
    [InlineData(10, 2, 5)]
    [InlineData(5, 2, 2.5)]
    [InlineData(double.MaxValue, double.MaxValue, 1)] // Edge Cases
    [InlineData(double.MaxValue, 1, double.MaxValue)]
    public void Check_ResultCalculation(double dividend, double divisor, double expected)
    {
        // Arrange
        double result = 0;

        // Act
        var state = Calculator.DivideTwoValues(dividend, divisor, ref result);

        // Assert
        Assert.Equal(expected, result);
    }
}
}

```

Wie setzen wir das Ganze um?

```

dotnet new sln -o unit-testing-example
cd unit-testing-example
dotnet new classlib -o CalcService           // Code der Divisionsoperation einfügen
mv CalcService/Class1.cs CalcService/Division.cs
dotnet sln add ./CalcService/CalcService.csproj
dotnet new xunit -o CalcService.Tests        // obigen Testcode einfügen
dotnet add ./CalcService.Tests/CalcService.Tests.csproj reference ./CalcService/CalcService.csproj
dotnet sln add ./CalcService.Tests/CalcService.Tests.csproj

```

Damit entsteht eine **solution**, die zwei **project** umfasst - die eigentliche Anwendung als **classlib** und die Testfälle.

```
.
├── CalcService
```

```

    └── CalcService.csproj
    └── Division.cs
    └── CalcService.Tests
        └── CalcService.Tests.csproj
        └── UnitTest1.cs
    └── unit-testing-example.sln

```

Das Ausführen der Tests ist nun mit `dotnet test` möglich.

Eine automatische Generierung von Test Merkmalen ist mit Hilfe zusätzlicher Tools, die in dotnet integriert sind möglich.

```

dotnet add package coverlet.collector
dotnet tool install --global dotnet-reportgenerator-globaltool
dotnet tool install --global coverlet.console

dotnet test --collect:"XPlat Code Coverage" --results-directory:"./.coverage"
reportgenerator "-reports:.coverage/**/*.cobertura.xml" "-targetdir:.coverage-report/" "-reporttypes:HT

```

Das Argument “XPlat Code Coverage” bezieht sich auf das Zwischenformat der Darstellung. Das `./.coverage` dient zur Angabe des Verzeichnisses, in dem die Ergebnisse gespeichert werden sollen. Wenn keines angegeben wird, wird standardmäßig ein `TestResults`-Verzeichnis innerhalb jedes Projekts verwendet. `reportgenerator` erzeugt dann die entsprechende html-Repräsentation.

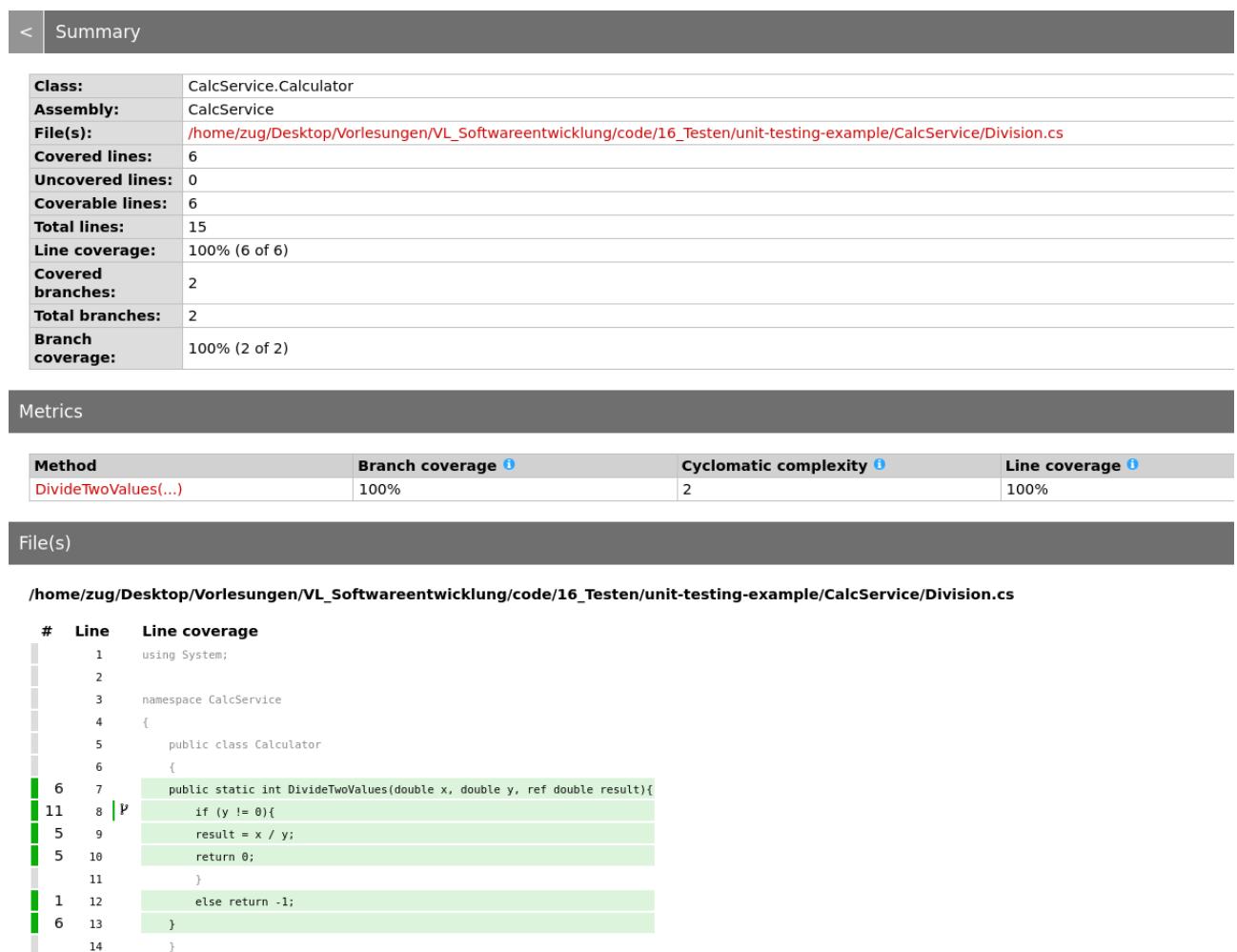


Figure 17.3: instruction-set

# Chapter 18

## Dokumentation und Build-Tools

Parameter	Kursinformationen
<b>Veranstaltung</b>	Vorlesung Softwareentwicklung
<b>Semester</b>	Sommersemester 2021
<b>Hochschule:</b>	Technische Universität Freiberg
<b>Inhalte:</b>	Realisierung von Dokumentationen, Anwendung von Build-Tools
<b>Link auf den GitHub:</b>	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/17_Dokumentation_BuildTools.md">https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/17_Dokumentation_BuildTools.md</a>
<b>Autoren</b>	@author

## Dokumentation

### Wer braucht schon eine Doku?

*Eine Softwaredokumentation ist mangelhaft, wenn in ihr in nennenswertem Umfang Bildschirmdialoge nicht (mehr) aktuell sind, nicht mit den im Programm vorhandenen Dialogen übereinstimmen oder gar nicht dokumentiert sind. ... Eine Softwaredokumentation ist mangelhaft, wenn sie den Anwender nicht in die Lage versetzt, die Software im Bedarfsfalle erneut oder auf einer anderen Anlage zu installieren. [LG Bonn, 19.12.2003]*

Als Softwaredokumentation bezeichnet man die Beschreibung einer Software für Entwickler, Anwender und Benutzer. Entsprechend den unterschiedlichen Rollen, wird erläutert, wie die Software funktioniert, was sie erzeugt und verarbeitet (z. B. Daten), wie sie zu benutzen ist, was zu ihrem Betrieb erforderlich ist und auf welchen Grundlagen sie entwickelt wurde.

*Klassifikation 1 - Intern/Extern* ... bezieht sich dabei auf die Frage, ob das Ganze für den internen Gebrauch oder den externen Gebrauch, also zur Weitergabe an Kunden, realisiert werden muss. Letztgenannte Variante unterliegt einer Vielzahl von rechtlichen Normierungen und Standards.

### *Klassifikation 2 - Inhalt*

Art der Dokumentation	Installationsdokumentation: erforderlichen Hardware und Software, mögliche Betriebssysteme und -Versionen, vorausgesetzte Software-Umgebung wie etwa Standardbibliotheken und Laufzeitsysteme. Erläuterung der Prozeduren zur Installation, außerdem zur Pflege (Updates) und De-Installation, bei kleinen Produkten eine Readme-Datei.
Benutzerdokumentation	Informationsmaterial für die tatsächlichen Endbenutzer, etwa über die Benutzerschnittstelle. Den Anwendern kann auch die Methodendokumentation zugänglich gemacht werden, um Hintergrundinformationen und ein allgemeines Verständnis für die Funktionen der Software zu vermitteln.

---

Art

der

Dokumentation

---

Datendokumentation: Beschreibungen zu den Daten erforderlich. Es sind die Interpretation der

Informationen in der realen Welt, Formate, Datentypen, Beschränkungen (Wertebereich, Größe) zu benennen. Die Datendokumentation kann oft in zwei Bereiche aufgeteilt werden: Innere Datenstrukturen, wie sie nur für Programmierer sichtbar sind und Äußere Datendokumentation für solche Datenelemente, die für Anwender sichtbar sind – von Endbenutzern einzugebende und von der Software ausgegebene Informationen. Dazu gehört auch die detaillierte Beschreibung möglicher Import-/Exportschnittstellen.

Testdokumentation: Testfällen, mit denen die ordnungsgemäße Funktion jeder Version des Produkts getestet werden können, sowie Verfahren und Szenarien, mit denen in der Vergangenheit erfolgreich die Richtigkeit überprüft wurde.

Entwicklungsdocumentation: Versionen auf Grund von Veränderungen, der jeweils zugrundegelegten Ziele und Anforderungen und der als Vorgaben benutzten Konzepte (z. B. in Lastenheften und Pflichtenheften); beteiligte Personen und Organisationseinheiten; erfolgreiche und erfolglose Entwicklungsrichtungen; Planungs- und Entscheidungsunterlagen etc.

---

Häufig fasst ein Projekt alle Arten der Dokumentation gleichermaßen zusammen. Im folgenden soll zum Beispiel die Implementierung der avrlibc für Mikrocontroller der AtTiny, AtMega und XMega Familie auf die entsprechenden Beiträge hin untersucht werden.

<https://www.nongnu.org/avr-libc/>

### Klassifikation 3 - Autoren

Entwickler:

- empfindet die Softwaredokumentation oft als lästiges Übel
- generiert ggf. sehr spezifische Dokumentationen ohne Anspruch auf Allgemeinverständlichkeit
- ist aber der unmittelbare Experte!

Technischer Redakteur:

- fehlendes technisches Detailwissen, dichter am Wissensstand des Kunden
- geeignetes Abstraktionsvermögen
- erfahren im Dokumentenmanagement

## Programmiererdokumentation

“Code is like humor. When you have to explain it, it’s bad.”

“Warum soll ich dokumentieren, es ist doch mein Code!”

“Bei einem gut geschriebenen und formatierten Code braucht man weniger zu dokumentieren.”

Denken Sie in Zielgruppen, wenn Sie die Dokumentation erstellen. Welche Hilfestellung erwartet welcher Nutzer der Implementierung? Welche Voraussetzungen können Sie annehmen?

Entsprechend differenzieren wir Zielgruppen und fragen uns welche Personenkreise wir davon bedienen wollen. Aus dieser Fragestellung ergeben sich die Schwerpunkte der Dokumentationsarbeit:

- Praktiker ... starker Bezug zur Umsetzung, benötigt Code-Kommentare, Code-Beispiele
- Systematiker ... liest zuerst einmal die Grundlagen, bemüht sich alle Hintergrundinfo zu API/Framework, zu erfassen. benötigt Architekturbeschreibung, Konzepte allgemeiner Programmieraufgaben (Error handling, Lokalisierung &Co.)
- Bedarfsleser ... situationsgetriebene Auswertung der Dokumentation, erwartet Antworten auf spezifische Fragen, benötigt Code-Kommentare, Hintergrundinformationen und Code-Beispiele

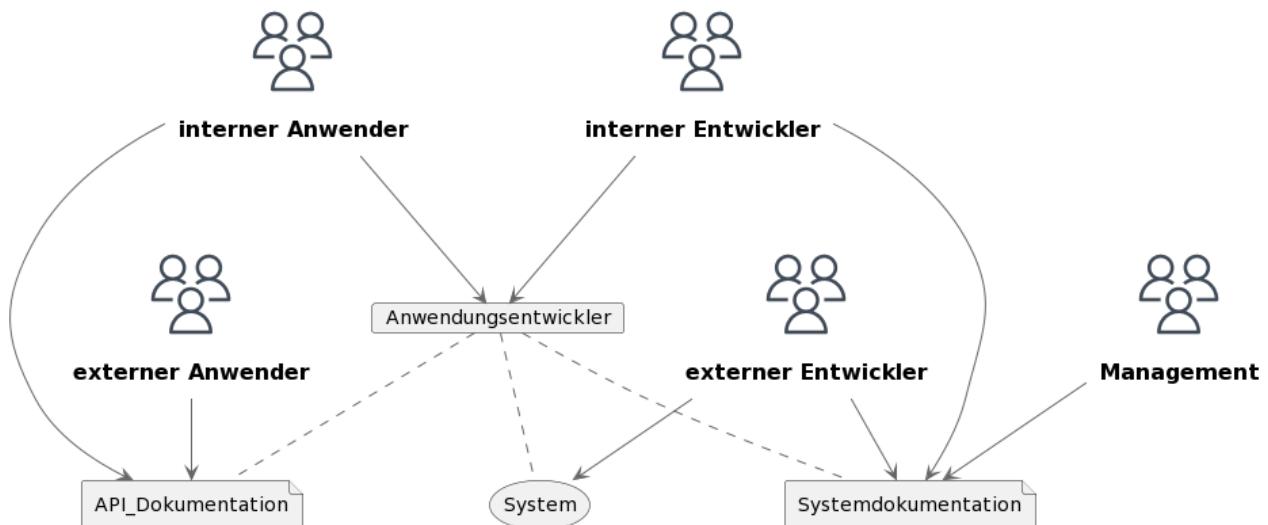


Abbildung motiviert aus <sup>1</sup>

Entsprechend ergeben sich vielfältige Dokumentationstypen, die ggf. erfasst werden sollten:

- Programmier Kochbuch
- Wiki
- Code Kommentare
- API Dokumentation

die in unterschiedlichen Formaten (online, offline, html, pdf, doc, usw.) realisiert werden können.

## Benutzerdokumentation

Auch die Benutzerdokumentation muss einer starken Zielgruppenorientierung unterliegen sowie unterschiedliche Konzepte der Handhabung einer Software beschreiben. Für Handbücher lassen sich zum Beispiel folgende Typen unterscheiden:

- Trainings-Handbuch
- Referenz-Handbuch
- Referenzkarte (auch als Cheat-Sheets bezeichnet)
- Benutzer-Leitfaden

vgl. zum Beispiel Python Pandas [Cheat Sheet](#)

## Realisierung der Dokumentation in Csharp

Merke: Anhand einer Semantik werden aus formlosen Kommentaren automatisch auswertbare Elemente einer Benutzerdokumentation!

Gliederungselemente für die Dokumentationsgenerierung sind dabei:

- Zuordnungen von Informationen zu Klassen, Methoden, Variablen
- Erläuterung von Methodensignaturen (Input/Outputs)
- Beschreibung der Funktion von Variablen, Properties usw.
- Integration von Beispielcode

Unter C# wird hinsichtlich der Darstellung zusätzlich zwischen Kommentaren mit // oder /\* \*/ und Dokumentationsinhalten unterschieden, die mit /// eingeleitet werden.

`using System;`

```
// Eine zweielementige Vektorklasse ohne Methoden
public class Vector {
    public double X;
    public double Y;
    public Vector (double x, double y){
```

<sup>1</sup>Uwe Friedrichsen, Optimale Systemdokumentation mit agilen Prinzipien, 06/11, <https://www.codecentric.de/publikation/optimale-systemdokumentation-mit-agilen-prinzipien/>

```

    this.X = x;
    this.Y = y;
}
public static bool operator !=(Vector p1, Vector p2){
    // hier hatte ich keine Lust mehr
    // TODO die Methode müsste noch implementiert werden.
    return true;
}
}

/// <summary>
/// Klasse mit dem Einsprungspunkt für die Main zu Testzwecken.
/// </summary>
public class Program
{
    /// <summary>
    /// Main Funktion mit exemplarischer Initialisierung zweier Vektoren.
    /// </summary>
    public static void Main(string[] args)
    {
        Vector a = new Vector (3,4);
        Vector b = new Vector (9,6);
        Console.WriteLine (a == b);
    }
}

```

Über entsprechende Tags lassen sich den Dokumentationsfragmenten Bedeutungen geben, die eine entsprechende Gliederung und Zuordnung erlaubt:

Tag	Erläuterung
<summary>	Umfasst kurze Informationen über einen Typ oder Member.
<remarks>	Ergänzt weiterführende Informationen zu Typen und Membern.
<returns>	Beschreibt den Rückgabewert einer Methode
<value>	Beschreibt Bedeutung einer Eigenschaft
<example>	Ermöglicht mit <code> die Einbettung von (Code-)Beispielen.
<para>	Ermöglicht die Beschreibung der Eingabeparameter einer Methode
<c>	Indikator für Inline-Codefragmente
<exception>	Erlaubt die Beschreibung möglicher Exceptions, die in einer Methode auftreten können.
<see>	Klickbare Links in Verbindung mit <cref>

Was lässt sich damit umsetzen?

```

// Divides an integer by another and returns the result
// Codebeispiel aus der Microsoft Dokumentation
// siehe https://docs.microsoft.com/de-de/dotnet/csharp/codedoc

    /// <summary>
    /// Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the r
    /// </summary>
    /// <returns>
    /// The quotient of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Divide(4, 5);
    /// if (c > 1)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>

```

```

    /// </example>
    /// <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.</exception>
    /// See <see cref="Math.Divide(double, double)"> to divide doubles.
    /// <seealso cref="Math.Add(int, int)">
    /// <seealso cref="Math.Subtract(int, int)">
    /// <seealso cref="Math.Multiply(int, int)">
    /// <param name="a">An integer dividend.</param>
    /// <param name="b">An integer divisor.</param>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

```

Offensichtlich bläht diese Struktur den Code, der im Beispiel aus insgesamt 4 Zeilen besteht unschön auf. Die Lesbarkeit, die ja eigentlich gesteigert werden sollte leidet darunter. Welche Lösungsmöglichkeit sehen Sie?

### Separate Dokumentationsdateien

Mit dem <include>-Tag lassen sich externe Dokumentationen während des Generierungsprozesses referenzieren. Damit umfasst die Dokumentation lediglich einen entsprechenden Link in Kombination mit einem einfachen Kommentar.

```

<docs>
    <members name="math">
        <Math>
            <summary>
                The main <c>Math</c> class.
                Contains all methods for performing basic math functions.
            </summary>
            <remarks>
                <para>This class can add, subtract, multiply and divide.</para>
                <para>These operations can be performed on both integers and doubles.</para>
            </remarks>
        </Math>
        <AddInt>
            <summary>
                Adds two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
            </summary>
            <returns>
                The sum of two integers.
            </returns>
        </AddInt>
        <DivideInt>
            <summary>
                Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the quotient.
            </summary>
            <returns>
                The quotient of two integers.
            </returns>
        </DivideInt>
    </members>
</docs>

// Adds two integers and returns the result
/// <include file='docs.xml' path='docs/members[@name="math"]/AddInt/*'>
public static int Add(int a, int b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}

```

```
}
```

Leider funktioniert dieser Mechanismus unter dem gleich vorzustellen Tool Doxygen nicht.

### Konkrete Umsetzung mit C#

**Anwendung 1:** Generierung separater XML Dateien zur Verwendung in Visual Studio Code oder Visual Studio. Um die entsprechende XML Datei sollte den gleichen Namen tragen wie das Assembly und sich im gleichen Ordner befinden.

```
csc -out:MyAssembly.exe File.cs -doc:MyAssembly.xml
```

Aufbauend auf den Inhalten der XML Datei ist IntelliSense in Visual Studio dann in der Lage, die Textinformationen zu Klassen und Membern bei der Eingabe anzuzeigen.

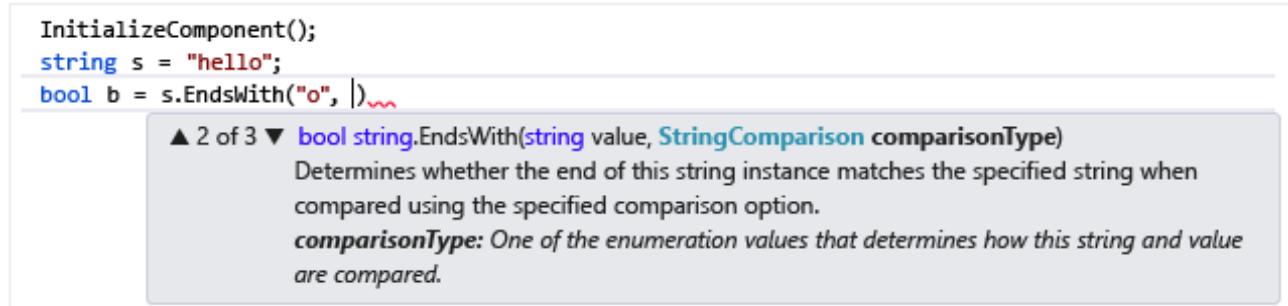


Figure 18.1: IntelliSense

Sie können die Parameterinformation manuell aufrufen, indem Sie STRG+UMSCHALT+LEERTASTE drücken (die alternativen Methoden mit der Maus braucht ohnehin niemand).

**Anwendung 2:** Die XML basierten Dokumentationsinhalte können in html oder pdf Dokumente transformiert werden, um eine losgelöste Dokumentation darzustellen. Hierfür können externe Tools herangezogen werden. Beispiele dafür sind [Javadoc](#), [Sphinx](#) oder [Doxygen](#). Ursprünglich bot Microsoft eine eigene Toolchain für die Code-Generierung an, diese wird gegenwärtig unter dem Projektnamen Sandcastle als Open-Source Projekt weitergeführt.

<https://github.com/EWSSoftware/SHFB>

Im folgenden soll beispielhaft auf die Anwendung von Doxygen eingegangen werden.

[!?doxygenmovie](#)

Die Anwendung von Doxygen wird im folgenden Foliensatz anhand eines Beispielprojektes gezeigt. Dabei werden zwei Verbesserungen der Darstellung realisiert.

1. Auswertung der Doxygen Ausgaben im Hinblick auf die Abdeckung der Dokumentation.
2. Einbindung des Quellcodes über das SOURCE\_BROWSER Flag.

Anpassung des entsprechenden Eintrages von NO auf YES.

```
# If the SOURCE_BROWSER tag is set to YES then a list of source files will be
# generated. Documented entities will be cross-referenced with these sources.
#
# Note: To get rid of all source code in the generated output, make sure that
# also VERBATIM_HEADERS is set to NO.
# The default value is: NO.
```

```
SOURCE_BROWSER      = NO
```

3. Integration des Projektfiles README.md in die Dokumentation.

```
INPUT              += README.md
USE_MDFILE_AS_MAINPAGE = README.md
```

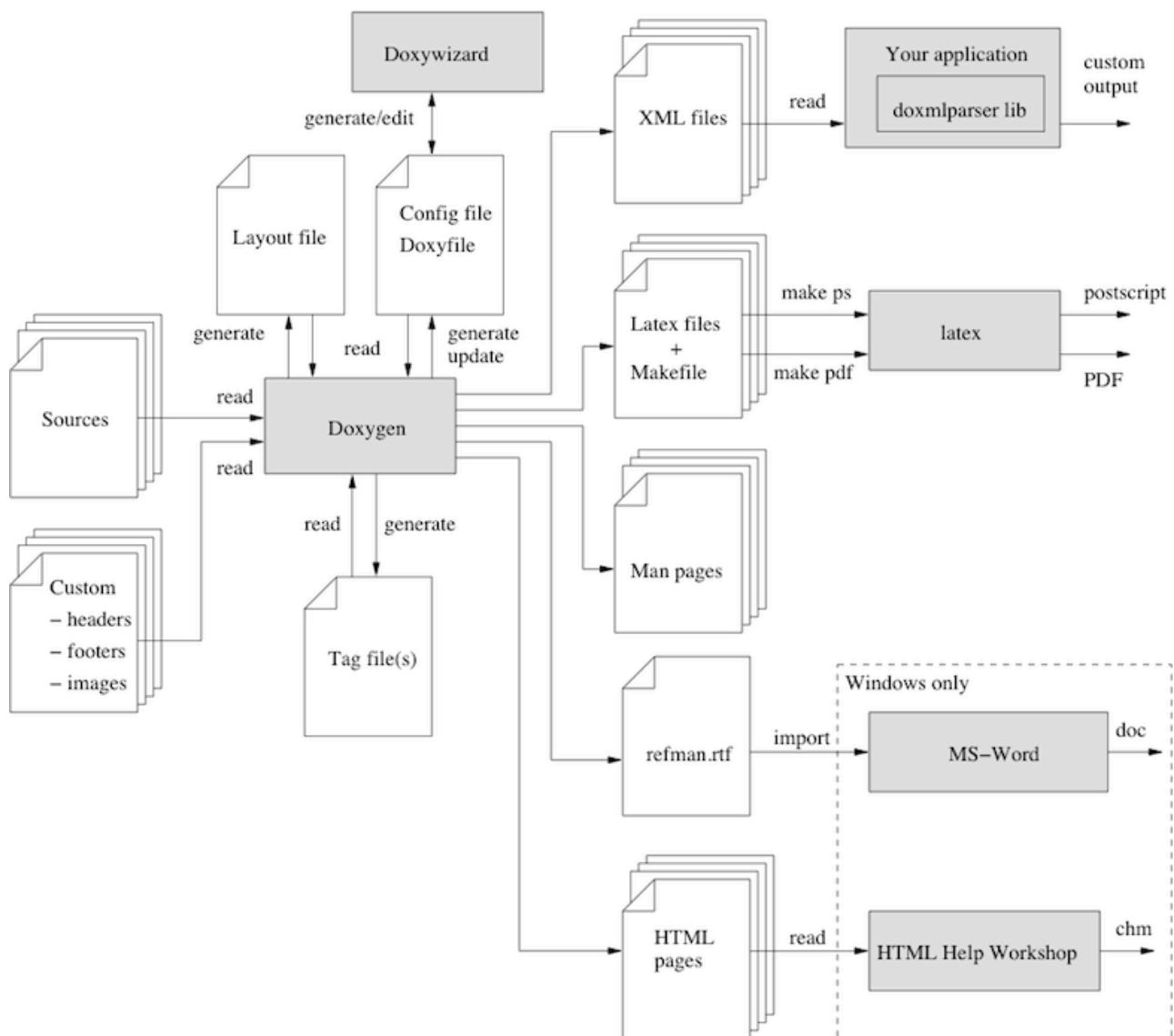


Figure 18.2: doxygen

## Paketmanagement

**Merke:** Erfinde das Rad nicht neu!

Wie schaffen es erfahrene Entwickler innerhalb kürzester Zeit Prototypen mit beeindruckender Funktionalität zu entwerfen? Sicher, die Erfahrung spielt hier eine Große Rolle aber auch die Wiederverwendung von existierendem Code. Häufig wiederkehrende Aufgaben wie zum Beispiel:

- das Logging
- der Zugriff auf Datenquellen
- mathematische Operationen
- Datenkapselung und Abstraktion
- ...

werden bereits durch umfangreiche Bibliotheken implementiert und werden entsprechend nicht neu geschrieben.

Ok, dann ziehe ich mir eben die zugehörigen Repositories in mein Projekt und kann die Bibliotheken nutzen. In individuell genutzten Implementierungen mag das ein gangbarer Weg sein, aber das Wissen um die zugehörigen Abhängigkeiten - Welche Subbibliotheken und welches .NET Framework werden vorausgesetzt? - liegt so nur implizit vor.

Entsprechend brauchen wir ein Tool, mit dem wir die Abhängigkeiten UND den eigentlichen Code kombinieren und einem Projekt hinzufügen können. NuGet löst diese Aufgabe für .NET und schließt auch gleich die Mechanismen zur Freigabe von Code ein. NuGet definiert dabei, wie Pakete für .NET erstellt, gehostet und verarbeitet werden.

Ein NuGet-Paket ist eine gepackte Datei mit der Erweiterung .nupkg die: + den kompilierten Code (DLLs), + ein beschreibendes Manifest, in dem Informationen wie die Versionsnummer des Pakets, ggf. der Speicherort des Source Codes oder die Projektwebseite enthalten sind sowie + die Abhängigkeiten von anderen Paketen und dessen Versionen enthalten sind Ein Entwickler, der seinen Code veröffentlichen möchte generiert die zugehörige Struktur und lädt diese auf einen NuGet Server. Unter dem [Link](#) kann dieser durchsucht werden.

### Anwendungsbeispiel: Symbolisches Lösen von Mathematischen Gleichungen

Eine entsprechende Bibliothek steht unter [Projektwebseite](#). Das Ganze wird als Nuget Paket gehostet [MathNet](#).

Unter der Annahme, dass wir dotnet als Buildtool benutzen ist die Einbindung denkbar einfach.

```
dotnet new console -o SymbolicMath
cd SymbolicMath
dotnet add package MathNet.Symbolics
Determining projects to restore...
Writing /tmp/tmpNsaYtc.tmp
info : Adding PackageReference for package 'MathNet.Symbolics' into project '/home/zug/Desktop/Vorlesun
info :   GET https://api.nuget.org/v3/registration5-gz-semver2/mathnet.symbolics/index.json
...

```

Danach findet sich in unserer Projektdatei .csproj ein entsprechender Eintrag

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="MathNet.Symbolics" Version="0.24.0" />
  </ItemGroup>
</Project>

using System;
using System.Collections.Generic;
using MathNet.Symbolics;
using Expr = MathNet.Symbolics.SymbolicExpression; // Platzhalter für verkürzte Schreibweise

class Program
```

```

{
    static void Main(string[] args)
    {
        Console.WriteLine("Beispiele für die Verwendung des MathNet.Symbolics Paketes");
        var x = Expr.Variable("x");
        var y = Expr.Variable("y");
        var a = Expr.Variable("a");
        var b = Expr.Variable("b");
        var c = Expr.Variable("c");
        var d = Expr.Variable("d");
        Console.WriteLine("a+a =" + (a + a + a).ToString());
        Console.WriteLine("(2 + 1 / x - 1) =" + (2 + 1 / x - 1).ToString());
        Console.WriteLine("((a / b / (c * a)) * (c * d / a) / d) =" + ((a / b / (c * a)) * (c * d / a) / d);
        Console.WriteLine("Der zugehörige Latex Code lautet " + ((a / b / (c * a)) * (c * d / a) / d).ToLaTeX());
    }
}

<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>
<ItemGroup>
    <PackageReference Include="MathNet.Symbolics" Version="0.24.0" />
</ItemGroup>
</Project>

```

## Build Tools

### Konzepte

Wir haben bisher über das Compilieren des Codes und die Realisierung von Tests gesprochen, nun kommt auch noch die Erstellung einer Dokumentation hinzu ... und für all diese Teilaufgaben gibt es jeweils eigene Tools. Das möchte doch niemand manuell angehen!

```
mcs program.cs
mono program.exe
doxygen Doxyfile
...
```

Wie kann man den Codeerstellungsprozess automatisieren und organisieren ohne jedes mal über eine Vielzahl von CLI-Parametern nachdenken zu müssen? Welche Aspekte sollte diese Straffung des Entwicklungsflusses abdecken:

- Auflösung der Abhängigkeiten von anderen Paketen
- Kompilierung
- Anwendung von Qualitätsmetriken
- Programmausführung
- Tests
- Generierung der Dokumentation

Dabei wäre es sinnvoll, wenn ausgehend von einer tatsächlichen Veränderung der Eingabedateien eine Realisierung des gewünschten Targets erfolgt.

Vorgang	Kompilierung	Tests	Qualitätscheck	Doku
Änderung in einer Code Datei	X	X	X	X
Hinzufügen eines Tests		X		
Anpassen einer Dokumentationsdatei				X

Um diese Idee abzubilden müssen wir offenbar Abhängigkeiten beschreiben, die ausgehend von einer Veränderung, eine bestimmte Folge von Aktionen auslösen. Ausgangspunkt für diese Aktionen können unterschiedliche Quellen sein (vgl. Generierung der Dokumentation in obiger Tabelle).

## dotnet

dotnet ist ein Tool für das Verwalten von .NET-Quellcode und Binärdateien. Das Programm stellt Befehle zur Verfügung, die bestimmte Aufgaben erfüllen, die zudem jeweils über eigene Argumente verfügen.

Befehl	Bedeutung
dotnet new	Initialisiert ein C#- oder F#-Projekt für eine bestimmte Vorlage.
dotnet restore	Stellt die Abhängigkeiten für eine bestimmte Anwendung wieder her.
dotnet build	Erstellt eine .NET Core-Anwendung.
dotnet clean	Bereinigen von Buildausgaben.
dotnet test	Ausführen der entsprechenden Testanwendungen

Beispielanwendung: Entwerfen Sie eine C# Programm, dass Excel-Files erstellt, für die bestimmte Felder vorinitialisiert sind.

```
export DOTNET_SKIP_FIRST_TIME_EXPERIENCE=1
dotnet --help
dotnet new console -n MyExcelGenerator
cd MyExcelGenerator
cat MyExcelGenerator.csproj
dotnet add package EPPlus
cat MyExcelGenerator.csproj
... Program.cs anpassen ... siehe Codebeispiel im Codeordner
dotnet build
dotnet run
soffice -calc myworkbook.xlsx
```

dotnet ermöglicht keine Defintion von Abhängigkeiten (ohne auf MSBuild zurückzugreifen) standardisiert aber den Erstellungs- und Testprozess, sowie das Pakethandling!

## MSBuild

MSBuild ist Build-Tool, das insbesondere für das Erstellen von .NET-basierten Anwendungen genutzt wird. Microsofts Visual Studio ist in wesentlichem Maße von MSBuild abhängig; MSBuild selbst ist aber nicht von Visual Studio abhängig. Dadurch lassen sich mit MSBuild auch Visual-Studio-Projekte ohne den Einsatz von Visual Studio bauen.

Im Wesentlichen besteht MSBuild aus der Datei `msbuild.exe` und dll-Dateien, die auch im .NET Framework enthalten sind, und XML-Schemas, nach deren Vorgaben die von msbuild.exe verwendeten Projektdateien aufgebaut sind. Wegen der XML-Basiertheit wird MSBuild auch als Auszeichnungssprache eingeordnet.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="A">
    <Message Text="Hello World!"/>
  </Target>
</Project>
```

Innerhalb der xml-Struktur definieren Sie sogenannte Targets als Einsprungpunkte für den Erstellungsprozess. Im Beispiel sind dies zunächst nur HelloWorld-Ausgaben, im Weiteren wird dies auf konkrete Kompilierungsvorgänge ausgeweitet.

Ein spezifisches Target kann mit `msbuild filename /t:targetname` aufgerufen werden.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="A">
    <Message Text="Hello World! A"/>
  </Target>
  <Target Name="B" DependsOnTargets="A">
    <Message Text="Hello World! B"/>
  </Target>
</Project>
```

Ein minimales Konfigurationsfile für die Build-Prozess einer einzelnen C# Datei könnte folgende Konfiguration haben:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <Compile Include="helloworld.cs" />
  </ItemGroup>
  <Target Name="Build">
    <Csc Sources="@(&Compile)" />
  </Target>
</Project>
```

Ein Beispiel für eigene Experimente findet sich unter ‘code/17\_ToolChain/msbuildProject’

Die zuvor besprochenen dotnet Befehle bauen auf MSBuild auf und kapseln diese. Die Ausführung von `dotnet build` entspricht `dotnet msbuild -restore -target:Build`.

Arbeiten Sie auch die Dokumentation von MSBuild durch, diese stellt auch das umfangreiche Featureset (vordefinierte Targets, integrierte Tools, die Möglichkeit externe Anwendungen einzubetten) vor, das deutlich über die Beispiele hinausgeht.

<https://docs.microsoft.com/de-de/visualstudio/msbuild/msbuild?view=vs-2019>

## Make

`make` wird beispielsweise, um in Projekten, die aus vielen verschiedenen Dateien mit Quellcode bestehen, automatisiert alle Arbeitsschritte (Übersetzung, Linken, Dateien kopieren etc.) zu steuern, bis hin zum fertigen, ausführbaren Programm.

`make` liest ein sogenanntes *Makefile* (ACHTUNG Großschreibung erforderlich) und realisiert den beschriebenen Übersetzungsprozesses entsprechend der Abhängigkeiten.

```
A: B
  generate_A
```

```
B: D E
  generate_B
```

Daneben können Sie entsprechende Makros `$(MAKRO_NAME)`, `wildcard` und Platzhalter verwenden, um die Beschreibung effizienter und kompakter zu gestalten. Ein typisches Makefile für eine eingebettetes C Projekt (Arduino) stellt sich wie folgt dar:

```
# Source, Executable, Includes, Library Defines
INCL   = loop.h defs.h
SRC    = a.c b.c d.c      # c Code-Dateien
OBJ    = $(SRC:.c=.o)
LIBS   = -lgen
EXE    = program

# Compiler, Linker Defines
CC     = /usr/bin/gcc
CFLAGS = -ansi -pedantic -Wall -O2
LIBPATH = -L.
LDFLAGS = -o $(EXE) $(LIBPATH) $(LIBS)
RM     = /bin/rm -f

# Compile and Assemble C Source Files into Object Files
%.o: %.c
  $(CC) -c $(CFLAGS) $*.c

# Link all Object Files with external Libraries into Binaries
$(EXE): $(OBJ)
  $(CC) $(LDFLAGS) $(OBJ)

# Objects depend on these Libraries
$(OBJ): $(INCL)
```

```
# Clean Up Objects, Executables, Dumps out of source directory
clean:
    $(RM) $(OBJ) $(EXE) core a.out
```

MERKE: Die Einschübe im MAKEFILE sind keine Leerzeichen, sondern Tabulatorshifts!

Das Hilfsprogramm make ist Teil des POSIX-Standards.

# Chapter 19

## Continuous Integration

Parameter	Kursinformationen
<b>Veranstaltung</b>	Vorlesung Softwareentwicklung
Semester	Sommersemester 2021
<b>Hochschule:</b>	Technische Universität Freiberg
Inhalte:	Verwendung der Buildtools und der Features von GitHub in einer CI Toolchain
Link auf den GitHub:	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/18_ContinuousIntegration.md">https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/18_ContinuousIntegration.md</a>
Autoren	@author

### Exkurs: Alternative Konzepte der Programmentwicklung

Das Jupyter Notebook ist eine Open-Source-Webanwendung Dokumente zu erstellen und zu teilen, die Live-Code, Gleichungen, Berechnungsergebnisse, Visualisierungen und andere Multimedia-Ressourcen zusammen mit erklärendem Text in einem einzigen Dokument integrieren. Dabei werden Markdown-Elemente mit verschiedenen Ausführungsumgebungen unterschiedlicher Sprachen (sogenannten Kernels) kombiniert. Notebooks sind insbesondere im Data Science-Kontext präsent, wo ganze Verarbeitungsketten von der Datenbereinigung und -transformation, numerische Simulation, explorative Datenanalyse, Datenvisualisierung, statistische Modellierung, maschinelles Lernen, Deep Learning usw. damit in Python umgesetzt werden.

Ein Jupyter-Notebook strukturiert die Implementierung in einzelnen Codeblöcke, die in beliebiger Reihung ausgeführt werden können. Zunächst geben Datenwissenschaftler Programmiercode oder Text in rechteckige "Zellen" auf einer Front-End-Webseite ein. Der Browser leitet den Code dann an einen Back-End-"Kernel" weiter, der den Code ausführt und die Ergebnisse zurück gibt. Es wurden bereits viele Jupyter-Kernel erstellt, die Dutzende von Programmiersprachen - unter anderem C# - unterstützen. Die Kernel müssen sich nicht auf dem lokalen Computer befinden.

Damit ist es eine interaktive Datenverarbeitung möglich, eine Umgebung, in der Benutzer Code ausführen, beobachten was passiert, interaktive Änderungen einpflegen usw. Aus diesem Grund eignet sich das Format gut um Tutorials oder interaktive Handbücher zu erstellen.

Neben lokalen Jupyter Installationen bieten sich verschiedene Webdienste an:

Projekt	Link	Kernel	Besonderheit
Collaboratory-Projekt von Google	<a href="#">colab</a>	Python	GPU Unterstützung, Teamarbeit möglich
Binder	<a href="#">binder</a>	Python, R, Julia, ...	
Kaggle	<a href="#">kaggle</a>	Python, R	Sammlung von ausführbaren Lerninhalten

Eine gute Übersicht zu den Features bietet die Webseite [DataSchool](#).

Visual Studio Code integriert ein eigenes Plugin für die Ausführung von .Net Interactive Sessions [Tutorial](#)

**Nachteil 1:** Die Kombinierbarkeit von Markdown und ausführbarem Sourcecode ist die zentrale Stärke von Jupyter Notebooks aber auch ihre Schwäche!

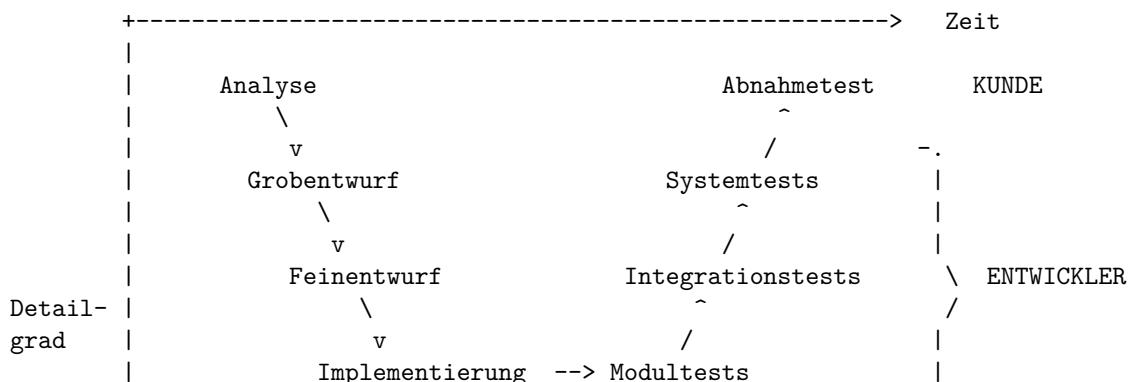
Das übergreifende Datenformat macht die Nachvollziehbarkeit von Code Änderungen schwer. Vor jedem Commit sollten entsprechend zumindest die Ausgaben gelöscht werden.

```
"cells": [
{
  "cell_type": "markdown",
  "id": "33b4a983",
  "metadata": {},
  "source": [
    "# C# in Jupyter Notebooks"
  ]
},
{
  "cell_type": "code",
  "execution_count": null,
  "id": "6c91aadb",
  "metadata": {},
  "outputs": [],
  "source": [
    "Console.WriteLine(\"Jupyter + .Net\")"
  ]
},
{
  "cell_type": "code",
  "execution_count": null,
  "id": "bc55e2c6",
  "metadata": {},
  "outputs": [],
  "source": [
    "source": [
      "var i = 5;\n",
      "var j = 6;"
    ]
  ],
  "text": "var i = 5;\nvar j = 6;\""
}];
```

**Nachteil 2:** Die beliebige Reihung der Aufrufe lässt einen mitunter die Übersicht verlieren.

**Merke:** Jupyter Notebooks sind ein hervorragendes Werkzeug für schnelle Prototypen, API-Dokumentationen oder Vorträge mit Live Hacks aber ungeeignet für Projekte [persönliche Meinung des Vortragenden :-)].

## Continuous integration (CI)



|  
v

Continuous integration (CI) zielt darauf ab, die Qualität der Software zu verbessern und die Lieferzeit zu verkürzen, indem die das Gesamtprojekt kontinuierlich realisiert wird. Darauf aufbauend können die Konzepte der Qualitätskontroller automatisiert auf einer höheren Integrationsebene umgesetzt werden.

**Tests lokal ausführen** ... erste Stufe der CI ist die Durchführung lokaler Unit-Test in der lokalen Umgebung des Entwicklers. Auf diese Weise wird vermieden, dass die individuelle, aktuelle Entwicklungsarbeit das Gesamtprojekt beeinflusst.

**Code in CI kompilieren** ... ein Build-Server kompiliert den Code periodisch oder sogar nach jedem Commit und meldet die Ergebnisse an die Entwickler.

**Tests in CI durchführen** ... Zusätzlich zu automatisierten Unit-Tests werden kontinuierliche Prozesse der Qualitätskontrolle implementieren, die statische Analysen durchführen, die Leistung vermessen und profilieren, Dokumentationen extrahieren, etc.

**Bereitstellen eines Artifacts von CI** ... Nun ist CI oft mit kontinuierlicher Bereitstellung oder kontinuierlichem Einsatz in der so genannten CI/CD-Pipeline verflochten. CI stellt sicher, dass die auf der Hauptleitung eingecheckte Software immer in einem Zustand ist, der den Benutzern zur Verfügung gestellt werden kann, und CD macht den Bereitstellungsprozess vollständig automatisiert.

Damit ergeben sich folgende Aktivitäten, die für einen CI Realisierung benötigt werden:

- Aufbau und Management eines zentralen Code-Repositorys.
- Aufbau und Management eines zentralen Build-Tools.
- Schreiben von automatisierten Tests und Integration in ein Feedbacksystem
- Darstellung von Build-Resultaten und Artifacts

Merke: Unterschätzen Sie den Aufwand für die Realisierung und Wartung der Tool-Chain nicht. Häufig müssen hier zu Beginn des Projektes grundsätzliche Entscheidungen getroffen werden, die zumindest mittelfristige Auswirkungen auf das Projekt haben.

## CI Umsetzung mit GitHub

Welche Elemente machen somit eine CI Pipeline aus:

- **Wann** (Trigger) soll
- **Was** (Job)
- **Wo** (Runner) ausgeführt werden um
- **Welches Ergebnis** (Artefakt) zu generieren

Beschrieben werden die Pipelines unter GitHub und Gitlab in YAML einer Auszeichnungssprache für Datenstrukturen über sogenannte Folgen. Die YAML-Datei, die die Pipeline-Konfiguration spezifizieren müssen im GitHub-Repositorys im Verzeichnis .github/workflows liegen.

```
# Kommentare sind auch erlaubt
scalar : 5
collection:
  variable1 : Tralla
  variable2 : Trulla
  variable3: |
    Hier steht jetzt plötzlich
    Ein Ausdruck mit mehreren Zeilen
list:
  - variableA: Wert1
  - variableB: 5
```

Die Grundstruktur folgt dabei einer Kombination der Elemente `name`, `on` (Wann) und `jobs` (was):

Generate Documentation

```
on:
  push:
    branches:
      - master
```

```

paths:
- 'docs/**'

jobs:
  build:
    name: Build
    runs-on: ubuntu-latest
    steps:
      - run: scripts/generate_documentation.sh
    env:

```

**Trigger**

Das Ausführen einer Pipeline wird aus verschiedenen Quellen resultieren und mit Bedingungen verknüpft werden.

```

on: push

on: [push, pull_request]

on:
  push:
    branches:
    - master
  pull_request:
    branches:
    - master

on:
  push:
    branches:
    - master
  paths:
    - 'docs/**'

on:
  schedule:
  - cron: '5 * * * *'

```

## Runner

Die Ausführung des “Was” kann auf jedem Rechner in Form eines Runners erfolgen, der entweder bei GitHub oder unabhängig gehostet wird. Wenn ein Runner einen Job aufnimmt, führt er die Aktionen des Jobs aus und meldet den Fortschritt, die Protokolle und die Endergebnisse an GitHub zurück.

```

runs-on: ubuntu-latest

runs-on: [self-hosted, linux, ARM32]

strategy:
  matrix:
    os: [ubuntu-16.04, ubuntu-18.04]
    node: [6, 8, 10]

```

## Jobs

Der Runner stellt eine Ausführungsumgebung bereit innerhalb derer Sie nun alle auch lokal möglichen Operationen umsetzen können. Ein häufig verwendeter Ansatz ist die Verwendung von `docker`-Containern, die eine Softwarekonfiguration bereitstellen.

Der Marketplace [Link](#) stellt verschiedene Actions bereit, die häufige Verwendung finden. Das bedeutet, dass man sich die Funktionalität nicht selbstständig zusammentragen muss, sondern allein über die Parameterisierung eine Anpassung vornimmt.

Für den Buildprozess einer .NET Anwendung werden im folgenden Beispiel Actions für das auschecken und die Bereitstellung des .NET Cores der Version 3.1.1 genutzt.

```

name: .NET Build App

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2
      - name: Setup .NET Core
        uses: actions/setup-dotnet@v1
        with:
          dotnet-version: 3.1.101
      - name: Install dependencies
        run: |
          cd src/App
          dotnet restore
      - name: Build
        run: |
          cd src/App
          dotnet build --configuration Release --no-restore

```

## Ergebnis

Artefakte ermöglichen es, Daten zwischen Aufträgen in einem Workflow auszutauschen und Daten zu speichern, sobald dieser Workflow abgeschlossen ist.

```

- uses: actions/upload-artifact@v2
  with:
    name: my-artifact
    path: path/to/artifact/world.txt

```

Die Stärken von GitHub-Actions liegen in der unmittelbaren Integration in das Repository-System. Damit entfällt die Bereitstellung einer weiteren Infrastruktur zumal automatische Check-ins von Ergebnissen einer CI Toolchain sehr kompakt möglich sind.

Welche Mängel gibt es noch im Gebrauch der GitHub-Actions?

- Workflows können nur eingeschränkt wiederverwendet werden.
- Testergebnisse und Analyseresultate können nicht über integrierte Formate ausgegeben werden (Wir sehen es gleich bei der Visualisierung der Unit-Tests im Beispiel)
- die Sequenz der Ausführungen mehrerer Pipelines ist nicht steuerbar
- die Liste der verfügbaren Actions ist noch sehr überschaubar

Merke: Man merkt an einigen Stellen der GitHub CI Implementation an, dass diese noch recht jung ist. Features die für andere Tools etabliert sind, fehlen noch.

Alternative Realisierungen lassen sich zum Beispiel mit [Jenkins](#) oder [TravisCI](#) unabhängig von einem ursprünglichen Versionsmanagementsystem evaluieren.

## Anwendungsbeispiel 1

Geben Sie die jeweiligen Anteile der verschiedenen Mitstreiter an einem Projekt in der README.md aus. Wer hat zum Beispiel wie viele Codezeilen realisiert.

Werfen wir dazu einen Blick auf das [Anwendungsbeispiel](#). Folgende Bearbeitungsschritte werden im zugehörigen Python Skript durchlaufen:

---

### Bedeutung

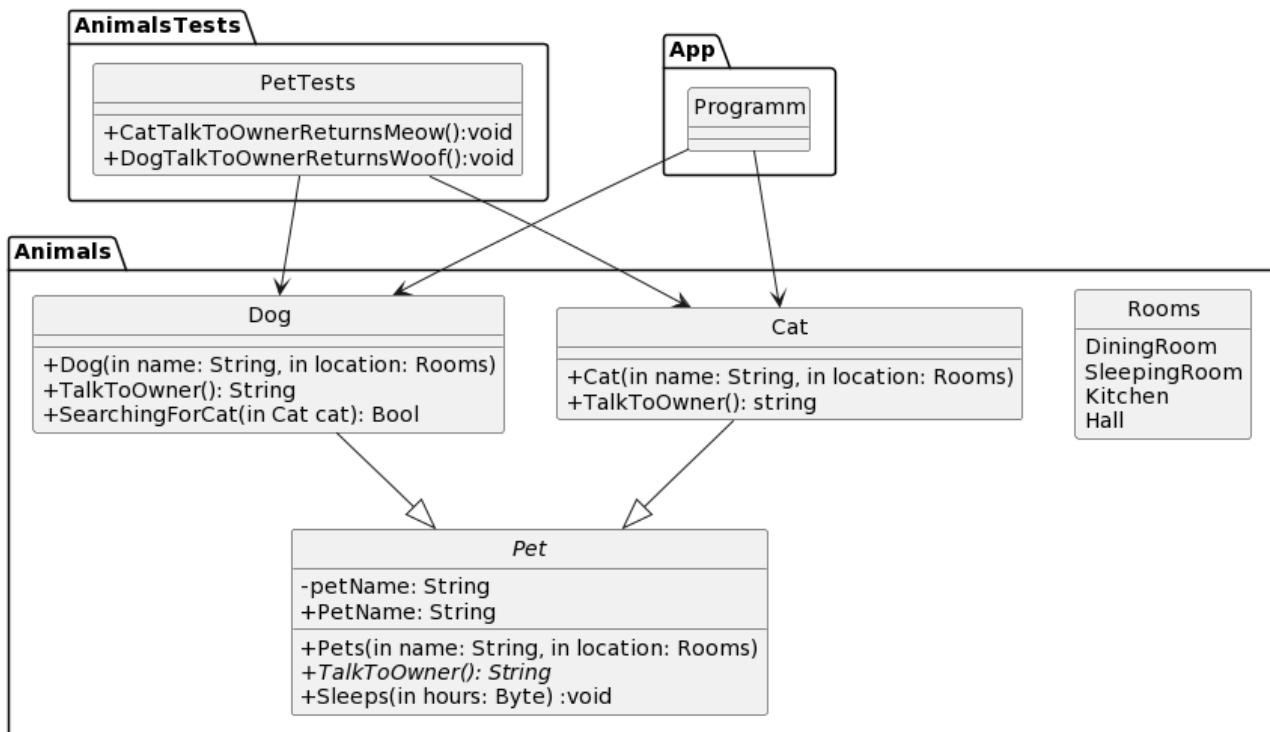
---

1. Extrahieren der Informationen mit Hilfe des Paketes `github2pandas`
  2. Generierung der Basisdatentabelle durch Merge von `pdEdits` und `pdCommits`
  3. Aggregieren der hinzugefügten und gelöschten Zeilen durch `groupby`
  4. Austauschen der Daten in der `README.md` Datei
  5. Generieren eines neuen Diagramms das bereits in der `README.md` Datei eingebunden ist.
  6. Commit und Push Operation mit den neuen Daten
- 

Die zweimalige Ausführung der Action pro Tag wird durch einen Timer getriggert.

## Anwendungsbeispiel 2

Wir wollen folgenden Entwurf in einem kontinuierlichen Entwicklungsfluss realisieren.



Für das Projekt entwerfen wir folgende Struktur:

```

doc
└ ...
Doxyfile
README.md
src
└ Animals
    └ Rooms.cs
    └ Cat.cs
    └ Dog.cs
    └ Pet.cs
    └ bin
        └ Debug
            └ ...
    └ obj
        └ ...
    └ Animals.csproj      <- .NET Projektdatei
App
└ Program.cs          <- "Main" Klasse
  
```

```

    └── App.csproj
  └── test           <- Unit-Tests für die Anwendung
    └── AnimalsTests
      ├── AnimalsTests.csproj  <- .NET Projektdatei
      ├── bin
      │   └── Debug
      │       └── ...
      ├── obj
      │   └── ...
      └── PetTest.cs
  .github
    └── workflow        <- CI Toolchain

```

Sie finden das Projekt unter <https://github.com/SebastianZug/CSharpExample>.

## Realisierung der Projektstruktur

```

mkdir src
cd src
mkdir Animals
dotnet new classlib
cd ..
mkdir App
cd App
dotnet new console

```

Allerdings fehlt jetzt noch die Verbindung zwischen den beiden Projekten! Entsprechend müssen wir die zugehörigen Referenzen in den Project-Files integrieren. Dies kann entweder in der Konsole oder mittels Textoperation erfolgen.

```

dotnet add reference ..\Animals\Animals.csproj
<ItemGroup>
  <ProjectReference Include="..\Animals\Animals.csproj" />
</ItemGroup>

```

Unsere Anwendung ruft einige der Funktionalitäten des `classlib` Projektes auf.

```

using System;
using System.Collections.Generic;
using Pets;

namespace ConsoleApplication {
  public class MyLittleZoo {
    public static void Main (string[] args) {
      Dog Willy = new Dog("Willy", Rooms.DiningRoom);
      Cat Kitty = new Cat ("KatziTatzi", Rooms.Kitchen);
      List<Pet> pets = new List<Pet> {Willy, Kitty};

      foreach (var pet in pets) {
        Console.WriteLine (pet.TalkToOwner ());
      }

      Willy.SearchForCat(Kitty);
      Willy.LocalizedInRoom = Rooms.Kitchen;
      Willy.SearchForCat(Kitty);
    }
  }
}

```

## Automatischer Build Prozess

Der automatische Buildprozess ist im einfachsten Fall eine Kopie des lokalen. Allerdings ist das Resultat in diesem Fall nur bedingt aussagekräftig. Vielmehr wollen wir mit dem serverseitigen Test ja die Übertragbarkeit

unserer Lösung auf unterschiedliche Betriebssysteme /.NET Frameworks evaluieren.

Für den ersten Fall müssen wir im Action File statt eines einfachen `run-on` eine Matrix von Runnern angeben. Im zweiten Fall muss auch der zugehörige Programmcode angepasst werden. Informieren Sie sich unter [Link](#), wie Sie unterschiedliche Frameworks adressieren können.

```
name: .NET Build App

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2
      - name: Setup .NET Core
        uses: actions/setup-dotnet@v1
        with:
          dotnet-version: 3.1.101
      - name: Install dependencies
        run: |
          cd src/App
          dotnet restore
      - name: Build
        run: |
          cd src/App
          dotnet build --configuration Release --no-restore
```

## Generierung der Dokumentation

Die Dokumentation wird unter Hinzuziehung von Doxygen erzeugt. Dieser Vorgang besteht aus drei Schritten:

1. dem Checkout des aktuellen Projektes in den Runner
2. dem Erzeugen der HTML Dokumentation ausgehend von der Konfiguration in der Datei `Doxyfile`
3. dem Publizieren des Ergebnisses im Branch `gh-pages`.

Mit dem letzten Schritt entsteht eine Projektwebseite, die die Dokumentation enthält. Zusätzlich wurde im Doxyfile die Integration des README.md Files als “Masterseite” konfiguriert.

```
...
INPUT          += ./README.md \
                  ./src/Animals

USE_MDFILE_AS_MAINPAGE = ./README.md
...

name: Doxygen

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:
  build:
```

```

runs-on: ubuntu-latest

steps:
- uses: actions/checkout@v2
- name: Doxygen Action
  uses: mattnotmitt/doxygen-action@v1.1.0
  with:
    doxyfile-path: "./Doxyfile"
    working-directory: "."
- name: Deploy
  uses: peaceiris/actions-gh-pages@v3
  with:
    publish_dir: ./docs

```

Die Darstellung ist unter <https://sebastianzug.github.io/CSharpExample/> für das Projekt sichtbar.

## Erweiterung der Tests

Bisher werden nur die Ausgaben von `Dog` und `Cat` untersucht. Lassen Sie uns einen Test integrieren, der die Methode `.SearchForCat()` evaluiert.

```

[Theory]
[InlineData(Rooms.Kitchen, Rooms.DiningRoom, false)]
[InlineData(Rooms.Kitchen, Rooms.Kitchen, true)]
public void DocCheckCatSearching(Rooms kittysLocation, Rooms willysLocation, bool pattern) {
    Cat Kitty = new Cat ("KatziTatzi", kittysLocation);
    Dog Willy = new Dog("Willy", willysLocation);
    bool actual = Willy.SearchForCat(Kitty);
    Assert.Equal (pattern, actual);
}

```

Die Tests sollen nun als CI-Tests auf dem GitHub-Server ausgeführt werden. Entsprechend ist die Integration einer neuen Action notwendig.

```

name: .NET Test Animals

on:
  push:
    branches: [ master ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Setup .NET Core
      uses: actions/setup-dotnet@v1
      with:
        dotnet-version: 3.1.101
    - name: Run Tests
      run: |
        cd src/Tests
        dotnet test

```

## Aufgaben der Woche

[ ] Klonen Sie mein Repository von <https://github.com/SebastianZug/CSharpExample> [ ] Ergänzen Sie in der Build Action ein weiteres Target, zum Beispiel “Windows” [ ] Erweitern Sie das Ganze um weitere Unit-Tests, ergänzen Sie den Eintrag in der README.md Datei [ ] Übernehmen Sie das Konzept der automatischen

Generierung eines Klassendiagramm aus den Übungsblättern, so dass auf dem Deckblatt der Dokumentation die aktuelle Klassenstruktur, die automatisch generiert wurde, sichtbar wird.

# Chapter 20

## Generics

---

Parameter	Kursinformationen
Veranstaltung	Vorlesung Softwareentwicklung
Semester	Sommersemester 2021
Hochschule:	Technische Universität Freiberg
Inhalte:	Generics und deren Anwendung
Link auf den GitHub:	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/19_Generics.md">https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/19_Generics.md</a>
Autoren	@author

---

## Motivation

Versuchen wir nach dem Exkurs zur UML Modellierung und den Werkzeugen, den roten Faden der C# Programmierung wieder aufzunehmen. In dieser Woche wollen wir uns mit der Frage der *Generics* und damit der Frage beschäftigen, wie wir ohne eine spezifische Berücksichtigung von Datentypen wiederverwendbarer Code schreiben können.

Nehmen wir an, dass Sie ohne die entsprechenden .NET-Bibliotheken eine Liste für `int`-Werte implementieren sollen. Warum funktioniert das Ganze nicht mit unserem bisherigen Array-Konzept?

```
using System;
using System.Reflection;
using System.Reflection.Emit;

public class Program{
    public static void Main(string[] args){
        var myArray = new[] { 1, 2, 3, 4, 5 };
        foreach (int i in myArray)
        {
            System.Console.Write("{0} ", i);
        }
        Type valueType = myArray.GetType();
        Console.WriteLine("\nmyArray: Type is {0}", valueType);
        Console.WriteLine("\nmyArray is Array? {0}", valueType.isArray);
        //myArray.Add(7);
    }
}
```

Die Dokumentation von `Array` findet sich unter <https://docs.microsoft.com/de-de/dotnet/api/system.array?view=netcore-3.1>

**Merke:** Arrays sind in C# statisch definiert und haben keine veränderliche Größe.

Lassen Sie uns einen alternativen Ansatz bestreiten. Wir implementieren ein eigenes Konzept, dass eine verketzte Liste repräsentiert.

```

Start
| +-----+ +-----+ +-----+ +-----+
+-->| Wert [0] | +-->| Wert [1] | +-->| Wert [2] | +-->| Wert [2] | +-->
    | Referenz |---+ | Referenz |---+ | Referenz |---+ | Referenz |---+
    +-----+ +-----+ +-----+ +-----+
using System;

public class Node{
    public Node next;
    public int value;
    public Node(Node next, int value){
        this.next = next;
        this.value = value;
    }
}

public class LinkedList{
    public Node head;
    public LinkedList(int initial) {
        head = new Node(null, initial);
    }

    public void Add(int value){
        Node current = head;
        while (current.next != null){
            current = current.next;
        }
        current.next = new Node(null, value);
    }

    public int this[int index]{
        get {
            Node current = head;
            int count = 0;
            while (current != null){
                if (count == index){
                    return current.value;
                }
                current = current.next;
                count++;
            }
            return 1;
        }
    }
}

public class Program{
    public static void Main(string[] args){
        LinkedList linkedList = new LinkedList(121);
        linkedList.Add(140);
        linkedList.Add(280);
        linkedList.Add(309);
        int i = 2;
        Console.WriteLine($"Der Wert des {i}. Eintages ist {linkedList[i]}.");
    }
}

```

Was sind die Nachteile in dieses Konstrukts auf der Listenebene? Welche Lösungsansätze sehen Sie?

Im Hinblick auf die Wiederverwendbarkeit stellt sich noch eine weiteres Problem - die Lösung ist typabhängig, die Speicherung eines anderen Datentypen macht eine Neuimplementierung notwendig. Zählen Sie doch mal durch, wie oft wir aus dem `int` ein `float` machen müssten, um eine Übertragbarkeit auf Fließkommazahlen zu realisieren. Damit entstünde dann aber auch ein überwiegend redundanter Code, der eine konsistente Realisierung und Wartung erheblich erschwert.

Lösungsansatz könnte die Arbeit mit dem allgemeinen `Object`-Datentyp sein. Mittels Boxing und Unboxing würden die spezifischen Datentypen auf diesen abgebildet.

Merke (Wiederholung): Alle C# Datentypen sind von `Object` abgeleitet.

```
int i = 123;
object o = i; // The following line boxes i.

o = 123;
i = (int)o; // unboxing
```

Nachteilig daran ist, dass

- diese Operation Laufzeit kostet,
- beim Auslesen der Daten eine externe (außerhalb unserer Liste liegende) Cast-Operation erforderlich macht. `float x = (float) linkedList[i]`,
- die Klasse würde alle Datentypen akzeptieren. Unter Umständen ist das aber nicht gewünscht weil zum Beispiel mit Zahlenwerten arithmetische Operationen ausgeführt werden sollen. Eine Beschränkung ist aber nicht möglich.

```
using System;

public class Cat{
    public void catchMouse(){
        Console.WriteLine("Dies kann allein die Katze!");
    }
    public void makeSound(){
        Console.WriteLine("Miau");
    }
}

public class Dog{
    public void huntCat(){
        Console.WriteLine("Dies kann allein der Hund!");
    }
    public void makeSound(){
        Console.WriteLine("Wuff");
    }
}

public class Program{
    public static void Main(string[] args){
        Cat Kitty = new Cat();
        Dog Wally = new Dog();
        Object Animal = Kitty;
        (Animal as Cat).catchMouse();
    }
}
```

*“Of course, we love bugs ... but not on run-time!” (Youtube Tutorial Generics in .NET)*

## Generische Typen

“Generics” sind seit der Version 2.0 Elemente der .Net-Sprachen und der Common Language Runtime (CLR). Sie definieren das Konzept der Typparameter, wodurch Klassen und Methoden keiner konkreten Zuordnung zu

einem Datentyp unterworfen werden. Platzhalter übernehmen die generische Repräsentation des Typen, die Ersetzung erfolgt zur Laufzeit.

```
// Generische Klassenspezifikation
public class LinkedList<T>{
    public void Add(T value){
        ...
    }
    public T this[int index]{
        ...
    }
}

// Instanzierung mit verschiedenen Datentypen
LinkedList<float> list1 = new LinkedList<float>(3.14);
LinkedList<ExampleClass> list2 = new LinkedList<ExampleClass>(myExampleClass);
LinkedList<ExampleStruct> list3 = new LinkedList<ExampleStruct>(myExampleStruct);
```

Die Vorteile des Konzepts sind offensichtlich:

- Der Compiler kann eine spezifische Typprüfung durchführen.
- Die Operationen sind effektiver, weil keine Typumwandlungen (wie beim Umweg über `Object`) realisiert werden müssen.
- Programme werden lesbarer.

Generische Klassen und Methoden vereinen Wiederverwendbarkeit, Typsicherheit und Effizienz so, wie es ihre nicht generischen Gegenstücke nicht können. Generics werden am häufigsten für Auflistungen und deren Methoden verwendet.

Was passiert eigentlich hinter den Kulissen? Im Unterschied zu C++ Templates werden C# Generics nicht zur Compile-Zeit konkretisiert, sondern zunächst in einen generischen Zwischencode übersetzt. Die eigentliche Konkretisierung findet zur Laufzeit statt, wobei Referenz- und Wertdatentypen unterschiedlich behandelt werden. Für jeden Werttyp, der den Platzhalter ersetzt wird eine konkrete Klasse erzeugt, während sich alle Referenztypen eine einzige Konkretisierung teilen. Das Laufzeitsystem erzeugt den typentsprechenden Code erst mit ersten Instanzierung der konkreten Klasse.

Die Parameterisierung eines generischen Typs beschränkt sich nicht nur auf einen Typ sondern kann mehrere Typen umfassen.

```
class MyGenericClass<T, U>
{
    ...
}
```

Hinsichtlich der Namenswahl für die generischen Typen sind sie frei, sollten aber berücksichtigen, dass für den Leser ggf. unklar ist, wie welcher konkrete Datentyp realisiert werden kann. Die Einbuchstabenvariante "T" sollte nur genutzt werden, wenn in Bezug auf einen Container die Bedeutung wirklich klar ist.

```
using System;

public class Stack<T>{
    int position = 0;
    T[] data = new T[100];

    public void Push(T newObj){
        if (position < 100){
            data[position++] = newObj;
        }
        else{
            Console.WriteLine("Array size exceeded!");
        }
    }

    public T Pop(){
        return data[position--];
```

```

        }

    public override string ToString(){
        string output = "";
        for (int i=0; i<position; i++){
            output = output + " " + data[i].ToString();
        }
        return output;
    }
}

public class Program{
    public static void Main(string[] args){
        var myStack = new Stack<int>();
        myStack.Push(3);
        myStack.Push(12);
        //myStack.Push("Hallo!");
        Console.WriteLine(myStack);
    }
}

```

## Generische Methoden

```

using System;

public class Program{

    // Tauscht zwei Variablen lhs und rhs
    static void Swap<T>(ref T lhs, ref T rhs)
    {
        T temp;
        temp = lhs;
        lhs = rhs;
        rhs = temp;
        Console.WriteLine("Hier wurde die generische Methode aufgerufen");
    }

    // Tauscht zwei Variablen lhs und rhs
    static void Swap(ref int lhs, ref int rhs)
    {
        int temp;
        temp = lhs;
        lhs = rhs;
        rhs = temp;
        Console.WriteLine("Hier wurde die konkrete Methode aufgerufen");
    }

    public static void Main(string[] args){
        int a = 99;
        int b = 1;
        //
        //      -----
        //      Abstimmung der Typen
        //      v
        Swap<int>(ref a, ref b);
        System.Console.WriteLine("a=" + a + " ,b=" + b);
        Swap(ref a, ref b);
        System.Console.WriteLine("a=" + a + " ,b=" + b);
        float x = 99F;
        float y = 1.2345F;
        Swap<float>(ref x, ref y);
    }
}

```

```

        System.Console.WriteLine("x=" + x + " ,y=" + y);
    }
}

```

Sie können das Typargument auch weglassen, der Compiler löst den Typ entsprechend auf. Eine Einschränkung oder ein Rückgabewert genügen ihm zur Ableitung des Typparameters nicht. Damit ist ein Typrückschluss bei Methoden ohne Parameter nicht möglich! Damit bewirken:

```

Swap<int>(ref a, ref b); // und
Swap(ref a, ref b);

```

einen analogen Aufruf.

Welches Fragestellungen ergeben sich aus dem Codefragment:

- Was passiert, wenn eine “identische” nicht-generische Methode bereitsteht? (vgl. Zeile 18 in obigem Beispiel)
- Welche Probleme entstehen, wenn die generische Methode in eine generische Klasse eingefügt wird?
- Wie stellen wir sicher, dass spezifische Methoden für den Datentyp überhaupt existieren?

Die erste Frage lässt sich schnell beantworten, in diesem Fall wird die nicht-generische Methode aus Effizienzgründen vorgezogen.

```

public class DoAnything<T>{

    // Tauscht zwei Variablen lhs und rhs
    static void Swap<T>(ref T lhs, ref T rhs)
    {
        T temp;
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }

    ...
}

```

Wenn eine generische Methode definiert wird, die die gleichen Typparameter wie die übergeordnete Klasse verwendet (hier T), gibt der Compiler die Warnung CS0693 aus. Innerhalb des Gültigkeitsbereichs der Methode wird der “äußere Klassentyp” durch den “inneren Methodentyp” ausgeblendet. Damit soll der Entwickler, der ggf. zwei unterschiedliche Typen avisiert darauf hingewiesen werden, dass diese hier keine Berücksichtigung finden.

“ .NET Dokumentation Compilerwarnung (Stufe 3) CS0693

Der Typparameter “Typparameter” hat denselben Namen wie der Typparameter des äußeren Typs “Typ”.

Dieser Fehler tritt bei einem generischen Member, z. B. einer Methode in einer generischen Klasse, auf. Da der Typparameter der Methode nicht notwendigerweise mit dem Typparameter der Klasse übereinstimmt, können Sie ihm nicht den gleichen Namen geben. Weitere Informationen finden Sie unter Generic Methods (Generische Methoden).

Um diese Situation zu vermeiden, verwenden Sie für einen der Typparameter einen anderen Namen. “

Verwenden Sie Beschränkungen, analog zu den generischen Typen, sinnvolle Einschränkungen für die Typparameter in Methoden gewährleisten. Das folgende Beispiel gibt als Beschränkung die Implementierung des Interfaces IComparable an, um unseren Vergleich zu realisieren.

```

using System;

public class Program{

    static void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T> {
        T temp;
        if (lhs.CompareTo(rhs) > 0)
        {
            temp = lhs;
            lhs = rhs;
            rhs = temp;
        }
    }
}

```

```

        rhs = temp;
    }
}

public static void Main(string[] args){
    int a = 99;
    int b = 1;
    SwapIfGreater<int>(ref a, ref b);
    System.Console.WriteLine("a=" + a + " ,b=" + b);
}
}

```

Was verbirgt sich hinter dem Interface `IComparable`? Werfen Sie einen Blick auf die entsprechende Dokumentation und benennen Sie die Methoden, die in Klassen, die dieses Interface implementieren, existieren müssen.

<https://docs.microsoft.com/de-de/dotnet/api/system.icomparable?view=netframework-4.8>

Achtung: Dieses Beispiel benutzt die typbehaftete Variante des `IComparable` Interfaces! Diese generiert über das Boxing und Unboxing einen unnötigen Aufwand und ist wie zu Beginn gezeigt nicht typsicher. Ersetzen Sie `IComparable` durch `IComparable`.

```

using System;

public class Animal : IComparable {
    private string name;
    private int weight;

    public Animal(string name, int weight){
        this.name = name;
        this.weight = weight;
    }

    public string Name{
        get { return name; }
    }

    public int Weight{
        get { return weight; }
    }

    public override string ToString(){
        return name + " weights " + weight + " kg";
    }

    public int CompareTo (object obj){
        if (obj == null)
            throw new ArgumentException("Object is not a valid");
        else {
            Animal otherAnimal = obj as Animal;
            return (otherAnimal.weight - weight);
        }
    }
}

public class Program{

    static void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable{
        T temp;
        if (lhs.CompareTo(rhs) > 0)
        {
            temp = lhs;
            lhs = rhs;
            rhs = temp;
        }
    }
}

```

```

        }
    }

    public static void Main(string[] args){
        Animal AnimalA = new Animal("Kitty", 10);
        Console.WriteLine(AnimalA);
        Animal AnimalB = new Animal("Wally", 30);
        Console.WriteLine(AnimalB);
        SwapIfGreater<Animal>(ref AnimalA, ref AnimalB);
        Console.WriteLine("After ordering ...");
        Console.WriteLine(AnimalA);
        Console.WriteLine(AnimalB);
    }
}

```

## Beschränkungen

Wie bereits bei den generischen Methoden angedeutet können wir mittels “Beschränkungen” sicherstellen, dass eine gültige Operation für einen Datentyp existiert.

```

using System;

public class Program{

    static int Plus<T>(T x, T y){
        return (x + y);
    }

    public static void Main(string[] args){
        int a = 1;
        int b = 2;
        Console.WriteLine(Plus<int>(a, b));
    }
}

```

Folglich ist es notwendig die Allgemeinheit der generischen Methoden oder Klassen zu beschränken. Man definiert Beschränkungen oder *Constraints*, die die Breite der verwendbaren Datentypen einschränken. Die Typprüfung bezieht diese Informationen dann ein.

Beschränkung	Das Typargument muss ...
where T : struct	... ein Werttyp sein.
where T : class	... ein Verweistyp sein.
where T : <Basisklasse>	... die Basisklasse sein oder von ihr abgeleitet sein.
where T : <Schnittstelle>	... die Schnittstelle sein oder diese implementieren.

Das folgende Beispiel setzt die Möglichkeiten der Beschränkung konsequent um und lässt nur `Employee` selbst oder abgeleitete Typen zu. Damit wird sichergestellt, dass die Methoden, die in `GenericList` verwendet werden, im Parametertypen auch existieren.

```

public class Human
{
    public Employee(string s, int i) => (Name, ID) = (s, i);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class Employee : Human
{
    ...
}

```

```

public class Customer : Human
{
...
}

public class GenericList<T> where T : Human, IComparable {

    // hier werden Methoden oder Felder der Klasse Employee unter Ausnutzung
    // der Vergleichbarkeit genutzt.

}

```

Da die Konkretisierung von Generics erst zur Laufzeit realisiert werden, ist es ggf. notwendig die spezifischen Parameter des Datentyps zur Laufzeit auszuwerten. Im folgenden sollen die Beispiele die Bedeutung dieser Vorgehensweise aufzeigen.

```

using System;
using System.Reflection;

public class Base {}

class SampleClass<T> where T : Base
{
    void Swap(ref T lhs, ref T rhs) { }
}

public class Program{

    public static void Main(string[] args){
        Type t = typeof(SampleClass<>);
        Console.WriteLine("Liegt ein generischer Typ vor? {0}",
            t.IsGenericTypeDefinition);
        Console.WriteLine("Wie ist der Typparameter benannt? {0}",
            t.GetGenericArguments());

        Type[] defparams = t.GetGenericArguments();
        foreach (Type tp in defparams){
            Console.WriteLine("\r\nType parameter: {0}", tp.Name);
            Type[] tpConstraints = tp.GetGenericParameterConstraints();
            foreach (Type constr in tpConstraints){
                Console.WriteLine("\t{0}", constr);
            }
        }
    }
}

```

## Vererbung bei generischen Typen

In der UML werden generische Typen über eine separate Box in der oberen linken Ecke der Klassendarstellung im Klassendiagramm realisiert.

Generischen Typen können wie andere Typen von einer Klasse erben und Interfaces implementieren. Die Basisklassen können dabei selbst wieder generische sein.

Ableitung	Die generische Klasse A erbt ...	Bemerkung
class A<X>: B {}	... vom konkreten Typ B.	
class A<X>: B<int> {}	... vom konkretisierten generischen Typ B	
class A<X>: B<X> {}	... vom generischen Typ mit gleichem Platzhalter	
class A: B<X> {...}		nicht erlaubt!

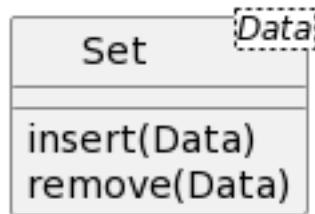


Figure 20.1: UseCaseOnlineShopII

Beispiele

```

class BaseNode { }

class BaseNodeGeneric<T> { }

class NodeConcrete<T> : BaseNode { }           // concrete type
class NodeClosed<T> : BaseNodeGeneric<int> { } //closed constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }     //open constructed type

```

Spannend wird die Typparameterisierung für generische Klassen, die von offenen konstruierten Typen erben. Hier müssen für sämtliche Basisklassen-Typparameter Typargumente bereitgestellt werden.

```

class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> { }

```

Im letzten Fall kann ausgehend von der Spezifikation von `Node6<int> A = new Node6<int>();` der Compiler nicht auf die konkrete Realisierung von U schließen.

## Anwendung

Erläutern Sie die Anwendung von generischen Typen anhand des folgenden UML-Diagramms.

Im folgenden Beispiel soll eine Liste für zwei Klassen dienen, die unterschiedliche Felder umfassen. Die Werte der Felder und die Feldnamen sollen ausgegeben werden.

Anmerkung: In der verwendeten Konfiguration gibt `GetType().GetFields()` lediglich die als public markierten Felder zurück. Entsprechend wurden diese in Angestellter und Kunde definiert.

```

using System;
using System.Reflection;

public abstract class Human{
    public Human (string name, int alter){
        this.Name = name;
        this.Alter = alter;
    }
    public string Name;
    public int Alter;
}

public class Angestellter : Human{
    public int BeiUnsSeit = 0;
    public Angestellter(string name, int alter, int beiUnsSeit) : base(name, alter){
        this.BeiUnsSeit = beiUnsSeit;
    }
}

```

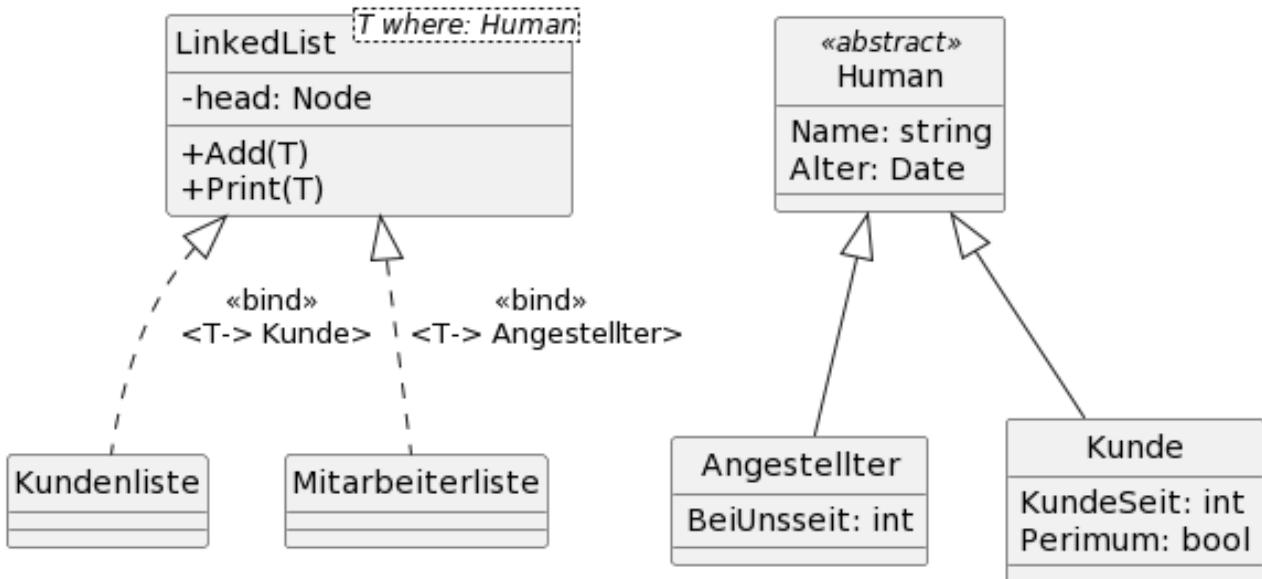


Figure 20.2: ObjectBasedList

```

    }

}

public class Kunde : Human{
    public int KundeSeit = 0;
    public bool Premium = true;
    public Kunde(string name, int alter, int kundeSeit) : base(name, alter){
        this.KundeSeit = kundeSeit;
    }
}

class LinkedList<T> where T : Human
{
    private class Node{
        public Node next;
        public T data;
        public Node(Node next, T data){
            this.next = next;
            this.data = data;
        }
    }

    private Node head;
    public LinkedList(T initial) {
        head = new Node(null, initial);
    }

    public void Add(T value){
        Node current = head;
        while (current.next != null){
            current = current.next;
        }
        current.next = new Node(null, value);
    }

    public void Print(){
        Node current = head;
        var Fields = current.data.GetType().GetFields();
    }
}

```

```

foreach (var Field in Fields)
    Console.WriteLine("{0,-20}",Field.Name);
Console.WriteLine();
while (true) {
    Fields = current.data.GetType().GetFields();
    foreach (var Field in Fields){
        Console.WriteLine("{0,-20}", Field.GetValue(current.data));
    }
    Console.WriteLine();
    if (current.next == null) break;
    current = current.next;
}
}

public class Program{
    public static void Main(string[] args){
        LinkedList<Angestellter> AngestelltenListe = new LinkedList<Angestellter>(new Angestellter("Peter", 30, 12));
        AngestelltenListe.Add(new Angestellter("Viola", 39, 14));
        AngestelltenListe.Add(new Angestellter("Miriam", 32, 5));
        AngestelltenListe.Print();
        Console.WriteLine();
        LinkedList<Kunde> KundeListe = new LinkedList<Kunde>(new Kunde("Garfield", 12, 4));
        KundeListe.Add(new Kunde("Bart", 5, 14));
        KundeListe.Add(new Kunde("Tim", 19, 5));
        KundeListe.Print();
    }
}

```

## Aufgaben der Woche

- [ ] Integrieren Sie in die oben gezeigte Liste Suchfunktionen, die nach bestimmten Namen oder Typen filtert.

# Chapter 21

## Collections

---

Parameter	Kursinformationen
<b>Veranstaltung:</b>	Vorlesung Softwareentwicklung
<b>Semester:</b>	Sommersemester 2021
<b>Hochschule:</b>	Technische Universität Freiberg
<b>Inhalte:</b>	Containertypen und deren Implementierung in C#
<b>Link auf den GitHub:</b>	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/20_Container.md">https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/20_Container.md</a>
<b>Autoren</b>	@author

---

## Collections

**Merke:** Sogenannte Container sind ein zentrales Element jeder Klassenbibliothek. Sie erlauben die Abbildung verschiedener Entitäten in einem Objekt. Im Kontext von C# wird dabei von *Collections* gesprochen.

In der vergangenen Vorlesung haben wir über die Vorteile von generischen Speicherstrukturen am Beispiel der Liste gesprochen. Allerdings ist die Möglichkeit durch die Struktur hindurchzuiterieren nicht immer die günstigste. In dieser Vorlesung wollen wir alternative Konzepte und deren Implementierung im C# Framework untersuchen.

Beginnen wir zunächst mit einem Vergleich einiger listenähnlichen Konstrukte. Diese sind in den Namespaces `System.Collections` und `System.Collections.Generic` enthalten. Um zu vermeiden, dass diese beständig mitgeführt werden, betten wir sie mit `using` in unseren Code ein.

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

public class Animal
{
    public string name;
    public Animal(string name){
        this.name = name;
    }
}

public class Program{
    public static void Main(string[] args){
        Animal[] arrayOfAnimals = new Animal[3]
        {
            new Animal("Beethoven"),
            new Animal("Bach"),
            new Animal("Mozart")
        };
    }
}
```

```

        new Animal("Kitty"),
        new Animal("Wally"),
    };
    ArrayList listOfAnimals = new ArrayList()
    {
        new Animal("Beethoven"),
        new Animal("Kitty"),
        new Animal("Wally"),
    };
    List<Animal> genericlistOfAnimals = new List<Animal>()
    {
        new Animal("Beethoven"),
        new Animal("Kitty"),
        new Animal("Wally"),
    };
    foreach (Animal pet in listOfAnimals){
        Console.WriteLine(pet.name);
    }
    listOfAnimals.RemoveAt(1);
    listOfAnimals.Add(new Animal("Flipper"));
    Console.WriteLine();
    foreach (Animal pet in listOfAnimals){
        Console.WriteLine(pet.name);
    }
    Console.WriteLine("\n");
}
}

```

Dabei setzen die vielfältigen Methoden Anforderungen an die im Container gespeicherten Werte.

```

using System;

public class Point
{
    public int x;
    public int y;
    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }
}

public class ArrayExamples {

    // Return true if X times Y is greater than 100000.
    private static bool ProductGT10(Point p)
    {
        return p.x * p.y > 100000;
    }

    public static void Main()
    {
        // Example 1 - Setzen
        String[,] myArr2 = new String[5,5];
        myArr2.SetValue( "one-three", 1, 3 );
        Console.WriteLine( "[1,3]: {0}", myArr2.GetValue( 1, 3 ) );

        // Example 2 - Sortieren
        String[] words = { "The", "QUICK", "BROWN", "FOX", "jumps",
                           "over", "the", "lazy", "dog" };
        Array.Sort(words, 1, 3);
        foreach (var word in words){

```

```

        Console.WriteLine(word + " ");
    }
    Console.WriteLine("\n");

    // Example 3 - Suchen
    // Create an array of five Point structures.
    Point[] points = { new Point(100, 200),
        new Point(150, 250), new Point(250, 375),
        new Point(275, 395), new Point(295, 450) };
    // Find the first Point structure for which X times Y
    // is greater than 100000.
    Point first = Array.Find(points, ProductGT10);
    // Display the first structure found.
    Console.WriteLine("Found: X = {0}, Y = {1}", first.x, first.y);
}
}

```

Worin liegt der Unterschied zu den bereits bekannten **Array** Implementierung?

Feature	Array	ArrayList	ArrayList<T>
Generisch?	nein	nein	ja
Anzahl der Elemente	feste Größe	variabel	variabel
Datentyp	muss homogen sein (typsicher)	kann variieren (nicht streng typisiert)	muss homogen sein
null	nicht akzeptiert	wird akzeptiert	wird akzeptiert
Dimensionen	multidimensional	-	-
	array[X] [Y]		

Die Methoden von **ArrayList** sind zum Beispiel unter <https://docs.microsoft.com/de-de/dotnet/api/system.collections.arraylist> 3.1 zu finden.

Neben den genannten existieren weitere Typen, die spezifischere Aufgaben umsetzen. Diese können entweder als sequenzielle oder als assoziative Container klassifiziert werden.

Container (in der C#-Welt sprechen wir von Collections) können durch die folgenden drei Eigenschaften charakterisiert werden:

1. Zugriff, d.h. die Art und Weise, wie auf die Objekte des Containers zugegriffen wird. Im Falle von Arrays erfolgt der Zugriff über den Array-Index. Im Falle von Stapeln (*Stack*) erfolgt der Zugriff nach der LIFO-Reihenfolge (last in, first out) und im Falle von Warteschlangen (*Queue*) nach der FIFO-Reihenfolge (first in, first out);
2. Speicherung, d.h. die Art und Weise, wie die Objekte des Containers gelagert werden;
3. Durchlaufen, d.h. die Art und Weise, wie die Objekte des Containers iteriert werden.

Von den Containerklassen wird entsprechend erwartet, dass sie folgende Methoden implementieren:

- einen leeren Container erzeugen (Konstruktor);
- Einfügen von Objekten in den Container;
- Objekte aus dem Container löschen;
- alle Objekte im Container löschen;
- auf die Objekte im Container zugreifen;
- auf die Anzahl der Objekte im Container zugreifen.

Sequenzielle-Container speichern jedes Objekt unabhängig voneinander. Auf Objekte kann direkt oder mit einem Iterator zugegriffen werden.

Ein assoziativer Container verwendet ein assoziatives Array, eine Karte oder ein Wörterbuch, das aus Schlüssel-Wert-Paaren besteht, so dass jeder Schlüssel höchstens einmal im Container erscheint. Der Schlüssel wird verwendet, um den Wert, d.h. das Objekt, zu finden, falls es im Container gespeichert ist.

Welche Container-Typen sind programmiersprachenunabhängig gängig?

Typ	Unmittelbarer Zugriff	Beschreibung
Dictionary	via Key	Wert-Schlüssel Paar
Liste	via Index	Folge von Elementen mit einem Index als Schlüssel
Queue	nur jeweils erstes Objekt	FIFO (First-In-First-Out) Speicher
Stack	nur jeweils letztes Objekt	LIFO (Last-In-First-Out) Speicher
Set		Werte ohne Dublikate
...		

Und wie sieht es mit der Performance aus? Der Beitrag des Autors [Serj-Tm](#) auf Stackoverflow vergleicht in einem Codebeispiel unterschiedliche Operationen für verschiedene Container-Typen.

Array	List<T>	Penalties	Method
00:00:01.3932446	00:00:01.6677450	1 vs 1,2	Generate
00:00:00.1856069	00:00:01.0291365	1 vs 5,5	Sum
00:00:00.4350745	00:00:00.9422126	1 vs 2,2	BlockCopy
00:00:00.2029309	00:00:00.4272936	1 vs 2,1	Sort

Fragenkatalog für die Auswahl von Collections:

Frage	Mögliche Lösungen
Sollen Elemente nach dem Auslesen verworfen werden?	Queue<T>, Stack<T>
Benötigen Sie Zugriff auf die Elemente in einer bestimmten Reihenfolge?	Queue<T> vs. LinkedList<T>
Wird die Collection in einer nebenläufigen Anwendung eingesetzt?	
Benötigen Sie Zugriff auf jedes Element über den Index?	ArrayList, StringCollection und List<T> vs. assoziativer Container
Sollen die Dateninhalte unveränderlich sein?	ImmutableArray<T>, ImmutableList<T>
Erfolgt die Indizierung anhand der Position oder anhand eines Schlüssels?	
Müssen Sie die Elemente abweichend von ihrer Eingabereihenfolge sortieren?	SortedList< TKey , TValue >
Soll der Container nur Zeichenfolgen annehmen?	StringCollection

## Containerimplementierung in Csharp

Um die Konzepte der Implementierung der Container in C# zu verstehen, versuchen wir uns nochmal an einem eigenen Konstrukt. Wir systematisieren dazu die Idee der verlinkten Liste aus der vorangegangenen Veranstaltung und fokussieren uns zunächst auf die Möglichkeit mit den C#-Bordmitteln über dieser Liste zu iterieren.

Zur Erinnerung, für die Möglichkeit der Iteration über einer Datenstruktur mittels `foreach` bedarf es der Implementierung der Interfaces `IEnumerable` und `IEnumerator`. Wir verbleiben dabei auf der generischen Seite.

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

public class GenericList<T> : IEnumerable<T>
{
    protected Node head;
    protected Node current = null;
    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
```

```
private T data;
public Node(T t){
    next = null;
    data = t;
}
public Node Next {
    get { return next; }
    set { next = value; }
}
public T Data {
    get { return data; }
    set { data = value; }
}
}

public GenericList(){
    head = null;
}

public void Add(T t)  {
    Node n = new Node(t);
    n.Next = head;
    head = n;
}

// Implementation of the iterator
public IEnumarator<T> GetEnumarator(){
    Node current = head;
    while (current != null)
    {
        yield return current.Data;
        current = current.Next;
    }
}

IEnumarator IEnumarable.GetEnumarator(){
    return GetEnumarator();
}

}

public class Animal
{
    string name;
    int age;
    public Animal(string s, int i){
        name = s;
        age = i;
    }
    public override string ToString() => name + " : " + age;
}

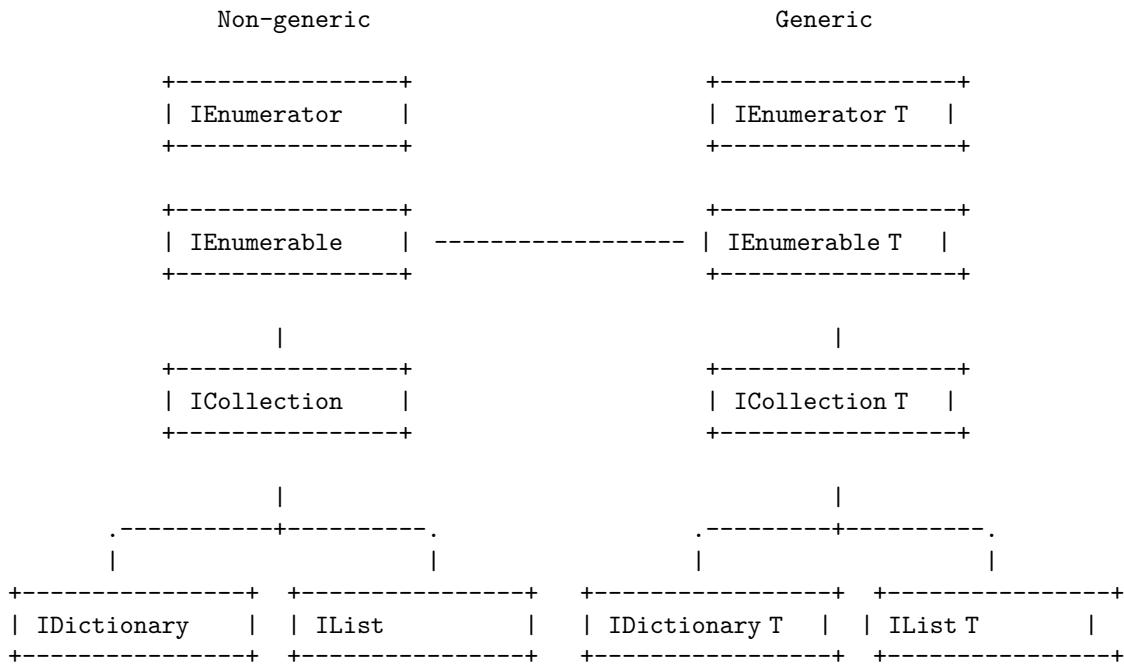
class Program
{
    public static void Main(string[] args)
    {
        GenericList<Animal> animalList = new GenericList<Animal>();
        animalList.Add(new Animal("Beethoven", 8));
        animalList.Add(new Animal("Kitty", 4));
        foreach (Animal a in animalList)
        {
            System.Console.WriteLine(a.ToString());
        }
    }
}
```

```

    }
}
}
}
```

**Achtung:** Das Beispiel implementiert das Iteratorkonzept mittels `yield`. Damit lässt sich einige Tipparbeit sparen, die bei der konventionellen Umsetzung anfallen würde, vgl. [Link](#).

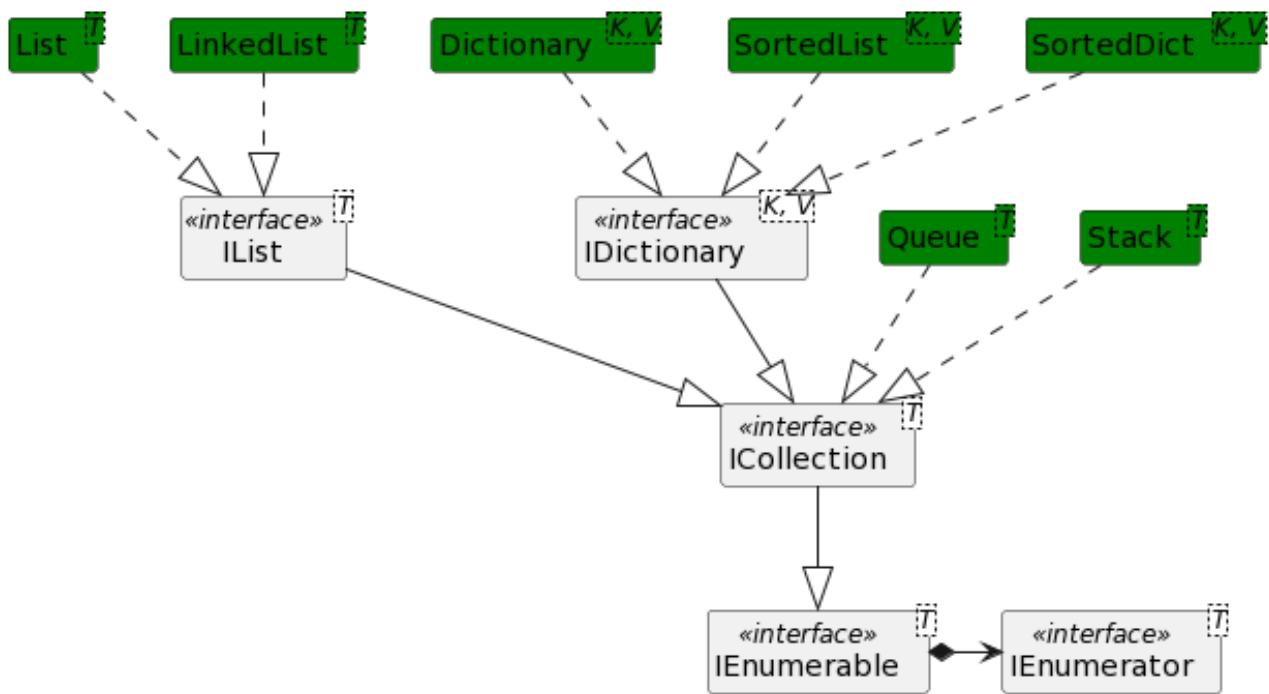
Die Methoden für das Handling der Daten beschränken sich aber auf ein `Add()` und die Iteration - hier braucht es noch deutlich mehr, um anwendbar zu sein. Um diese Funktionalität umzusetzen, greift die C#-Collections Implementierung auf eine ganze Reihe von Interfaces zurück, die den einzelnen Containern die notwendige Funktion geben.



An dieser Stelle greift das Interface `ICollection` und definiert die Methoden `Add`, `Clear`, `Contains`, `CopyTo` und `Remove`. Mit `Contains` kann geprüft werden, ob ein bestimmter Wert im Container enthalten ist. `CopyTo` extrahiert die Werte des Containers in ein Array. Dabei können bestimmte Ranges definiert werden. Die anderen Methoden sind selbsterklärend.

Schnittstelle	Spezifizierte Funktionen
<code>IEnumerable</code>	<code>GetEnumerator()</code>
<code>ICollection</code>	<code>Count()</code> , <code>Add()</code> , <code>Remove()</code>
<code>IList</code>	<code>IndexOf()</code> , <code>Insert()</code> , <code>RemoveAt()</code>
<code>IDictionary</code>	<code>Keys()</code> , <code>Values()</code> , <code>TryGetValue()</code>

Folgendes Klassendiagramm zeigt die Teile der in C# implementierten Collection-Types und deren Relationen zu den entsprechenden Interfaces.



Im Folgenden sollen Beispiele für die aufgeführten Datenstrukturen dargestellt werden.

C#	Collect	Bezeichnung	Bedeutung	
	List	unsortiertes Datenfeld indizierbarer Elemente	Im Unterschied zum Array "beliebig" erweiterbar	<a href="#">Link</a>
	SortedList	sortiertes Datenfeld	Abbildung der Reihenfolge über einen numerischen Schlüssel	<a href="#">Link</a>
	Stack	LIFO Datenstruktur		<a href="#">Link</a>
	Queue	FIFO Datenstruktur		<a href="#">Link</a>
	Dictionary	assoziatives Datenfeld	... Datenstruktur mit nicht-numerischen (fortlaufenden ) Schlüsseln, um die enthaltenen Elemente zu adressieren.	<a href="#">Link</a>

## Anwendung der Generic Collections

### List

```

using System;
using System.Reflection;
using System.Collections.Generic;

public class Program{
    public static void Main(string[] args){
        // Initialisieren mit Basiswerten, Ergänzungen der Liste
        var animals = new List<string>() { "bird", "dog" };
        animals.Add("cat");
        animals.Add("lion");
        // Fügt mehrere Objekte in die Liste ein
        animals.InsertRange(1, new string[] { "frog", "snake" });
        foreach (string value in animals)
        {
            Console.WriteLine("RESULT: " + value);
        }
    }
}
  
```

```

        }
        Console.WriteLine("In der Liste finden sich " + animals.Count + " Elemente");
        Console.WriteLine("Für die Liste reservierter Speicher (Einträge) " + animals.Capacity);
        Console.WriteLine("lion findet sich an " + animals.IndexOf("lion") + " Stelle");
        animals.Remove("lion");
        Console.WriteLine("In der Liste finden sich nun " + animals.Count + " Elemente");
    }
}

```

**Dictionary<T, U>**

```

using System;
using System.Reflection;
using System.Collections.Generic;

public class Program{
    public static void Main(string[] args){
        Dictionary<string, int> Telefonbuch = new Dictionary<string, int>();
        Telefonbuch.Add("Peter", 1234);
        Telefonbuch.Add("Paula", 5234);
        foreach( string s in Telefonbuch.Keys )
        {
            Console.WriteLine("Key = {0}\n", s);
        }
        // Enthält das Dictionary bestimmte Einträge?
        if (Telefonbuch.ContainsKey("Paula")){
            Console.WriteLine(Telefonbuch["Paula"]);
        }
        // Effektiver Zugriff
        int value;
        string key = "Peter";
        if (Telefonbuch.TryGetValue(key, out value))
        {
            Telefonbuch[key] = value + 1;
            Console.WriteLine("Wert von " + key + " " + Telefonbuch[key]);
        }
        // Mehrfache Nennung eines Eintrages
    }
}

```

**HashSet**

```

using System;
using System.Reflection;
using System.Collections.Generic;

public class Program{
    public static void Main(string[] args){
        HashSet<string> Telefonbuch1 = new HashSet<string>();
        Telefonbuch1.Add("Peter");
        Telefonbuch1.Add("Paula");
        Telefonbuch1.Add("Nadja");
        Telefonbuch1.Add("Paula");
        Console.Write("Telefonbuch 1: ");
        foreach(string s in Telefonbuch1){
            Console.Write(s + " ");
        }

        HashSet<string> Telefonbuch2 = new HashSet<string>();
        Telefonbuch2.Add("Klaus");
        Telefonbuch2.Add("Paula");
        Telefonbuch2.Add("Nadja");
    }
}

```

```

Console.WriteLine("\nTelefonbuch 2: ");
foreach(string s in Telefonbuch2){
    Console.Write(s + " ");
}

//Telefonbuch1.ExceptWith(Telefonbuch2);
Telefonbuch1.UnionWith(Telefonbuch2);
Console.WriteLine("\nMerge      2: ");
foreach(string s in Telefonbuch1){
    Console.Write(s + " ");
}
}
}

```

## Achtung!

Die heute besprochenen Inhalte finden sich in verschiedenen Formen in allen höheren Programmiersprachen wieder.

```

# initialize my_set
my_set = {1, 3}
print(my_set)

# my_set[0]
# if you uncomment the above line
# you will get an error
# TypeError: 'set' object does not support indexing

# add an element
# Output: {1, 2, 3}
my_set.add(2)
print(my_set)

# add multiple elements
my_set.update([2, 3, 4])
print(my_set)

your_set = {4, 5, 6, 7, 8}
print(your_set)

print(my_set | your_set)

```

@Pyodide.eval

## Aufgaben der Woche

- [ ] Erklären Sie, warum `Array` keine `Add`-Methode umfasst, obwohl es das Interface `IList` implementiert, dass wiederum diese einschließt. Tipp: Rufen Sie Ihr wissen um die explizite Methodenimplementierung noch mal auf.
- [ ] Die Erläuterung zu den Beschränkungen beim Einsatz von Generics im Dokument 19 basiert auf der nicht generischen Implementierung des Interfaces `IComparable`. Ersetzen Sie diese im Codebeispiel durch die generische Variante.
- [ ] Evaluieren Sie verschiedene Container in Bezug auf Methoden zum Einfügen, Löschen, etc. Generieren Sie dazu entsprechende künstliche Objekte, die Sie manipulieren „Füge 100.000 int Werte in eine Liste ein.“. Messen Sie die dafür benötigten Zeiten.



# Chapter 22

## Delegaten

---

Parameter Kursinformationen

---

**Veranstaltung:** Vorlesung Softwareentwicklung  
**Semester:** Sommersemester 2021  
**Hochschule:** Technische Universität Freiberg  
**Inhalte:** Delegaten - Konzepte und Anwendung  
**Link auf den GitHub:** [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/21\\_Delegaten.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/21_Delegaten.md)  
**Autoren:** @author

---

## Motivation und Konzept der Delegaten

Ihre Aufgabe besteht darin folgendes Code-Fragment so umzuarbeiten, so dass unterschiedliche Formen der Nutzer-Notifikation (neben Konsolenausgaben auch Emails, Instant-Messenger Nachrichten, Tonsignale) möglich sind. Welche Ideen haben Sie dazu?

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

public class VideoEncodingService{

    private string userId;
    private string filename;

    public VideoEncodingService(string filename, string userId){
        this.userId = userId;
        this.filename = filename;
    }

    public void StartVideoEncoding(){
        Console.WriteLine("The encoding job takes a while!");
        NotifyUser();
    }

    public void NotifyUser(){
        Console.WriteLine("Dear user {0}, your encoding job {1} was finished",
                        userId, filename);
    }
}
```

```

public class Program{
    public static void Main(string[] args){
        VideoEncodingService myMovie = new VideoEncodingService("007.mpeg", "12321");
        myMovie.StartVideoEncoding();
    }
}

```

Gegen das Hinzufügen weiterer Ausgabemethoden in die Klasse `VideoEncodingService` spricht die Tatsache, dass dies nicht deren zentrale Aufgabe ist. Eigentlich sollte sich die Klasse gar nicht darum kümmern müssen, welche Art der Notifikation genutzt werden soll, dies sollte dem Nutzer überlassen bleiben.

Folglich wäre es sinnvoll, wenn wir `StartVideoEncoding` eine Funktion als Parameter übergeben könnten, die wir unabhängig von der eigentlichen Klasse definiert haben.

```

public void TriggerMe(){
    // TODO
}

VideoEncodingService myMovie = new VideoEncodingService("007.mpeg",
                                                       "12321",
                                                       triggerMe);

```

In C würden wir an dieser Stelle von einem Funktionspointer sprechen.

```

// https://www.geeksforgeeks.org/function-pointer-in-c/

#include <stdio.h>
void add(int a, int b)
{
    printf("Addition is %d\n", a+b);
}

void subtract(int a, int b)
{
    printf("Subtraction is %d\n", a-b);
}

void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}

int main()
{
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[]) (int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for subtract and 2 "
           "for multiply\n");
    scanf("%d", &ch);

    if (ch > 2) return 0;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
}

```

## Grundidee

Merke: Ein Delegat ist eine Methodentyp und dient zur Deklaration von Variablen, die auf eine Methode verweisen.

Für die Anwendung sind drei Vorgänge nötig:

1. Anlegen des Delegaten (Spezifikation einer Signatur)
2. Instantiierung (Zuweisung einer signaturkorrekten Methode)
3. Aufruf der Instanz

```
// Schritt 1
// [Zugriffsattribut] delegate Rückgabewert DelegatenName(Parameterliste);
public delegate int Rechenoperation(int x, int y);

static int Addition(int x, int y){
    return x + y;
}

static int Modulo(int dividend, int divisor){
    return dividend % divisor;
}

// Schritt 2 - Instanzieren
Rechenoperation myCalc = new Rechenoperation(Addition);
// oder
Rechenoperation myCalc = Addition;

// Schritt 3 - Ausführen
myCalc(7, 9);
```

Lassen Sie uns dieses Konzept auf unsere `VideoEncodingService`-Klasse anwenden.

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

// Schritt 1
public delegate void NotifyUser(string userId, string filename);

public class VideoEncodingService
{
    private string userId;
    private string filename;

    public VideoEncodingService(string filename, string userId){
        this.userId = userId;
        this.filename = filename;
    }

    public void StartVideoEncoding(NotifyUser notifier){
        Console.WriteLine("The encoding job takes a while!");
        // Schritt 3
        notifier(userId, filename);
    }
}

public class Program
{
    // Die Notifikationsmethode ist nun Bestandteil der "Nutzerklasse"
    public static void NotifyUserByText(string userId, string filename){
        Console.WriteLine("Dear user {0}, your encoding job {1} was finished",
            userId, filename);
    }
}
```

```

    }

    public static void Main(string[] args){
        VideoEncodingService myMovie = new VideoEncodingService("007.mpeg", "12321");
        // Schritt 2
        NotifyUser notifyMe = new NotifyUser(NotifyUserByText);
        myMovie.StartVideoEncoding(notifyMe);
    }
}

```

## Was passiert hinter den Kulissen?

Was wird anhand des Aufrufes

```
NotifyUser notifyMe = new NotifyUser(NotifyUserByText);
```

deutlich? Handelt es sich bei `notifyMe` wirklich nur um eine Methode?

Delegattypen werden von der `Delegate`-Klasse im .NET Framework abgeleitet.

<https://docs.microsoft.com/de-de/dotnet/api/system.delegate?view=netframework-4.8>

Wenn der C#-Compiler Delegiertentypen verarbeitet, erzeugt er automatisch eine versiegelte Klassenableitung aus `System.MulticastDelegate`. Diese Klasse (in Verbindung mit ihrer Basisklasse, `System.Delegate`) stellt die notwendige Infrastruktur zur Verfügung, damit der Delegierte eine Liste von Methoden vorhalten kann. Der Compiler erzeugt insbesondere drei Methoden, um diese aufzurufen:

- die synchrone `Invoke()`-Methode, die aber nicht explizit von Ihrem C#-Code aufgerufen wird
- die asynchrone `BeginInvoke()` und
- `EndInvoke()` als Methoden, die die Möglichkeit bieten, die die eigentliche Methode in einem separaten Ausführungsthread zu handhaben.

Entsprechend der Codezeile `delegate int Transformer(int x);` generiert der Compiler eine spezielle `sealed class Transformer`

```
sealed class Transformer : System.MulticastDelegate
{
    ...
    public int Invoke(int x);
    public IAsyncResult BeginInvoke(int x, AsyncCallback cb, object state);
    public int EndInvoke(IAsyncResult resut);
    ...
}
```

Seit C# 2.0 ist die Syntax für die Zuweisung einer Methode an eine Delegate-Variable vereinfacht. Statt

```
NotifyUser notifyMe = new NotifyUser(this.NotifyUserByText);
```

kann nunmehr auch

```
NotifyUser notifyMe = this.NotifyUserByText;
NotifyUser notifyMe = NotifyUserByText;
```

verwendet werden.

## Multicast Delegaten

Sollten wir uns mit dem Aufruf einer Methode zufrieden geben?

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

public class Program
{
    delegate int Calc(int x, int y);
```

```

static int Add(int x, int y){
    Console.WriteLine("x + y");
    return x + y;
}

static int Multiply(int x, int y){
    Console.WriteLine("x * y");
    return x * y;
}

static int Divide(int x, int y){
    Console.WriteLine("x / y");
    return x / y;
}

public static void Main(string[] args){
    // alte Variante
    // Calc computer1 = new Calc(Divide);
    // neue Variante:
    // Calc computer2 = Divide;
    Calc computer3 = Add;
    computer3 += Multiply;
    computer3 += Multiply;
    computer3 += Divide;
    computer3 -= Add;
    Console.WriteLine("Zahl von eingebundenen Delegates {0}",
                      computer3.GetInvocationList().GetLength(0));
    Console.WriteLine("Ergebnis des letzten Methodenaufrufes {0}",
                      computer3(15, 5));
}
}

```

Merke: Der Rückgabewert des Aufrufes entspricht dem der letzten Methode.

## Schnittstellen vs. Delegaten

```

void delegate XYZ(int p);

interface IXyz {
    void doit(int p);
}

class One {
    // All four methods below can be used to implement the XYZ delegate
    void XYZ1(int p) {...}
    void XYZ2(int p) {...}
    void XYZ3(int p) {...}
    void XYZ4(int p) {...}
}

class Two : IXyz {
    public void doit(int p) {
        // Only this method could be used to call an implementation through an interface
    }
}

```

Sowohl Delegaten als auch Schnittstellen ermöglichen einem Klassendesigner, Typdeklarationen und Implementierungen zu trennen. Eine bestimmte Schnittstelle kann von jeder Klasse oder Struktur geerbt und implementiert werden. Ein Delegat kann für eine Methode in einer beliebigen Klasse erstellt werden, sofern die Methode zur Methodensignatur des Delegaten passt.

Merke: In beiden Fällen kann die Schnittstellenreferenz oder ein Delegat von einem Objekt verwendet werden.

det werden, das keine Kenntnis von der Klasse hat, die die Schnittstellen- oder Delegatmethode implementiert.

Wann ist welche der beiden Varianten vorzuziehen? Verwenden Sie einen Delegaten unter folgenden Umständen:

- Ein Ereignis-Entwurfsmuster wird verwendet.
- Es ist wünschenswert, eine statische Methode einzukapseln.
- Der Aufrufer muss nicht auf andere Eigenschaften, Methoden oder Schnittstellen des Objekts zugreifen, das die Methode implementiert.
- Eine einfache Zusammensetzung ist erwünscht.
- Eine Klasse benötigt möglicherweise mehr als eine Implementierung der Methode (siehe oben).

Verwenden Sie eine Schnittstelle wenn:

- Es gibt eine Gruppe verwandter Methoden, die aufgerufen werden können.
- Eine Klasse benötigt nur eine Implementierung der Methodesignatur.
- Die Klasse, die die Schnittstelle verwendet, möchte diese Schnittstelle in andere Schnittstellen- oder Klassentypen umwandeln.
- Die implementierte Methode ist mit dem Typ oder der Identität der Klasse verknüpft, z. B. mit Vergleichsmethoden.

Ein Beispiel für die kombinierte Anwendung eines Delegaten ist `IComparable` oder die generische Version `IComparable<T>`. `IComparable` deklariert die `CompareTo`-Methode, die eine Ganzzahl zurückgibt, die eine Beziehung angibt, die kleiner, gleich oder größer als zwei Objekte desselben Typs ist. Damit kann `IComparable` Grundlage für einen Sortieralgorithmus verwendet werden. Alternativ kann aber auch eine Delegatenvergleichsmethode übergeben werden.

Obwohl die Verwendung einer Delegatenvergleichsmethode als Grundlage eines Sortieralgorithmus gültig ist, gestaltet sich die fehlende Zuordnung nicht ideal. Da die Fähigkeit zum Vergleichen zur Klasse gehört und sich der Vergleichsalgorithmus zur Laufzeit nicht ändert, ist eine Einzelmethodenschnittstelle ideal.

Vergleichen Sie dazu [Link!](#)

## Praktische Implementierung

Neben dem Basiskonzept der Delegaten können in C# spezifischere Realisierungen umgesetzt werden, die die Anwendung flexibler bzw. effizienter machen.

### Anonyme / Lambda Funktionen

Entwicklungshistorie von C# in Bezug auf Delegaten

Version	Delegatdefinition
< 2.0	benannte Methoden
≥ 2.0	anonyme Methoden
ab 3.0	Lambdaausdrücke

Dabei lösen Lambdaausdrücke die anonymen Methoden als bevorzugten Weg zum Schreiben von Inlinecode ab. Allerdings bieten anonymous Methode eine Funktion, über die Lambdaausdrücke nicht verfügen. Anonyme Methoden ermöglichen das Auslassen der Parameterliste. Das bedeutet, dass eine anonymous Methode in Delegaten mit verschiedenen Signaturen konvertiert werden kann.

#### Anonyme Methoden

Das Erstellen anonymer Methoden verkürzt den Code, da nunmehr ein Codeblock als Delegatparameter übergeben wird.

```
// Declare a delegate pointing at an anonymous function.
Del d = delegate(int k) { /* ... */ };
```

Das folgende Codebeispiel illustriert die Verwendung. Dabei wird auch deutlich, wie eine Methodenreferenz durch einen anderen ersetzt werden kann.

```
using System;
using System.Reflection;
```

```

using System.Collections.Generic;

// Declare a delegate.
delegate void Printer(string s);

public class Program{

    static void DoWork(string k)
    {
        System.Console.WriteLine(k);
    }

    public static void Main(string[] args){
        // Anonyme Deklaration
        Printer p = delegate(string j)
        {
            Console.WriteLine(j);
        };

        p("The delegate using the anonymous method is called.");
        // Der existierende Delegat wird nun mit einer konkreten Methode
        // verknüpft
        p = DoWork;

        // alternativ könnte man auch einen neuen Delegaten anlegen
        //Printer p1 = new Printer(DoWork);
        p("The delegate using the named method is called.");
    }
}

```

## Lambda Funktionen

Ein Lambdaausdruck ist ein Codeblock, der wie ein Objekt behandelt wird. Er kann als Argument an eine Methode übergeben werden und er kann auch von Methodenaufrufen zurückgegeben werden.

```

(<Paramter>) => { expression or statement; }

(int a) => a * 2;           // einzelner Ausdruck - Ausdruckslambda
(int a) => { return a * 2; }; // Anweisungsblock - Anweisungslambda

using System;
using System.Reflection;
using System.Collections.Generic;

public class Program
{
    public delegate int Del( int Value);
    public static void Main(string[] args){
        Del obj = (Value) => {
            int x=Value*2;
            return x;
        };
        Console.WriteLine(obj(5));
    }
}

```

Jeder Lambdaausdruck kann in einen Delegat-Typ konvertiert werden. Der Delegattyp, in den ein Lambdaausdruck konvertiert werden kann, wird durch die Typen seiner Parameter und Rückgabewerte definiert.

```

using System;
using System.Collections.Generic;

public static class demo
{

```

```

public static void Main()
{
    List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 };
    List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);
    foreach (var num in evenNumbers)
    {
        Console.WriteLine(num);
    }
    Console.ReadLine();
    Console.Read();
}
}

```

## Generische Delegaten

Delegaten können auch als Generics realisiert werden. Das folgende Beispiel wendet ein Delegate “Transformer” auf ein Array von Werten an. Dabei stellt C# sicher, dass der Typ der übergebenen Parameter in der gesamten Verarbeitungskette übernommen wird.

```

using System;
using System.Reflection;
using System.Collections.Generic;

// Schritt I - Generisches Delegat
delegate T Transformer<T>(T x);

class Utility
{
    public static void Transform<T>(ref T[] values, Transformer<T> trans)
    {
        for (int i = 0; i < values.Length; ++i)
            values[i] = trans(values[i]);
    }
}

public class Program
{
    // Schritt II - Spezifische Methode, die der Delegatensignatur entspricht
    static int Square(int x){
        Console.WriteLine("This is method Square(int x)");
        return x*x;
    }

    static double Square(double x){
        Console.WriteLine("This is method Square(double x)");
        return x*x;
    }

    static void printArray<T>(T[] values){
        foreach(T i in values)
            Console.WriteLine(i + " ");
        Console.ReadLine();
    }

    public static void Main(string[] args){
        int[] values = { 1, 2, 3 };
        printArray<int>(values);

        Transformer <int> t = new Transformer<int>(Square);
        Utility.Transform<int>(ref values, t);
        printArray(values);
    }
}

```

```

    }
}
```

## Action / Func

Der generischen Idee entsprechend kann man auf die explizite Definition von eigenen Delegates vollständig verzichten. C# implementiert dafür zwei Typen vor:

```

delegate TResult Func<out TResult>();
delegate TResult Func<in T1, out TResult>(T1 arg1);
delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
delegate TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3);

delegate void Action();
delegate void Action<in T1>(T1 arg1);
delegate void Action<in T1, in T2>(T1 arg1, T2);
delegate void Action<in T1, in T2, in T3>(T1 arg1, T2 arg2, T3 arg3);
```

Im folgenden Beispiel wird die Anwendung illustriert. Dabei werden 3 Delegates genutzt um die Funktionen `PrintHello` und `Square()` zu referenzieren.

Delegate-Variante	Bedeutung
myOutput	C#1.0 Version mit konkreter Methode und individuellem Delegaten (Zeile 8)
myActionOutput	Generischer Delegatentyp ohne Rückgabewert <code>Action</code>
myFuncOutput	Generischer Delegatentyp mit Rückgabewert <code>Func</code>

```

using System;
using System.Reflection;
using System.Collections.Generic;

public delegate void Output(string text);

public class Program
{
    static void PrintHello(string text)
    {
        Console.WriteLine(text);
    }

    static int Square(int x)
    {
        Console.WriteLine("This is method Square(int x)");
        return x*x;
    }

    static double Square(double x)
    {
        Console.WriteLine("This is method Square(double x)");
        return x*x;
    }

    public static void Main(string[] args){
        Output myOutput = PrintHello;
        myOutput("Das ist eine Textausgabe");
        Action<string> myActionOutput = PrintHello;
        myActionOutput("Das ist eine Action-Testausgabe!");
        Func<float, float> myFuncOutput = Square;
        Console.WriteLine(myFuncOutput(5));
    }
}
```

Natürlich lassen sich auf `Func` und `Action` auch anonyme Methoden und Lambda Expressions anwenden!

```
Func<string, string> MyLambdaAction = text => text + "modified by Lambda";  
Console.WriteLine(MyLambdaAction("Tests"));
```

Warum würde die Verwendung von `Action` an dieser Stelle einen Fehler generieren?

## Aufgaben der Woche

- [ ] Vertiefen Sie das erlernte anhand von zusätzlichen Materialien

!?[alt-text](#)

# Chapter 23

## Events

Parameter	Kursinformationen
<b>Veranstaltung</b>	Vorlesung Softwareentwicklung
<b>Semester</b>	Sommersemester 2021
<b>Hochschule:</b>	Technische Universität Freiberg
<b>Inhalte:</b>	Events und Delegaten in der Anwendung
<b>Link auf den GitHub:</b>	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/22_Events.md">https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/22_Events.md</a>
<b>Autoren</b>	@author

## Organatorisches

### 1. Alternative Prüfungsleistung

Datum	Bemerkung
22. Juni 2021	Rückmeldung der Gruppen zur Teilnahme an der praktischen Prüfungsleistung (per Email an Prof. Zug)
bis 8. Juli 2021	Anlegen eines Repositories und Erläuterung Ihrer Zielstellungen im Wiki
bis 15. Juli 2021	Spezifikation einer Fragestellung und zugehöriger Softwareentwurf
	Bestätigung der Idee und des Entwurfs
	Bearbeitung der Aufgabenstellung in einem GitHub Projekt mit eingeladenen Betreuern
27. Juli 2021	(Tag der Klausur) Abschluss der Bearbeitung, die Repos werden gespiegelt.

- Einrichten eines eigenen Repositories
- Einladen des Partners
- Einladen von Frau Dr. Rudolf ([galinarudolf](#)) und mir ([sebastianzug](#))
- Einrichten eines Wikis für die Darstellung der Projektidee

### 2. Warum eigentlich LiaScript?

### 3. Tutor gesucht!

Für die Realisierung der Übungen in der Veranstaltung *Prozedurale Programmierung* suchen wir einen motivierten Studierenden der Freude daran hat, Nicht-Informatikern den Einstieg in die Materie zu erleichtern. Gegenstand der Veranstaltung sind die Grundzüge der Programmierung in C, ein wenig Objektorientierung in C++ und darauf aufbauend etwas Anwendung mit einem Mikrocontroller.

### 4. Morgen keine Vorlesung!

## Nachgefragt

In der letzten Veranstaltung fragte einer von Ihnen wie der selektive Zugriff auf die MultiCastDelegaten realisieren kann. Zur Erinnerung MulticastDelegate verfügt über eine verknüpfte Liste von Delegaten, die als Aufruf Liste bezeichnet wird und aus einem oder mehreren Elementen besteht. Wenn ein Multicast Delegat aufgerufen wird, werden die Delegaten in der Aufruf Liste synchron in der Reihenfolge aufgerufen, in der Sie angezeigt werden.

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

public class Program
{
    public delegate int Calc(int x, int y);

    static int Add(int x, int y){
        Console.WriteLine("x + y");
        return x + y;
    }

    static int Multiply(int x, int y){
        Console.WriteLine("x * y");
        return x * y;
    }

    static int Divide(int x, int y){
        Console.WriteLine("x / y");
        return x / y;
    }

    public static void Main(string[] args){
        Calc computer3 = Add;
        computer3 += Multiply;
        computer3 += Multiply;
        computer3 += Divide;
        computer3 -= Add;
        Console.WriteLine("Zahl von eingebundenen Delegates {0}",
                         computer3.GetInvocationList().GetLength(0));

        var x = computer3.GetInvocationList();
        Console.WriteLine("Typ der Invocation List {0}", x.GetType());
        Console.WriteLine(x[0].DynamicInvoke(1,2));
        Console.WriteLine(x[1].DynamicInvoke(3,5));
    }
}
```

## Wiederholung

Wie war das noch mal, welche Elemente (Member) zeichnen einen Klasse unter C# aus?

Bezeichnung |  
 Konstanten |  
 Felder |  
 Methoden |  
 Eigenschaften |  
 Indexer |  
 Ereignisse |  
 Operatoren |  
 Konstruktoren |  
 Finalizer |

Typen |

### Bezeichnung | Bedeutung |

Konstanten | Konstante Werte innerhalb einer Klasse |

Felder | Variablen der Klasse |

Methoden | Funktionen, die der Klasse zugeordnet sind |

Eigenschaften | Aktionen beim Lesen und Schreiben auf geschützter Variablen |

Indexer | Spezifikation eines Indexoperators für die Klasse |

Ereignisse | ? |

Operatoren | Definition von eigenen Symbolen für die Arbeit mit Instanzen der Klasse |

Konstruktoren | Bündelung von Aktionen zur Initialisierung der Klasseninstanzen |

Finalizer | Aktionen, die vor dem "Löschen" der Instanz ausgeführt werden |

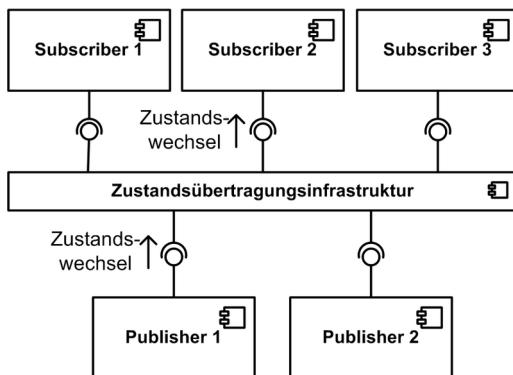
Typen | Geschachtelte Typen, die innerhalb der Klasse definiert werden |

## Motivation und Idee der Events

Was haben wir mit den Delegaten erreicht? Wir sind in der Lage aus einer Klasse, auf Methoden anderer Klassen zurückzugreifen, über die wir per Referenz informiert wurden. Die aufgerufene Methode wird der aufrufenden Klasse über einen Delegaten bekannt gegeben.

Damit sind die beiden Klassen nur über eine Funktionssignatur miteinander "lose" gekoppelt. Welche Konzepte lassen sich damit umsetzen?

### Publish-Subscribe Prinzip



Publish-Subscribe (Pub / Sub) ist ein Nachrichtenmuster, bei dem Publisher Nachrichten an Abonnenten senden. In der Softwarearchitektur bietet Pub / Sub-Messaging Ereignisbenachrichtigungen für verteilte Anwendungen, insbesondere für Anwendungen, die in kleinere, unabhängige Bausteine entkoppelt sind.

Das Publish/Subscribe-Nachrichtenmodell hat folgende Vorteile:

- Der Publisher braucht gar nicht zu wissen, wer Subscriber ist. Es erfolgt keine explizite Adressierung
- Der Subscriber ist vom Publisher entkoppelt, er empfängt nur die Nachrichten, die für ihn auch relevant sind.
- Der Subscriber kann sich jederzeit aus der Kommunikation zurückziehen oder ein anderes Topic abonnieren. Auf den Publisher hat das keine Auswirkung.
- Damit ist die Messaging-Topologie dynamisch und flexibel. Zur Laufzeit können neue Subscriber hinzukommen.

C# etabliert für die Nutzung der Pub-Sub Kommunikation *Events*. Dies sind spezielle Delegate-Variablen, die mit dem Schlüsselwort `event` als Felder von Klassen deklariert werden.

### Events in C

Der Publisher ist eine Klasse, die ein Delegaten enthält. Der Publisher entscheidet damit darüber, wann Nachrichten versand werden. Auf der anderen Seite finden sich die Subscriber-Methoden, die ausgehend vom aktivierte Delegaten im Publisher zur Ausführung kommen. Ein Subscriber hat keine Kenntnis von anderen Subscribers. Events sind ein Feature aus C# dass dieses Pattern formalisiert.

Merke: Ein Event ist ein Klassenmember, dass diejenigen Features des Delegatenkonzepts nutzt, um eine Publisher-Subscribe Interaktion zu realisieren.

Im einfachsten Fall lässt sich das Event-Konzept folgendermaßen anwenden:

```
// Schritt 1
// Wir definieren einen Delegat, der das Format der Subscriber Methoden
// (callbacks) spezifiziert - in diesem Beispiel ohne Parameter
// Hier wird ein nicht-generischer Handler genutzt.
public delegate void varAChangedHandler();

// Schritt 2
// Wir integrieren ein event in unser Publisher Klasse, dass den Delegaten
// abbildet
public class Publisher{
    public event varAChangedHandler OnAChangedHandler;

    // Schritt 3
    // Wir implementieren das "Feuern" des Events
    public magicMethod(){
        if (oldA != newA) OnAChangedHandler();
    }
}

// Schritt 4
// Implementieren des Subscribers - in diesem Fall wurde eine separate Klasse
// gewählt. Natürlich kann die Methode, die der Signatur von
// varAChangedHandler entspricht, auch in der Publisher-Klasse implementiert
// sein
public class Subscriber{
    public static void m_OnPropertyChanged(){
        Console.WriteLine("A was changed!");
    }
}

// Schritt 5
// "Einhängen" der Subscriber Methode in den Publisher-Delegate
public static void Main(string[] args){
    Publisher myPub = new Publisher();
    Subscriber mySub = new Subscriber();
    myPub.OnAChangedHandler += new varAChangedHandler(mySub.m_OnPropertyChanged);
}
```

Entsprechend der Darstellung in [Link](#) sind Events durch folgende Eigenschaften definiert:

- Der Publisher bestimmt, wann ein Ereignis ausgelöst wird. Die Subscriber bestimmen, welche Aktion als Reaktion auf das Ereignis ausgeführt wird.
- Ein Ereignis entstammt einem Publisher, kann aber mehrere Subscriber adressieren. Ein Abonnent kann mehrere Ereignisse von mehreren Herausgebern behandeln.
- Ereignisse, die keine Subscriber haben, werden nie ausgelöst. Das bedeutet, dass ein Publisher die Subscriber kennen muss.
- Ereignisse werden in der Regel verwendet, um Benutzeraktionen wie Mausklicks oder Menüauswahlen in GUI-Schnittstellen zu signalisieren.
- In der .NET Framework-Klassenbibliothek basieren Ereignisse auf dem EventHandler-Delegaten und der EventArgs-Basisklasse.

### Was passiert hinter den Kulissen?

Wenn wir ein `event` deklarieren

```
public class Publisher{
    public event VarAChangedHandler AChanged;
}
```

passieren folgende 3 Dinge:

- Der Compiler erzeugt einen privaten Delegaten mit sogenannten Event Accessoren (add und remove).

```
VarAChangedHandler aChanged; // private Delegate
public event VarAChangedHandler AChanged
{
    add {aChanged += value;}
    remove {aChanged -= value;}
}
```

- Der Compiler sucht innerhalb der Publisher Klasse nach Referenzen, die auf AChanged und lenkt diese auf das private Delegate um.
- Der Compiler mappt alle += Operationen außerhalb auf die Zugriffsmethoden add and remove.

Ok, nett, aber das würde ich gern an einem Beispiel sehen!

### Beispiel 1

Das Beispiel vereinfacht das Vorgehen, in dem es Publisher und Subscriber in einer Funktion zusammenfasst. Damit kann auf das Event uneingeschränkt zurückgegriffen werden. Dazu gehört auch, dass das Event mit Invoke ausgelöst wird.

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

public delegate void DelEventHandler();

class Program
{
    public static event DelEventHandler myEvent;
    public static void Main(string[] args){
        myEvent += new DelEventHandler(Fak1);
        myEvent += Fak2;
        myEvent += () => {Console.WriteLine("Fakultät 3");};
        myEvent.Invoke();
    }

    static void Fak1()
    {
        Console.WriteLine("Fakultät 1");
    }

    static void Fak2()
    {
        Console.WriteLine("Fakultät 2");
    }
}
```

- Erweitern Sie den Code so, dass Parameter an die Callback-Methoden übergeben werden können

### Beispiel 2

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

public delegate void DelPriceChangedHandler();

public class Stock{
    decimal price = 5;
    public event DelPriceChangedHandler OnPropertyPriceChanged;
    public decimal Price{
        get { return price; }
        set { if (price != value){
                if (OnPropertyPriceChanged != null){
```

```

        OnPropertyChanged();
        price = value;
    }
}
}

public class MailService{
    public static void stock_OnPropertyChanged(){
        Console.WriteLine("MAIL - Price of stock was changed!");
    }
}

public class Logging{
    public static void stock_OnPropertyChanged(){
        Console.WriteLine("LOG - Price of stock was changed!");
    }
}

class Program {
    public static void Main(string[] args){
        Stock GoogleStock = new Stock();
        GoogleStock.OnPropertyChanged += MailService.stock_OnPropertyChanged;
        GoogleStock.OnPropertyChanged += Logging.stock_OnPropertyChanged;
        Console.WriteLine("We change the stock price now!");
        GoogleStock.Price = 10;
    }
}

```

**Ja, aber ...** Was unterscheidet eine Delegate von einem Event? Was würde passieren, wenn wir das Key-Wort aus dem Code entfernen?

Die Implementierung wäre nicht so robust, da folgende Möglichkeiten offen ständen:

Eingriff	Bedeutung	Verhindert
GoogleStock.OnPropertyChanged = null;	Löscht alle Callback-Handler	ja
GoogleStock.OnPropertyChanged = DelPriceChangedHandler(MailService.stock_OnPropertyChanged);	Setzt einen einzigen Handler (löscht alle anderen)	ja
GoogleStock.OnPropertyChanged.Invoke();	Auslösen des Events innerhalb eines Subscribers	ja

Was fehlt Ihnen an der Implementierung?

Richtig, die Möglichkeit auf die Daten zurückzugreifen.

## Events - Praktische Implementierung

Die Möglichkeit Parameter strukturiert und wiederverwendbar zu übergeben und entsprechend generische EventHandler zu integrieren.

### Parameter

Welche Informationen sollten an die Subscriber weitergereicht werden? Zum einen der auslösende Publisher (Wer ist verantwortlich?) und ggf. weitere Daten zum Event (Warum ist die Information eingetroffen?).

Im Beispiel konzentrieren wir uns auf ein Default-Delegate, dass Bestandteil der .NET Umgebung ist

EventHandler Delegate siehe <https://docs.microsoft.com/de-de/dotnet/api/system.eventhandler?view=netframework-4.8>

Der Delegat ist dabei wie folgt definiert:

```

void EventHandler(object sender, EventArgs e)

using System;
using System.Reflection;
using System.Collections.Generic;

//Brauchen wir nicht mehr, vielmehr wird auf einen vordefinierten EventHandler
//zurückgegriffen.
//public delegate void DelPriceChangedHandler();
public class Stock{
    decimal price = 5;
    // Hier ersetzen wir unser Delegate "DelPriceChangedHandler" durch den
    // Standard-Typ EventHandler
    public event EventHandler OnPropertyPriceChanged;
    public decimal Price{
        get { return price; }
        set { if (price != value){
                if (OnPropertyPriceChanged != null){
                    OnPropertyPriceChanged(this, EventArgs.Empty );
                    price = value;
                }
            }
        }
    }
}

public class MailService{
    public static void stock_OnPropertyChanged(object sender, EventArgs e){
        Console.WriteLine("MAIL - Price of stock was changed!");
        Console.WriteLine("Wer ruft? - {0}", sender);
        Console.WriteLine("Werte? - {0}", e);
    }
}

class Program {
    public static void Main(){
        Stock GoogleStock = new Stock();
        GoogleStock.OnPropertyPriceChanged += new
            EventHandler(MailService.stock_OnPropertyChanged);
        Console.WriteLine("We changed the stock price now!");
        GoogleStock.Price = 10;
    }
}

```

## Generic Events

Nun wollen wir einen Schritt weitergehen und spezifischere Informationen mit dem Event an die Subscriber weiterreichen. Dafür implementieren wir eine von `EventArgs` abgeleitete Klasse.

```

public class PriceChangedEventArgs : EventArgs
{
    public decimal Difference { get; set; }
}

```

Diese soll den Unterschied zwischen altem und neuen Preis umfassen, damit der Subscriber die Relevanz der Information beurteilen kann.

Damit brauchen wir aber auch ein neues Delegate für unser nun nicht mehr Standard Event

```
void EventHandler(object sender, PriceChangedEventArgs e)
```

Um diese Anpassungen beim Datentyp zu realisieren existiert bereits eine generischen Form von `EventHandler` mit der Signatur

```

public delegate void EventHandler<TEventArgs>(object source,
                                                TEventArgs e)
                                                where TEventArgs: EventArgs;

using System;
using System.Reflection;
using System.Collections.Generic;

public class Stock{
    decimal price = 5;
    public event EventHandler<PriceChangedEventArgs> OnPropertyPriceChanged;
    public decimal Price{
        get { return price; }
        set { if (price != value){
                if (OnPropertyPriceChanged != null){
                    PriceChangedEventArgs myEventArgs = new PriceChangedEventArgs();
                    myEventArgs.Difference = price - value;
                    OnPropertyPriceChanged(this, myEventArgs );
                    price = value;
                }
            }
        }
    }
}

public class PriceChangedEventArgs : EventArgs
{
    public decimal Difference { get; set; }
}

public class MailService{
    public static void stock_OnPropertyChanged(object sender, PriceChangedEventArgs e){
        Console.WriteLine("MAIL - Price of stock was changed!");
        Console.WriteLine("Wer ruft? - {0}", sender);
        Console.WriteLine("Preisänderung? - {0}", e.Difference);
    }
}

class Program {
    public static void Main(string[] args){
        Stock GoogleStock = new Stock();
        GoogleStock.OnPropertyPriceChanged += new
            EventHandler<PriceChangedEventArgs>(MailService.stock_OnPropertyChanged);
        Console.WriteLine("We manipulate the stock price now!");
        GoogleStock.Price = 10;
    }
}

```

# Chapter 24

## Threads

Parameter	Kursinformationen
<b>Veranstaltung</b>	Vorlesung Softwareentwicklung
<b>Semester</b>	Sommersemester 2021
<b>Hochschule:</b>	Technische Universität Freiberg
<b>Inhalte:</b>	Multithreading Konzepte und Anwendung
<b>Link auf den GitHub:</b>	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/23_Threads.md">https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/23_Threads.md</a>
<b>Autoren</b>	@author

## Neues aus GitHub

task	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	32
3	[2, 3]	[4, 5]	[6, 7]	[8, 9]	[10, 11]	[12, 13]	14	[15, 16]	[17, 18]	[19, 20]	[21, 22]	[23, 24]	[25, 26]	[27, 28]	[29, 30]	[32, 33]	34	nan
4	nan	[4, 5]	[6, 7]	[31, 8, 9]	[10, 11]	[12, 13]	14	[15, 18]	[17, 20]	[19, 22]	[21, 24]	[23, 25]	[26, 28]	[27, 30]	[29, 30]	[32, 33]	34	[2, 15]
5	nan	[4, 5]	[7, 6]	[8, 9]	[10, 11]	[13]	3	nan	[17]	[19, 20]	[22, 21]	[24, 23]	[25, 26]	nan	nan	[33, 32]	34	[2, 15]
6	nan	[5, 4]	[7, 6]	[9, 8]	[10, 11]	[13]	nannan	[17]	[19, 20]	[21, 23]	[24, 25]	[26, 25]	nan	[29, 30]	[33, 32]	34	[2, 15]	
7	nan	[4]	nan	[8, 9]	[11, 10]	nan	nannan	nan	[20, 19]	[21, 23]	[24, 25]	[26, 25]	nan	nan	[33, 32]	34	[15, 2]	
8	nan	nan	nan	[9, 8]	nan	nan	nannan	nan	[19]	nan	nan	nan	nan	nan	[33, 32]	nan	[2, 15]	

Offenbar ist die Teamkoordination die zentrale Herausforderung für die Umsetzung der Aufgaben. Um unsere Eingriffsmöglichkeiten hier besser abzustimmen möchten wir Sie bitten uns ein entsprechendes Feedback zur aktuellen Teamkonfiguration zu geben:

Hier ist der Link: <https://panel.ovgu.de/s/957ab2a9/de.html>

## Motivation - Threads

### Threads Basics

Ein Ausführungs-Thread ist die kleinste Sequenz von programmierten Anweisungen, die unabhängig von einem Scheduler verwaltet werden kann, der typischerweise Teil des Betriebssystems ist. Die Implementierung von

Threads und Prozessen unterscheidet sich von Betriebssystem zu Betriebssystem, aber in den meisten Fällen ist ein Thread ein Bestandteil eines Prozesses. Innerhalb eines Prozesses können mehrere Threads existieren, die gleichzeitig ausgeführt werden und Ressourcen wie Speicher gemeinsam nutzen, während verschiedene Prozesse diese Ressourcen nicht gemeinsam nutzen. Insbesondere teilen sich die Threads eines Prozesses seinen ausführbaren Code und die Werte seiner dynamisch zugewiesenen Variablen und seiner nicht thread-lokalen globalen Variablen zu einem bestimmten Zeitpunkt.

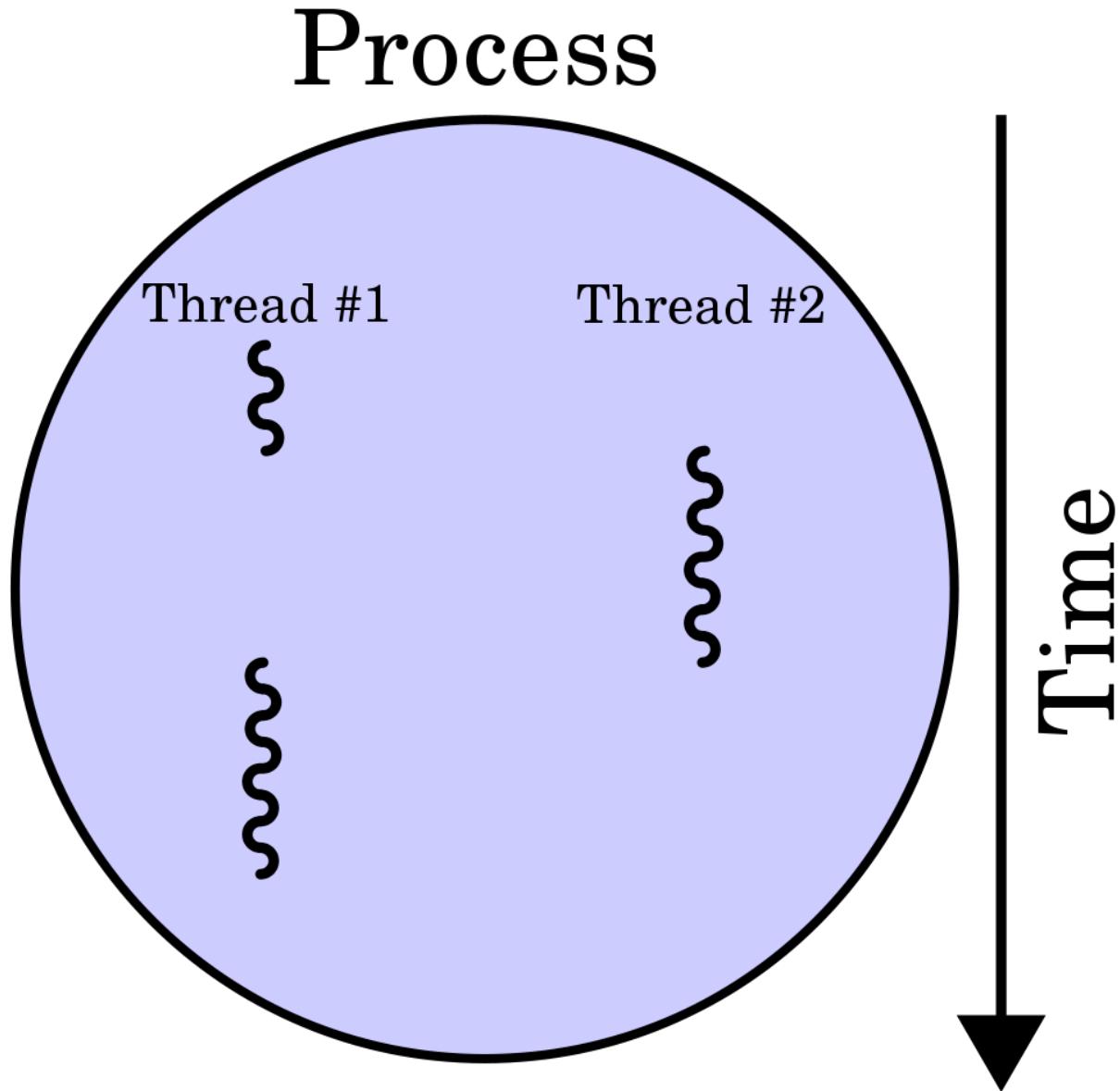


Figure 24.1: Vererbungsbeispiel

Auf eine Single-Core Rechner organisiert das Betriebssystem Zeitscheiben (unter Windows üblicherweise 20ms) um Nebenläufigkeit zu simulieren. Eine Multiprozessor-Maschine kann aber auch direkt auf die Rechenkapazität eines weiteren Prozessors ausweichen und eine echte Parallelisierung umsetzen, die allerdings im Beispiel durch den gemeinsamen Zugriff auf die Konsole limitiert ist.

## Implmementierung unter C

Die Implementierung der Klasse Thread unter C# umfasst dabei folgende Definitionen:

```
public delegate void ThreadStart();
public enum ThreadPriority {Normal, AboveNormal, BelowNormal, Highest, Lowest};
public enum ThreadState {Unstarted, Running, Suspended, Stopped, Aborted, ...};
```

```

public sealed class Thread{
    public Thread (ThreadStart startMethod);
    ...
    public string Name {get; set;};
    public ThreadPriority Priority {get; set;};
    public ThreadState TreadState {get;};
    public bool IsAlive {get;};
    public bool IsBackground{get;};
    public void Start();
    public void Join();
    public void Abort(Object);
    public static void Sleep(int milliseconds);
}

```

Um die grundlegende Verwendung des Typs `Thread` zu veranschaulichen, nehmen wir an, Sie haben eine Konsoleanwendung, in der die `CurrentThread`-Eigenschaft ein `Thread`-Objekt abruft, das den aktuell ausgeführten Thread repräsentiert.

Das folgende Beispiel kann aus spezifischen Sicherheitsgründen nicht unter Rextester ausgeführt werden.

```

using System;
using System.Threading;

class Program
{
    public static void Main(string[] args)
    {
        Thread t = Thread.CurrentThread;
        t.Name = "Primary_Thread";
        Console.WriteLine("Thread Name: {0}", t.Name);
        Console.WriteLine("Thread Status: {0}", t.IsAlive);
        Console.WriteLine("Priority: {0}", t.Priority);
        Console.WriteLine("Context ID: {0}", Thread.CurrentContext.ContextID);
        Console.WriteLine("Current application domain: {0}", Thread.GetDomain().FriendlyName);
    }
}

using System;
using System.Threading;

class Printer{
    char ch;
    int sleepTime;

    public Printer(char c, int t){
        ch = c;
        sleepTime = t;
    }

    public void Print(){
        for (int i = 0; i<10; i++){
            Console.Write(ch);
            Thread.Sleep(sleepTime);
        }
    }
}

class Program {
    public static void Main(string[] args){
        Printer a = new Printer ('a', 10);
        Printer b = new Printer ('b', 50);
        Printer c = new Printer ('c', 70);
    }
}

```

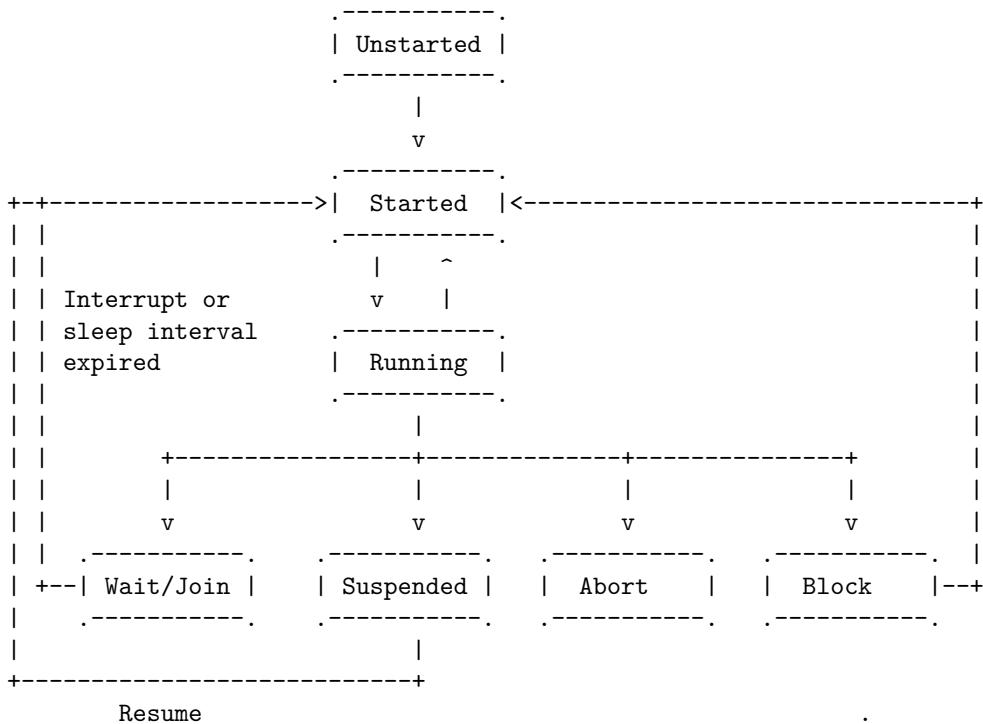
```

var watch = System.Diagnostics.Stopwatch.StartNew();
a.Print();
b.Print();
c.Print();
watch.Stop();
Console.WriteLine("\nDuration in ms: {0}", watch.ElapsedMilliseconds);

watch.Restart();
Thread PrinterA = new Thread(new ThreadStart(a.Print));
Thread PrinterB = new Thread(new ThreadStart(b.Print));
PrinterA.Start();
PrinterB.Start();
c.Print(); // Ausführung im Main-Thread
watch.Stop();
Console.WriteLine("\nDuration in ms: {0}", watch.ElapsedMilliseconds);
}
}

```

## Thread-Interaktion



Wie lässt sich eine Serialisierung von Threads realisieren? Im Beispiel soll die Ausführung des Printers C erst starten, wenn die beiden anderen Druckaufträge abgearbeitet wurden.

Methode	Bedeutung
t.Join()	Es wird so lange gewartet, bis der Thread t zum Abschluss gekommen ist.
Thread.Sleep()	Es wird für n Millisekunden gewartet.
Thread.Yield()	Gibt den erteilten Zugriff auf die CPU sofort zurück.

```

using System;
using System.Threading;

class Printer{
    char ch;
    int sleepTime;
    public Printer(char c, int t){
        ch = c;
    }
}

```

```

        sleepTime = t;
    }
    public void Print(){
        for (int i = 0; i<10; i++){
            Console.Write(ch);
            //Thread.Sleep(sleepTime);
            Thread.Yield();
        }
    }
}
class Program {
    public static void Main(string[] args){
        Printer a = new Printer ('a', 10);
        Printer b = new Printer ('b', 50);
        Printer c = new Printer ('c', 70);
        Thread PrinterA = new Thread(new ThreadStart(a.Print));
        Thread PrinterB = new Thread(new ThreadStart(b.Print));
        PrinterA.Start();
        PrinterB.Start();
        Thread.Sleep(1000);      // Zeitabhängige Verzögerung des Hauptthreads
        //PrinterA.Join();      // <-
        //PrinterB.Join();
        c.Print();
    }
}

```

Aus dem Gesamtkonzept des Threads ergeben sich mehrere Zustände, in denen sich dieser befinden kann:

Zustand	Bedeutung	Transition
unstarted	Thread ist initialisiert	t.Start();
running	Thread befindet sich gerade in der Ausführung	
WaitSleepJoin	Thread wird wegen eines Sleep oder eines Join-Befehls nicht ausgeführt. Er nutzt keine Prozessorzeit.	Ablauf des Zeitfensters, Ende des mit Join() referenzierten Threads
Suspended	Der Thread ist dauerhaft deaktiviert.	t.Resume() aktiviert ihn wieder
stopped	Bearbeitung beendet	

Jeder Thread umfasst eine Feld vom Typ `ThreadState`, dass auf verschiedenen Ebenen dessen Parameter abbildet. Um nur die für uns relevanten Informationen zu erfassen, benutzen wir eine kleine Funktion.

```

public static ThreadState DetermineThreadState(this ThreadState ts){
    return ts & (ThreadState.Unstarted |
                  ThreadState.Running |
                  ThreadState.WaitSleepJoin |
                  ThreadState.Stopped);

    bool blocked = (Thread_a.ThreadState & ThreadState.WaitSleepJoin) != 0;
}

```

Ein Thread in C# zu einem beliebigen Zeitpunkt existiert in einem der folgenden Zustände. Ein Thread liegt zu einem beliebigen Zeitpunkt nur in einem Zustand vor.

## Thread-Initialisierung

Wie wird der Thread-Objekt korrekt initialisiert? Viele Tutorials führen Beispiele auf, die wie folgt strukturiert sind, während im obigen Beispiel der Konstruktorauftrag von `Thread` ein weiteren Konstruktor `ThreadStart` adressiert:

```

Thread threadA = new Thread(ExecuteA);
threadA.Start();
// vs
Thread threadB = new Thread(new ThreadStart(ExecuteB));

```

```

using System;
using System.Threading;

class Calc
{
    int paramA = 0;
    int paramB = 0;

    public Calc(int paramA, int paramB){
        this.paramA = paramA;
        this.paramB = paramB;
    }

    // Static method
    public static void getConst()
    {
        Console.WriteLine("Static function const = {0}", 3.14);
    }

    public void process()
    {
        Console.WriteLine("Result = {0}", paramA + paramB);
    }
}

class Program
{
    static void Main()
    {
        ThreadStart threadDelegate = new ThreadStart(Calc.getConst);
        Thread newThread = new Thread(threadDelegate);
        newThread.Start();

        newThread = new Thread(Calc.getConst);      // impliziter Cast zu ThreadStart
        newThread.Start();

        Calc c = new Calc(5, 6);
        threadDelegate = new ThreadStart(c.process);
        newThread = new Thread(threadDelegate);
        newThread.Start();
    }
}

```

Der Konstruktor der Klasse `Thread` hat aber folgende Signatur:

---

Konstruktor	Initialisiert eine neue Thread Klasse ...
<code>Thread(ThreadStart)</code>	auf der Basis einer Instanz von <code>ThreadStart</code>
<code>Thread(ThreadStart,..)</code>	auf der Basis einer Instanz von <code>ThreadStart</code> unter Angabe der Größe des Stacks in Int32
	Byte (aufgerundet auf entsprechende Page Size und unter Berücksichtigung der globalen Mindestgröße)
<code>Thread(ParameterizedThreadStart)</code>	eine Instanz von <code>ParameterizedThreadStart</code>
<code>Thread(ParameterizedThreadStart,..)</code>	eine Instanz von <code>ParameterizedThreadStart</code> unter Angabe der Größe Int32 des Stacks

---

```

// impliziter Cast zu ParameterizedThreadStart
Thread threadB = new Thread(ExecuteB);
threadB.Start("abc");

// impliziter Cast und unmittelbarer Start
var threadC new Thread(SomeMethod).Start();

```

Ergänzen Sie das oben aufgeführte Beispiel `ThreadApplicationPrinter` um die Möglichkeit das auszugebene Zeichen als Parameter zu übergeben!

## Datenaustausch zwischen Threads

Jeder Thread realisiert dabei seinen eigenen Speicher, so dass die lokalen Variablen separat abgelegt werden. Die Verwendung der lokalen Variablen ist entsprechend geschützt.

```
using System;
using System.Threading;

class Program
{
    static void Execute(object output){
        for (int i = 0; i<10; i++){
            Console.WriteLine(output + i.ToString());
        }
    }

    public static void Main(string[] args){
        new Thread(Execute).Start("New Thread :");
        Execute("MainThread :");
    }
}
```

Warum werden die beiden Threads ohne Unterbrechung sequentiell abgearbeitet? Welche Ergänzung ist notwendig, um einen zyklischen Wechsel zu erzwingen?

Auf dem individuellen Stack eigenen Kopien der lokale Variablen `count` angelegt, so dass die beiden Threads keine Interaktion realisieren.

Was aber, wenn ein Datenaustausch realisiert werden soll? Eine Möglichkeit der Interaktion sind entsprechende Felder innerhalb einer gemeinsamen Objektinstanz.

Welches Problem ergibt sich aber dabei?

```
using System;
using System.Threading;

class InteractiveThreads
{
    // Gemeinsames Member der Klasse
    // [ThreadStatic] // <- gemeinsames Member innerhalb eines Threads
    public static int count = 0;

    public void AddOne(){
        count++;
        Console.WriteLine("Nachher {0}", count);
    }
}

class Program
{
    public static void Main(string[] args){
        InteractiveThreads myThreads = new InteractiveThreads();
        for (int i = 0; i<100; i++){
            new Thread(myThreads.AddOne).Start();
        }
        Thread.Sleep(10000);
        Console.WriteLine("\n Fertig {0}", InteractiveThreads.count);
    }
}
```

## Locking

Locking und Threadsicherheit sind zentrale Herausforderungen bei der Arbeit mit Multithread-Anwendungen. Wie können wir im vorhergehenden Beispiel sicherstellen, dass zwischen dem Laden von threadcount in ein Register, der Inkrementierung und dem Zurückschreiben nicht ein anderer Thread den Wert zwischenzeitlich manipuliert hat.

Für eine binäre Variable wird dabei von einem Test-And-Set Mechanismus gesprochen der Thread-sicher sein muss. Wie können wir dies erreichen? Die Prüfung und Manipulation muss atomar ausgeführt werden, das heißt an dieser Stelle darf der ausführende Thread nicht verdrängt werden.

Darauf aufbauend implementiert C# verschiedene Methoden:

Threadsicherheit	Bemerkung
“exclusive lock”	Alleiniger Zugriff auf eine Codeabschnitt
Monitor	Erweiterter lock mit Berücksichtigung von Ausnahmen
Mutex	Prozessübergreifende exklusive Sperrung
Semaphor	Zugriff auf einen Codeabschnitt durch n Threads

```
static readonly object locker = new object();

lock(locker){
    // kritische Region
}

using System;
using System.Threading;

class InteractiveThreads{
    public int count = 0;
    public void AddOne(){
        lock(this)
        {
            count = count + 1;
        }
        Console.WriteLine("count {0}", count);
    }
}

class Program {
    public static void Main(string[] args){
        InteractiveThreads myThreads = new InteractiveThreads();
        for (int i = 0; i<10; i++){
            new Thread(myThreads.AddOne).Start();
        }
        Thread.Sleep(1000);
    }
}
```

## Hintergrund und Vordergrund-Threads

Verwaltete Threads können als Hintergrund- oder Vordergrundthread definiert sein. Hintergrundthreads unterscheiden sich von Vordergrundthreads durch die Beibehaltung der Ausführungsumgebung nach dem Abschluss. Sobald alle Vordergrundthreads in einem verwalteten Prozess (wobei die EXE-Datei eine verwaltete Assembly ist) beendet sind, beendet das System alle Hintergrundthreads und fährt herunter.

```
using System;
using System.Threading;

class Printer{
    char ch;
    int sleepTime;
```

```

public Printer(char c, int t){
    ch = c;
    sleepTime = t;
}

public void Print(){
    for (int i = 0; i<10; i++){
        Console.WriteLine(ch);
        Thread.Sleep(sleepTime);
    }
}
}

class Program {
    public static void printThreadProperties(Thread currentThread){
        Console.WriteLine("{0} - {1} - {2}", currentThread.Name,
                           currentThread.Priority,
                           currentThread.IsBackground);
    }

    public static void Main(string[] args){
        Thread MainThread = Thread.CurrentThread;
        MainThread.Name = "MainThread";
        printThreadProperties(MainThread);
        Printer a = new Printer ('a', 170);
        Printer b = new Printer ('b', 50);
        Printer c = new Printer ('c', 10);
        Thread PrinterA = new Thread(new ThreadStart(a.Print));
        PrinterA.IsBackground = false;
        Thread PrinterB = new Thread(new ThreadStart(b.Print));
        printThreadProperties(PrinterA);
        printThreadProperties(PrinterB);
        PrinterA.Start();
        PrinterB.Start();
        c.Print();
    }
}

```

## Ausnahmebehandlung mit Threads

Ab .NET Framework, Version 2.0, erlaubt die CLR bei den meisten Ausnahmefehlern in Threads deren ordnungsgemäße Fortsetzung. Allerdings ist zu beachten, dass die Fehlerbehandlung innerhalb des Threads zu erfolgen hat. Unbehandelte Ausnahmen auf der Thread-Ebene führen in der Regel zum Abbruch des gesamten Programms.

Verschieben Sie die Fehlerbehandlung in den Thread!

```

using System;
using System.Threading;

class Program {
    public static void Calculate(object value){
        Console.WriteLine(5 / (int)value);
    }

    public static void Main(string[] args){
        Thread myThread = new Thread (Calculate);
        try{
            myThread.Start();
        }
        catch(DivideByZeroException)
    }
}

```

```
        {
            Console.WriteLine("Achtung - Division durch Null");
        }
    }
}
```

Analog kann das Abbrechen eines Threads als Ausnahme erkannt und in einer Behandlungsroutine organisiert werden.

```
// Beispiel aus Mösenböck, Kompaktkurs C# 7, Seite 159
using System;
using System.Threading;

class Program {
    static void Operate(){
        try{
            try{
                try{
                    while (true);
                }catch (ThreadAbortException){Console.WriteLine("inner aborted");}
                }catch (ThreadAbortException){Console.WriteLine("outer aborted");}
            }finally {Console.WriteLine("finally");}
    }

    public static void Main(string[] args){
        Thread myThread = new Thread (Operate);
        myThread.Start();
        Thread.Sleep(1);
        myThread.Abort();    // <- Expliziter Abbruch des Threads
        myThread.Join();
        Console.WriteLine("done");
    }
}
```

## Thread-Pool

Wann immer ein neuer Thread gestartet wird, bedarf es einiger 100 Millisekunden, um Speicher anzufordern, ihn zu initialisieren, usw. Diese relativ aufwändige Verfahren wird durch die Nutzung von ThreadPools beschränkt, da diese als wiederverwendbare Threads vorgesehen sind.

Die `System.Threading.ThreadPool`-Klasse stellt einer Anwendung einen Pool von "Arbeitsthreads" bereit, die vom System verwaltet werden und Ihnen die Möglichkeit bieten, sich mehr auf Anwendungsaufgaben als auf die Threadverwaltung zu konzentrieren.

```
using System;
using System.Threading;

class Program {
    // This thread procedure performs the task.
    static void Operate(Object stateInfo)
    {
        Console.WriteLine("Hello from the thread pool.");
    }

    public static void Main(string[] args){
        ThreadPool.QueueUserWorkItem(Operate);
        Console.WriteLine("Main thread does some work, then sleeps.");
        Thread.Sleep(1000);
        Console.WriteLine("Main thread exits.");
    }
}
```

Das klingt sehr praktisch, was aber sind die Einschränkungen?

- Für die Threads können keine Namen vergeben werden, damit wird das Debugging ggf. schwieriger.
- Pooled Threads sind immer Background-Threads
- Sie können keine individuellen Prioritäten festlegen.
- Blockierte Threads im Pool senken die entsprechende Performance des Pools

## Aufgaben der Woche

- [ ]



# Chapter 25

## Tasks

---

Parameter	Kursinformationen
<b>Veranstaltung</b>	Vorlesung Softwareentwicklung
<b>Semester</b>	Sommersemester 2021
<b>Hochschule:</b>	Technische Universität Freiberg
<b>Inhalte:</b>	Tasks und deren Anwendung
<b>Link auf den GitHub:</b>	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/24_Tasks.md">https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/24_Tasks.md</a>
<b>Autoren</b>	@author

---

## Logging

Unsere Ausgaben in der Console können auf Dauer ziemlich nerven ...

Lösung: Verwenden Sie ein Logging Framework!

- <https://logging.apache.org/log4net/release/manual/configuration.html>
- <https://blog.elmah.io/log4net-tutorial-the-complete-guide-for-beginners-and-pros/>

```
<log4net>
    <!-- A1 is set to be a ConsoleAppender -->
    <appender name="A1" type="log4net.Appender.ConsoleAppender">

        <!-- A1 uses PatternLayout -->
        <layout type="log4net.Layout.PatternLayout">
            <conversionPattern value="%-4timestamp [%thread] %-5level %logger %ndc - %message%newline" />
        </layout>
    </appender>

    <!-- Set root logger level to DEBUG and its only appender to A1 -->
    <root>
        <level value="DEBUG" />
        <appender-ref ref="A1" />
    </root>
</log4net>
```

## Tasks

Die prozedurale/objektorientierte Programmierung basiert auf der Idee, dass ausgehend von einem Hauptprogramm Methoden aufgerufen werden, deren Abarbeitung realisiert wird und danach zum Hauptprogramm zurückgekehrt wird.

```

using System;
using System.Threading;

class Program
{
    public static void TransmitsMessage(string output){
        Random rnd = new Random();
        Thread.Sleep(1);
        Console.WriteLine(output);
    }

    public static void Main(string[] args){
        TransmitsMessage("Here we are");
        TransmitsMessage("Best wishes from Freiberg");
        TransmitsMessage("Nice to meet you");
    }
}

```

An dieser Stelle spricht man von **synchronen** Methodenaufrufen. Das Hauptprogramm (Rufer oder Caller) stoppt, wartet auf den Abschluss des aufgerufenen Programms und setzt seine Bearbeitung erst dann fort. Das blockierende Verhalten des Rufers generiert aber einen entscheidenden Nachteil - eine fehlende Reaktionsfähigkeit für die Zeit, in der die aufgerufene Methode zum Beispiel eine Netzwerkverbindung aufbaut, Daten speichert oder Berechnungen realisiert.

Der Rufer könnte in dieser Zeit auch andere Arbeiten umsetzen. Dafür muss er aber nach dem Methodenaufruf die Kontrolle zurück bekommen und kann dann weiterarbeiten.

Ein Beispiel aus der "Praxis" - Vorbereitung eines Frühstücks:

1. Schenken Sie sich eine Tasse Kaffee ein.
2. Erhitzen Sie eine Pfanne, und braten Sie darin zwei Eier.
3. Braten Sie drei Scheiben Frühstücksspeck.
4. Toasten Sie zwei Scheiben Brot.
5. Bestreichen Sie das getoastete Brot mit Butter und Marmelade.
6. Schenken Sie sich ein Glas Orangensaft ein.

Das anschauliche Beispiel entstammt der Microsoft Dokumentation und ist unter <https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/async/> zu finden.

Eine Lösung für diesen Ansatz könnten Threads bieten.

```

using System;
using System.Threading;

class Program {
    static public int[] Result = { 0, 0, 0};
    static Random rnd = new Random();

    public static void TransmitsMessage(object index){
        Console.WriteLine("Thread {0} started!", Thread.CurrentThread.ManagedThreadId);
        // doing some fancy things here
        int delay = rnd.Next(200, 500);
        Thread.Sleep(delay); // arbitrary duration
        Result[(int)index] = delay;
        Console.WriteLine("\nThread {0} says Hello", Thread.CurrentThread.ManagedThreadId);
    }

    public static void Main(string[] args){
        Thread ThreadA = new Thread (TransmitsMessage);
        ThreadA.Start(0);
        Thread ThreadB = new Thread (TransmitsMessage);
        ThreadB.Start(1);
        Thread ThreadC = new Thread (TransmitsMessage);
        ThreadC.Start(2);
    }
}

```

```

    for (int i = 0; i<50; i++){
        Console.WriteLine("*");
        Thread.Sleep(1);
    }
    Console.WriteLine();
    Console.WriteLine("Well done, so far!");
    ThreadA.Join();
    ThreadB.Join();
    ThreadC.Join();
    Console.WriteLine("Aus die Maus!");
    foreach(int i in Result){
        Console.Write("{0} ", i);
    }
}
}
}

```

Welche Nachteile sehen Sie in dieser Lösung?

## Task Modell in C

C# stellt für die asynchrone Programmierung einen neuen Typen **Task** zur Verfügung und für die **await** und **async** Keywörter ein.

Die **Task**-Klasse bildet einen einzelnen Vorgang ab, gibt keinen Wert zurück und wird in der Regel asynchron ausgeführt. **Task** Objekte sind eine der zentralen Komponenten von der aufgabenbasierte asynchrone Muster in .NET Framework 4 eingeführt wurden. Da die Arbeit, indem geleistet eine Task Objekt in der Regel führt asynchron auf einem Threadpool-Thread anstatt synchron auf dem Hauptanwendungsthread, Sie verwenden die Thread Status-Eigenschaften des Threads, als auch die **IsCanceled**, **IsCompleted**, und **IsFaulted** Eigenschaften, um den Status eines Vorgangs zu bestimmen. In den meisten Fällen wird ein Lambda-Ausdruck verwendet, um die eigentliche Aufgabe zu spezifizieren.

```

public class Task{
    public Task (Action a);
    public TaskStatus Status {get;}
    public bool IsCompleted {get;}
    public static Task Run(Action a);
    public static Task Delay(int n);
    public void Wait();
    ...
}

public class Task<T>: Task{
    public Task (Func<T> f);
    ...
    public static Task<T> Run (Func <T> f);
    ...
}

```

Die Anwendung erfolgt dabei dem Muster:

```
Task task = new Task(() => {... Anweisungsblock ...});
Task.Start();
```

Hierbei wird deutlich, dass das **Task**-Objekt auf einem **Thread** aufbaut und lediglich eine höhere Abstraktionsstufe darstellt. Der verkürzte Aufruf mittels der statischen **Run**-Methode realisiert das gleiche Verhalten:

```
Task task = Task.Run(() => {... Anweisungsblock ...});
```

Es wäre nun möglich diesen laufenden Task aus dem Main-Thread anhand seiner Variablen **IsCompleted** zu überwachen oder mit **join** erfassen, um die Fertigstellung zu erkennen. Dieses Verhalten lässt sich zwar auch mit Threads umsetzen, mit dem höheren Abstraktionsgrad lässt sich die Komplexität des Aufrufes aber reduzieren.

Um für die Durchführung einer einzelnen Aufgabe zu warten, rufen Sie die Task. **Wait** Methode. Ein Aufruf der **Wait** Methode blockiert den aufrufenden Thread, bis die Instanz der Klasse die Ausführung abgeschlossen hat.

```
// Motiviert aus
// https://docs.microsoft.com/de-de/dotnet/api/system.threading.tasks.task?view=netframework-4.8
using System;
using System.Threading.Tasks;
using System.Threading;

public class Example
{
    public static void doSomething(){
        Console.WriteLine("Say hello!");
    }

    public static void Main()
    {
        Action<object> action = (object obj) =>
        {
            Console.WriteLine("Task={0}, obj={1}, Thread={2}",
                Task.CurrentId, obj,
                Thread.CurrentThread.ManagedThreadId);
        };

        // Create a task but do not start it.
        Task t1 = new Task(action, "alpha");
        t1.Start();
        Console.WriteLine("t1 has been launched. (Main Thread={0})",
            Thread.CurrentThread.ManagedThreadId);

        // Nur der Vollständigkeit halber ...
        Task t2 = new Task(doSomething);
        t2.Start();
        Console.WriteLine("t2 has been launched. (Main Thread={0})",
            Thread.CurrentThread.ManagedThreadId);

        Task t3 = Task.Run( () => {
            // Just loop.
            int ctr = 0;
            for (ctr = 0; ctr <= 1000000; ctr++)
            {}
            Console.WriteLine("Finished {0} loop iterations",
                ctr);
        } );
        t3.Wait();
    }
}
```

Wait ermöglicht auch die Beschränkung der Wartezeit auf ein bestimmtes Zeitintervall. Die `Wait(Int32)` und `Wait(TimeSpan)` Methoden blockiert den aufrufenden Thread, bis die Aufgabe abgeschlossen ist oder ein Time-outintervall abläuft, welcher Fall zuerst eintritt.

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program {
    public static void Main(string[] args){
        // Wait on a single task with a timeout specified.
        Task taskA = Task.Run( () => Thread.Sleep(2000));
        try {
            taskA.Wait(1000);           // Wait for 1 second.
            bool completed = taskA.IsCompleted;
            Console.WriteLine("Task A completed: {0}, Status: {1}",
                completed, taskA.Status);
        }
    }
}
```

```
                completed, taskA.Status);
        if (! completed)
            Console.WriteLine("Timed out before task A completed.");
    }
    catch (AggregateException) {
        Console.WriteLine("Exception in taskA.");
    }
}
```

Für komplexe Taskstrukturen kann man diese zum Beispiel in Arrays arrangieren. Für diese Reihe von Aufgaben jeweils durch Aufrufen der `Wait` Methode zu warten wäre aufwändig und wenig praktisch. `WaitAll` schließt diese Lücke und erlaubt eine übergreifende Überwachung.

Das folgenden Beispiel werden zehn Aufgaben erstellt, die wartet, bis alle zehn abgeschlossen werden, und klicken Sie dann ihren Status angezeigt.

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program {
    public static void Main(string[] args){
        // Wait for all tasks to complete.
        Task[] tasks = new Task[10];
        for (int i = 0; i < 10; i++)
        {
            tasks[i] = Task.Run(() => Thread.Sleep(2000));
        }

        try {
            Task.WaitAll(tasks);
        }
        catch (AggregateException ae) {
            Console.WriteLine("One or more exceptions occurred: ");
            foreach (var ex in ae.Flatten().InnerExceptions)
                Console.WriteLine("    {0}", ex.Message);
        }
        Console.WriteLine("Status of completed tasks:");
        foreach (var t in tasks)
            Console.WriteLine("    Task #{0}: {1}", t.Id, t.Status);
    }
}
```

Der Kanon der Möglichkeiten wird aber deutlich erweitert, wenn ein konkreter Rückgabewert genutzt werden soll. Anstatt wie bei Threads mit einer entsprechenden “außen stehenden” Variablen zu arbeiten, wird das Ergebnis im Task-Objekt selbst gespeichert und kann dann abgerufen werden. Dieser Aspekt wird über die generische Konfiguration des Tasks abgebildet:

```
Task<int> task = Task.Run(() => {int i;  
    //... Anweisungsblock ...;  
    return i});
```

Wie ist dieser Aufruf zu verstehen? Unser Task gibt anders als bei der synchronen Abarbeitung nicht unmittelbar mit dem Ende der Bearbeitung einen Wert zurück, sondern verspricht zu einem späteren Zeitpunkt einen Wert in einem bestimmten Format zu liefern. Dank der generischen Realisierung können dies beliebige Objekte sein.

Wie aber erfolgt die Rückgabe wann?

## Asynchrone Methoden

Eine asynchrone Methode ruft einen Task auf, setzt die eigene Bearbeitung aber fort und wartet auf dessen Beendigung.

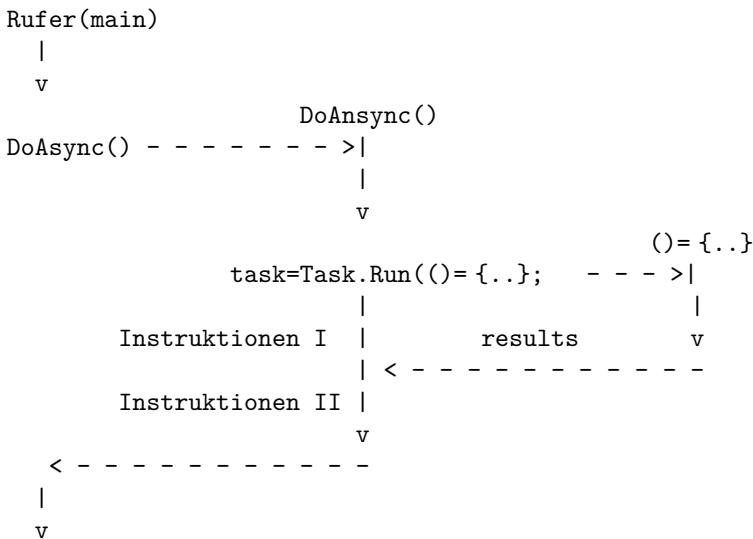
```

async void DoAsync(){
    Task<int> task = Task.Run(() => {int i;
                                         // Berechnungen
                                         return i;})
    // Instruktionen I
    // Methoden, die nach der Rückkehr nach DoAsync ausgeführt werden.
    int result = await task;
    // Instruktionen II
    // Hier wird nun mit dem Ergebnis result weitergearbeitet
}

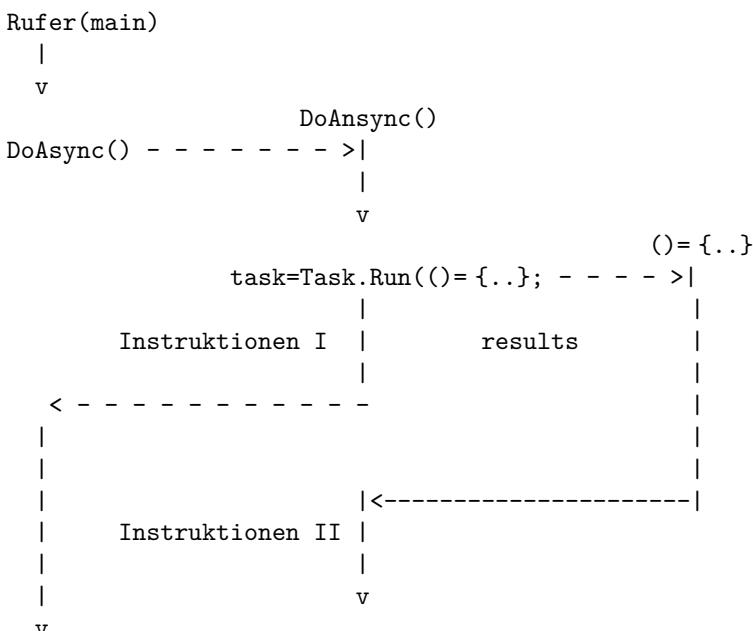
```

Das Ergebnis der Operation hängt dabei davon ab, welche Zeitalüufe sich im Programmablauf ergeben.

**Fall I** Das Ergebnis der Lambdafunktion liegt vor, bevor DoAsync die Zeile mit await erreicht hat (Quasi-Synchroner Fall)



**Fall II** Das Ergebnis der Lambdafunktion liegt erst später, nachdem DoAsync die Zeile mit await erreicht hat (und bereits nach main zurückgekehrt ist)



Zwei sehr anschauliche Beispiele finden sich im Code Ordner des Projekts.

---

Beispiel	Bemerkung
AsyncExampleI.cs	Generelle Einbettung des asynchronen Tasks

---

---

Beispiel	Bemerkung
<a href="#">AsyncExampleII.cs</a>	Illustration der Interaktionsfähigkeit eines asynchronen Programmes, das Berechnungen und Nutzereingaben gleichermaßen realisiert.

---

## Aufgaben der Woche

- [ ]



# Chapter 26

## Language-Integrated Query

---

Parameter Kursinformationen

---

**Veranstaltung:** Vorlesung Softwareentwicklung  
**Semester:** Sommersemester 2021  
**Hochschule:** Technische Universität Freiberg  
**Inhalte:** LINQ Konzepte und Anwendung  
**Link auf den** [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/25\\_LINQ.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/25_LINQ.md)  
**GitHub:**  
**Autoren** @author

---

## Motivation

Gegeben sei das Datenset eines Comic-Begeisterten in Form einer generischen Liste `List<T>`.

1. Bestimmen Sie die Zahl der Einträge unseres Datensatzes
2. Filtern Sie die Liste der Comic Figuren nach dem Alter und
3. Sortieren Sie die Liste nach dem Anfangsbuchstaben des Namens.

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

public class Character{
    protected string name;
    public int geburtsjahr;
    private static int Count;
    int index;
    public Character(string name, int geburtsjahr){
        this.name = name;
        this.geburtsjahr = geburtsjahr;
        index = Count;
        Count = Count + 1;
    }
    public override string ToString(){
        string row = string.Format("|{0,6} | {1,-15} | {2,8} |",
                                    index, name, geburtsjahr);
        return row;
    }
}

public class Program
```

```
{
    public static void Main(string[] args){
        List<Character> ComicHeros = new List<Character>{
            new Character("Spiderman", 1962),
            new Character("Donald Duck", 1931),
            new Character("Superman", 1938)
        };
        Console.WriteLine("Alle Einträge in der Datenbank:");
        Console.WriteLine("| Index | Name | Ursprung |");
        foreach (Character c in ComicHeros){
            Console.WriteLine(c);
        }
        // Und nun? Wie filtern wir?
    }
}
```

Die intuitive Lösung könnte folgendermaßen daher kommen:

Die Dokumentation von `List<T>` findet sich unter folgendem [Link](#)

1. Wir „erinnern“ uns an das `Count` Member der Klasse `List`.
2. Für die Filteroperation implementieren Sie eine Loop. Sie können dazu `foreach` verwenden, weil `List<T>` das Interface `IEnumerable` implementiert.
3. Die Sortieroperation bedingt die Anwendung einer Vergleichsoperation zwischen den Elementen der Liste. Eine Variante ist die Implementierung des Interfaces `IComparable` zu diesem Zweck.

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

public class Character: IComparable{
    protected string name;
    public int geburtsjahr;
    private static int Count;
    int index;
    public Character(string name, int geburtsjahr){
        this.name = name;
        this.geburtsjahr = geburtsjahr;
        index = Count;
        Count = Count + 1;
    }
    public override string ToString(){
        string row = string.Format("|{0,6} | {1,-15} | {2,8} |",
                                    index, name, geburtsjahr);
        return row;
    }
    public int CompareTo(object obj){
        if (obj == null) return 1;
        Character otherCharacter = obj as Character;
        return string.Compare(this.name, otherCharacter.name);
    }
}

public class Program
{
    public static void Main(string[] args){
        List<Character> ComicHeros = new List<Character>{
            new Character("Spiderman", 1962),
            new Character("Donald Duck", 1931),
            new Character("Superman", 1938)
        };
        Console.WriteLine($"\\nEinträge in der Datenbank: {ComicHeros.Count}");
        Console.WriteLine("\\nGefilterte Einträge in der Datenbank:");
    }
}
```

```

Console.WriteLine("| Index | Name | Ursprung |");
List<Character> ComicHerosFiltered = new List<Character>();
foreach (Character c in ComicHeros){
    if (c.geburtsjahr < 1950) ComicHerosFiltered.Add(c);
}
foreach (Character c in ComicHerosFiltered){
    Console.WriteLine(c);
}
Console.WriteLine("\nSortierte Einträge in der Datenbank:");
Console.WriteLine("| Index | Name | Ursprung |");
ComicHeros.Sort();
foreach (Character c in ComicHeros){
    Console.WriteLine(c);
}
}
}

```

Eine Menge Aufwand für einen simple Operation! Welche zusätzlichen Probleme werden auftreten, wenn Sie eine solche Kette aus Datenerfassung, Verarbeitung und Ausgabe in realen Anwendungen umsetzen?

Alternativ schauen wir uns weiter im Kanon der `List<T>` Klasse um und realisieren die Methoden `RemoveAll()` oder `Sort()`.

`RemoveAll()` zum Beispiel entfernt alle Elemente, die mit den Bedingungen übereinstimmen, die durch das angegebene Prädikat definiert werden. Interessant ist dabei die Umsetzung. Ein Prädikat ist eine generischer Delegat der einen Instanzen eines Typs `T` auf ein Kriterium hin evaluiert und einen Bool-Wert als Ausgabe generiert.

```

public int RemoveAll (Predicate<T> match);
public delegate bool Predicate<in T>(T obj);

```

Analog kann `Sort()` mit einem entsprechenden Delegaten `Comparison` verknüpft werden.

```

public void Sort (Comparison<T> comparison);
public delegate int Comparison<in T>(T x, T y);

using System;
using System.Collections.Generic;

```

```

public class Character: IComparable{
    protected string name;
    public int year;
    private static int Count;
    int index;
    public Character(string name, int year){
        this.name = name;
        this.year = year;
        index = Count;
        Count = Count + 1;
    }

    public override string ToString(){
        string row = string.Format("|{0,6} | {1,-15} | {2,8} |",
                                    index, name, year);
        return row;
    }

    public int CompareTo(object obj){
        if (obj == null) return 1;
        Character otherCharacter = obj as Character;
        return string.Compare(this.name, otherCharacter.name);
    }
}

```

```

}

public class Program
{
    private static bool before1950(Character entry)
    {
        return entry.year > 1950;
    }

    private static int sortByYear(Character x, Character y)
    {
        int output = 0;
        if (y.year < x.year) output = 1;
        if (y.year > x.year) output = -1;
        return output;
    }

    public static void Main(string[] args){
        List<Character> ComicHeros = new List<Character>{
            new Character("Spiderman", 1962),
            new Character("Donald Duck", 1931),
            new Character("Superman", 1938)
        };
        ComicHeros.RemoveAll(before1950);
        //ComicHeros.RemoveAll(x => x.year > 1950);
        //ComicHeros.Sort(sortByYear);
        foreach (Character c in ComicHeros){
            Console.WriteLine(c);
        }
    }
}
}

```

Allerdings bleibt die Darstellung von komplexeren Abfragen wie `filtere die Helden heraus, die vor 1950 geboren sind, extrahiere die Vornamen und sortiere diese in Aufsteigender alphabetischer Folge` zu einem unübersichtlichen Darstellungsformat.

Die Methoden für den Datenzugriff und die Manipulation abhängig vom Datentyp (Felder, Objektlisten) und der Herkunft (XML-Dokumente, Datenbanken, Excel-Dateien, usw.).

Welche alternativen Konzepte bestehen für die Verarbeitung von datengetriebenen Anfragen?

## Exkurs SQL

Hier folgt ein kurzer Einschub zum Thema *Structured Query Language* (SQL) ... um allen Teilnehmern eine sehr grundlegende Sicht zu vermitteln:

SQL ist eine Datenbanksprache zur Definition von Datenstrukturen in relationalen Datenbanken sowie zum Bearbeiten (Einfügen, Verändern, Löschen) und Abfragen von darauf basierenden Datenbeständen.

Ausgangspunkt sind Datenbanktabellen, die Abfragen dienen dabei der Generierung spezifischer Informationssets:

Buchnummer	Autor	Verlag	Datum	Titel
123456	Hans Vielschreiber	Musterverlag	2007	Wir lernen SQL
123457	J. Gutenberg	Gutenberg und Co.	1452	Drucken leicht gemacht
123458	Galileo Galilei	Inquisition International	1640	Eppur si muove
123459	Charles Darwin	Vatikan Verlag	1860	Adam und Eva

- “Alle Bücher mit Buchnummern von 123400 bis 123500”
- “Alle Buchnummern mit Autoren, die im 19. Jahrhundert erschienen.”
- “In welchem Jahrhundert veröffentlichte welcher Verlag die meisten Bücher?”
- ...

SQL basiert auf der relationalen Algebra, ihre Syntax ist relativ einfach aufgebaut und semantisch an die englische Umgangssprache angelehnt. Die Bezeichnung SQL bezieht sich auf das englische Wort "query" (deutsch: „Abfrage“). Mit Abfragen werden die in einer Datenbank gespeicherten Daten abgerufen, also dem Benutzer oder einer Anwendersoftware zur Verfügung gestellt. Durch den Einsatz von SQL strebt man die Unabhängigkeit der Anwendungen vom eingesetzten Datenbankmanagementsystem an.

SQL-Aufrufe sind deklarativ, weil der Entwickler hier nur das WAS und nicht das WIE festlegt. Dabei strukturieren sich die Befehle in 4 Kategorien:

- Befehle zur Abfrage und Aufbereitung der gesuchten Informationen
- Befehle zur Datenmanipulation (Ändern, Einfügen, Löschen)
- Befehle zur Definition des Datenbankschemas
- Befehle für die Rechteverwaltung und Transaktionskontrolle.

Eine Datenbanktabelle stellt eine Datenbank-Relation dar. Die Relation ist Namensgeber und Grundlage der relationalen Datenbanken.

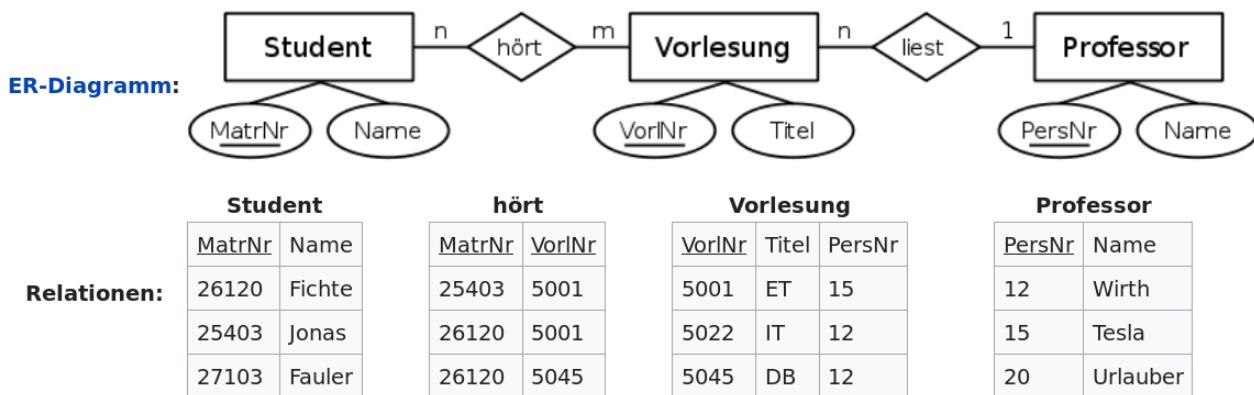


Figure 26.1: OOPGeschichte

### Erzeugung der Tabellen

```

CREATE TABLE Student;
INSERT INTO Student SELECT * from ?;

text -student.csv MatrNr,Name 26120,Fichte 25403,Jonas 27103,Fauler @AlaSQL.eval_with_csv

CREATE TABLE hoert;
INSERT INTO hoert SELECT * from ?;

text -hoert.csv MatrNr,VorlNr 26120,5001 25403,5001 27103,5045 @AlaSQL.eval_with_csv

CREATE TABLE Vorlesung;
INSERT INTO Vorlesung SELECT * from ?;

text -vorlesung.csv VorlNr,Titel,PersNr 5001,ET,15 5022,IT,12 5045,DB,12 @AlaSQL.eval_with_csv

CREATE TABLE Professor;
INSERT INTO Professor SELECT * from ?;

text -prof.csv PersNr,Name 12,Wirth 15,Tesla 20,Urlauber @AlaSQL.eval_with_csv

```

### Beispiele

```

sql      Auslesen aller Spalten und aller Zeilen SELECT * FROM Student; @AlaSQL.eval
sql      Abfrage mit Spaltenauswahl SELECT VorlNr, Titel FROM Vorlesung; @AlaSQL.eval
sql      Abfrage mit eindeutigen Werten SELECT DISTINCT MatrNr FROM hoert; @AlaSQL.eval
sql      Abfrage mit Filter und Sortierung SELECT VorlNr, Titel FROM Vorlesung WHERE Titel = 'ET'; @AlaSQL.eval

```

LIKE kann mit verschiedenen Platzhaltern verwendet werden: \_ steht für ein einzelnes beliebiges Zeichen, % steht für eine beliebige Zeichenfolge. Manche Datenbanksysteme bieten weitere solche Wildcard-Zeichen an, etwa für Zeichenmengen.

ORDER BY öffnet die Möglichkeit die Reihung anzupassen.

```
sql Verbund SELECT Vorlesung.VorlNr, Vorlesung.Titel, Professor.PersNr, Professor.Name
FROM Professor INNER JOIN Vorlesung ON Professor.PersNr = Vorlesung.PersNr; @AlaSQL.eval
```

JOIN erlaubt es die Relationen zwischen einzelnen Datenbanktabellen aufzulösen. Dabei kann mit INNER und OUTER bzw LEFT und RIGHT die Auswahl über der Schnittmenge beschrieben werden.

```
sql Gruppierung mit Aggregat-Funktionen SELECT COUNT(Vorlesung.PersNr) AS Anzahl, Professor.PersNr,
Professor.Name FROM Professor LEFT JOIN Vorlesung ON Professor.PersNr = Vorlesung.PersNr
GROUP BY Professor.Name, Professor.PersNr; @AlaSQL.eval
```

## LINQ Umsetzung

*Language Integrated Query* (LINQ) zielt auf die direkte Integration von Abfragefunktionen in die Sprache. Dafür definieren C# (wie auch VB.NET und F#) eigene Schlüsselwörter sowie eine Menge an vorbestimmten LINQ-Methoden. Diese können aber durch den Anwender in der jeweiligen Sprache erweitert werden.

```
var query =
    from e in employees
    where e.DepartmentId == 5
    select e;
```

LINQ-Anweisungen sind unmittelbar als Quelltext in .NET-Programme eingebettet. Somit kann der Code durch den Compiler auf Fehler geprüft werden. Andere Verfahren wie *ActiveX Data Objects* ADO und *Open Database Connectivity* ODBC hingegen verwenden Abfragestrings. Diese können erst zur Laufzeit interpretiert werden; dann wirken Fehler gravierender und sind schwieriger zu analysieren.

Innerhalb des Quellprogramms in C# oder VB.NET präsentiert LINQ die Abfrage-Ergebnisse als streng typisierte Aufzählungen. Somit gewährleistet es Typsicherheit bereits zur Übersetzungszeit wobei ein minimaler Codeeinsatz zur Realisierung von Filter-, Sortier- und Gruppierungsvorgänge in Datenquellen investiert wird.

Merkmale von LINQ

- Die Arbeit mit Abfrageausdrücken ist einfach, da sie viele vertraute Konstrukte der Sprache C# verwenden.
- Alle Variablen in einem Abfrageausdruck sind stark typisiert, obwohl dieser in der Regel nicht explizit angegeben wird. Der Compiler übernimmt die Ableitung.
- Eine Abfrage wird erst ausgeführt, wenn Sie über der Abfragevariable iteriert wird. Folglich muss die Quelle in einer iterierbaren Form vorliegen.
- Zur Kompilierzeit werden Abfrageausdrücke gemäß den in der C#-Spezifikation festgelegten Regeln in Methodenaufrufe des Standardabfrageoperators konvertiert. Die Abfragesyntax ist aber einfacher zu lesen.
- LINQ kombiniert Abfrageausdrücke und Methodenaufrufe (`count` oder `max`). Hierin liegt die Flexibilität des Konzeptes.

Diese Veranstaltung konzentriert sich auf die *LINQ to Objects* Realisierung von LINQ. Dabei können Abfragen mit einer beliebigen `IEnumerable`- oder `IEnumerable<T>`-Auflistungen angewandt werden.

## Exkurs “Erweiterungsmethoden”

Erweiterungsmethoden ergänzen den Umfang von bestehenden Methoden einer Klasse ohne selbst in diesem Typ deklariert worden zu sein. Man beschreibt eine statische Methode und ordnet diese einer Klasse über den Typ des ersten Parameters zu.

**Merke:** Erweiterungsmethoden stellen das bisherige Konzept der Deklaration von Klassen (etwas) auf den Kopf. Sie ermöglichen es zusätzliche Funktionalität “anzuhängen”.

Das folgende Beispiel unterstreicht den Unterschied zur bisher nur angedeuteten Methode der partiellen Methoden, die eine verteilte Implementierung einer Klasse erlaubt. Hierfür muss der Quellcode vorliegen, die Erweiterungsmethode `Print()` kann auch auf eine Bibliothek angewandt werden.

```
using System;
```

```
partial class myString
```

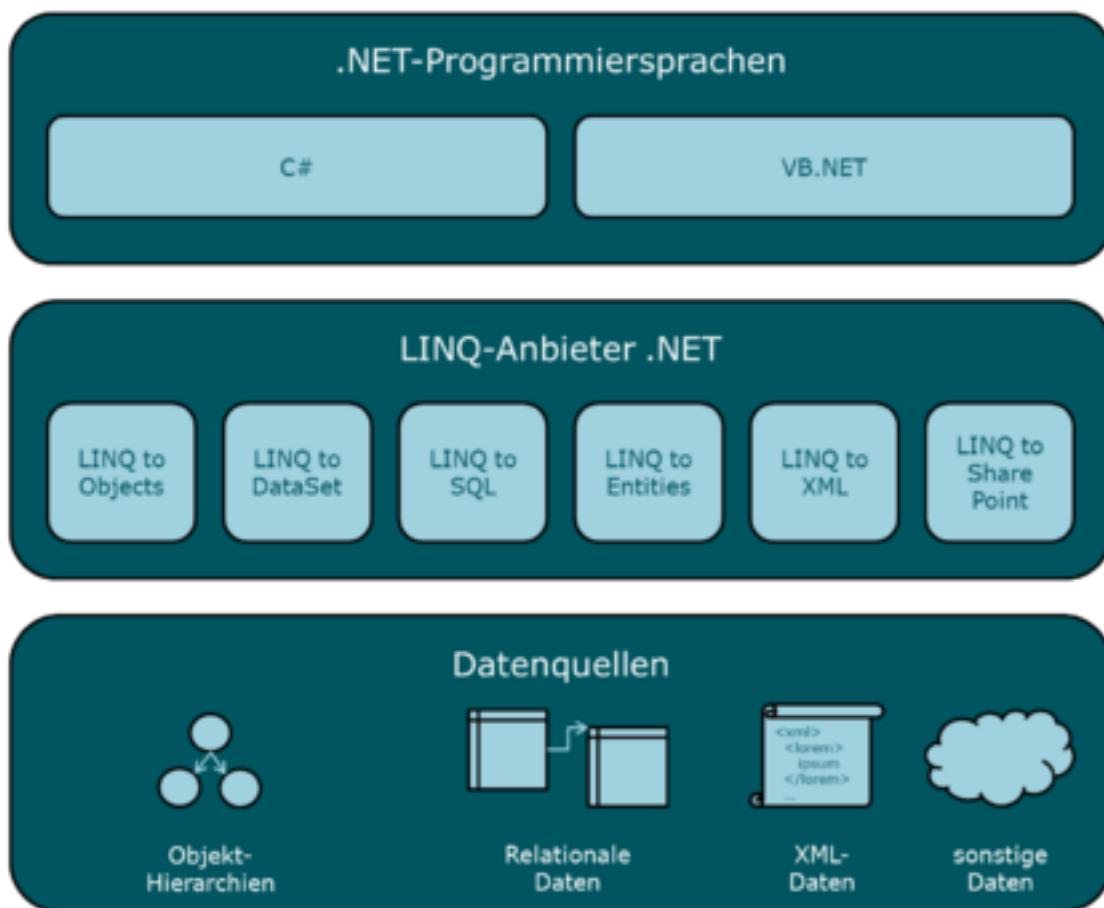


Figure 26.2: OOPGeschichte

```

{
    public string content;
    public myString(string content)
    {
        this.content = content;
    }
}

partial class myString
{
    public void sayHello() => Console.WriteLine("Say Hello!");
}

// Erweiterungsmethode in einer separaten Klasse
static class Exporter
{
    public static void Print(this myString input, string newString)
    {
        Console.WriteLine(input.content + newString);
    }
}

class Program
{
    public static void Main(string[] args)
    {
        myString text = new myString("Bla fasel");
        text.sayHello();
        text.Print("-Hossa");
    }
}

```

Erweiterungsmethoden schaffen uns die Möglichkeit weitere Funktionalität zu integrieren und gleichzeitig Datenobjekte durch eine Verarbeitungskette “hindurchzureichen”. Erweitern Sie die statische Klasse doch mal um eine Methode, die dem Inhalt der Membervariable `content` zusätzlichen Information einfügt.

Das Ganze ist natürlich noch recht behäbig, weil wir zwingend von einem bestimmten Typen ausgehen. Dies lässt sich über eine generische Implementierung lösen.

```

using System;

abstract class myAbstractString{
    public string content;
    public myAbstractString(string content)
    {
        this.content = content;
    }
    public void sayClassName() => Console.WriteLine(this.GetType().Name);
}

class myString: myAbstractString
{
    public myString(string content): base(content) {}
}

class yourString: myAbstractString
{
    public yourString(string content): base(content) {}
}

static class Exporter
{

```

```

public static void Print<T>(this T input) where T: myAbstractString
{
    Console.WriteLine(input.content);
    input.sayClassName();
}
}

class Program
{
    public static void Main(string[] args)
    {
        myString A = new myString("Bla fasel");
        A.Print();
        yourString B = new yourString("Bla blub");
        B.Print();
    }
}

```

Sie können Erweiterungsmethoden verwenden, um eine Klasse oder eine Schnittstelle zu erweitern, jedoch nicht, um sie zu überschreiben. Entsprechend wird eine Erweiterungsmethode mit dem gleichen Namen und der gleichen Signatur wie eine Schnittstellen- oder Klassenmethode nie aufgerufen.

## Exkurs “Anonyme Typen”

Anonyme Typen erlauben die Spezifikation eines Satzes von schreibgeschützten Eigenschaften, ohne zunächst explizit einen Typ definieren zu müssen. Der Typname wird dabei automatisch generiert.

Anonyme Typen enthalten mindestens eine schreibgeschützte Eigenschaft, alle anderen Arten von Klassenmembern sind ausgeschlossen.

```

using System;

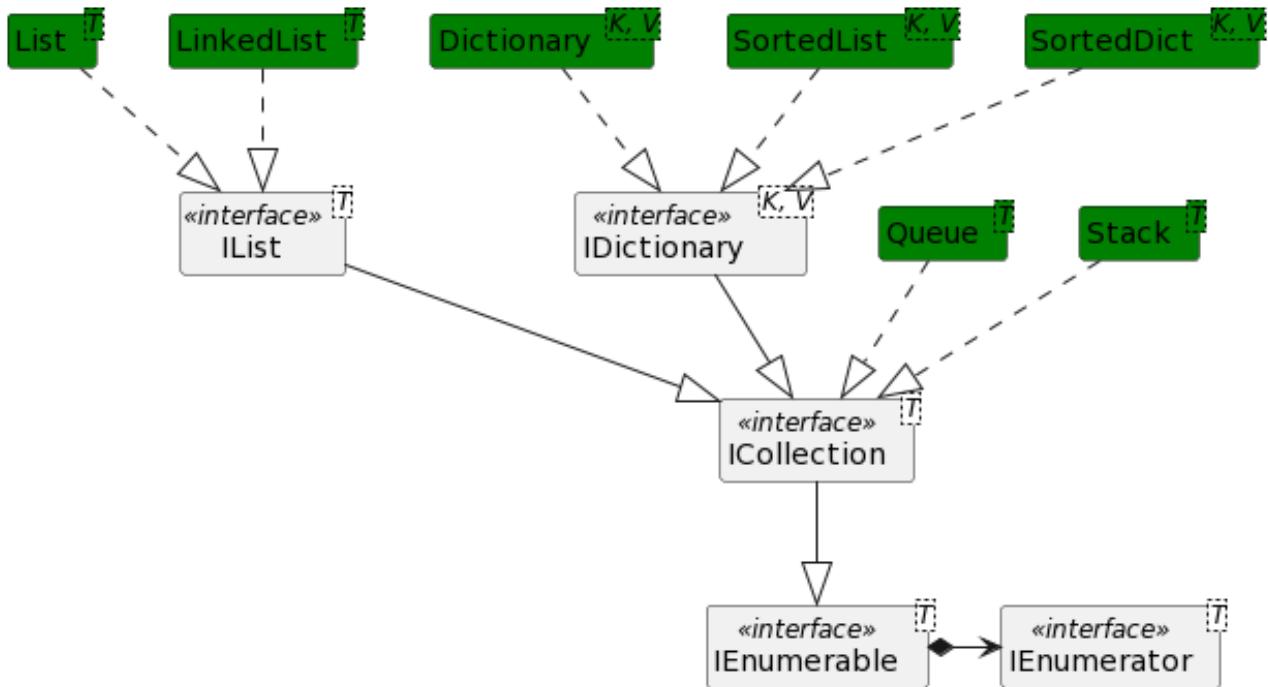
//class Irgendwas{
//    public string text;
//    public int zahl;
//}

class Program
{
    public static void Main(string[] args)
    {
        var v = new {text = "Das ist ein Text", zahl = 1};
        Console.WriteLine($"text = {v.text}, zahl = {v.zahl}");
        Console.WriteLine(v);
        //v.text = "asfsa";
        Console.WriteLine($"type = {v.GetType().Name}");
        var myPropertyInfo = v.GetType().GetProperties();
        Console.WriteLine("\nProperties:");
        for (int i = 0; i < myPropertyInfo.Length; i++)
        {
            Console.WriteLine(myPropertyInfo[i].ToString());
        }
    }
}

```

Der Vorteil alterntypen liegt in ihrer Flexibilität. Die eigentlichen Daten werden entsprechend den Ergebnissen einer Funktion erzeugt.

## Exkurs “Enumerable”



```

public interface IEnumerable<out T> : System.Collections.IEnumerable{
    public IEnumerator<T> = Get Enumerator();
}
  
```

```

public interface IEnumerator<out T> : IDisposable,
    System.Collections.IEnumerator{
    public object Current { get; }
    public bool MoveNext ();
    public void Reset ();
}
  
```

Zur Wiederholung soll nochmals ein kurzes Implementierungsbeispiel gezeigt werden. An dieser Stelle wird eine Klasse `myStrings` umgesetzt, die als Enumerationstyp realisiert werden soll. Entsprechend implementiert die Klasse `IEnumerable` das Interface `IEnumerable<string>` und referenziert einen Enumeratortyp `StringEnumerator`, der wiederum das Interface generische Interface `IEnumerator<string>` umsetzt.

Transformieren Sie folgendes Codefragment in eine UML Darstellung.

```

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

class myStrings : IEnumerable<string>{
    public string [] str_arr = new string[] {"one" , "two" , "three", "four", "five"};
    public IEnumerator<string> Get Enumerator()
    {
        I Enumerator<string> r = new StringEnumerator(this);
        return r ;
    }
    IEnumerator IEnumerable.Get Enumerator()
    {
        return GetEnumerator() ;
    }
}

class StringEnumerator : IEnumerator<string>{
    int index;
  
```

```

myStrings sp;
public StringEnumerator (myStrings str_obj){
    index = -1 ;
    sp = str_obj ;
}
object Ienumerator.Current{
    get
    { return sp.str_arr[ index ] ; }
}
public string Current{
    get
    { return sp.str_arr[ index ] ; }
}
public bool MoveNext( ){
    if ( index < sp.str_arr.Length - 1 ){
        index++ ;
        return true ;
    }
    return false ;
}
public void Reset( ){
    index = -1 ;
}
public void Dispose(){
    // pass
}
}

class Program {
    public static void Main(string[] args){
        myStrings spp = new myStrings();
        foreach( string i in spp)
            System.Console.WriteLine(i);
    }
}

```

Welchen Vorteil habe ich verglichen mit einer nicht-enumerate Datenstruktur, zum Beispiel einem array? Im Hinblick auf eine konkrete Implementierung ist zwischen dem Komfort der erweiterten API und den Performance-Eigenschaften abzuwegen.

Einen Überblick dazu bietet unter anderem die Diskussion unter <https://stackoverflow.com/questions/169973/when-should-i-use-a-list-vs-a-linkedlist/29263914#29263914>

## LINQ - Grundlagen

Sie können LINQ zur Abfrage beliebiger aufzählbarer Auflistungen wie List, Array oder Dictionary<TKey, TValue> verwenden. Die Auflistung kann entweder benutzerdefiniert sein oder von einer .NET Framework-API zurückgegeben werden.

Alle LINQ-Abfrageoperationen bestehen aus drei unterschiedlichen Aktionen:

- Abrufen der Datenquelle
- Erstellen der Abfrage
- Ausführen der Abfrage

Für ein einfaches Beispiel, das Filtern einer Liste von Zahlenwerten realisiert sich dies wie folgt:

```

using System;
using System.Threading;
using System.Collections.Generic;
using System.Linq;

```

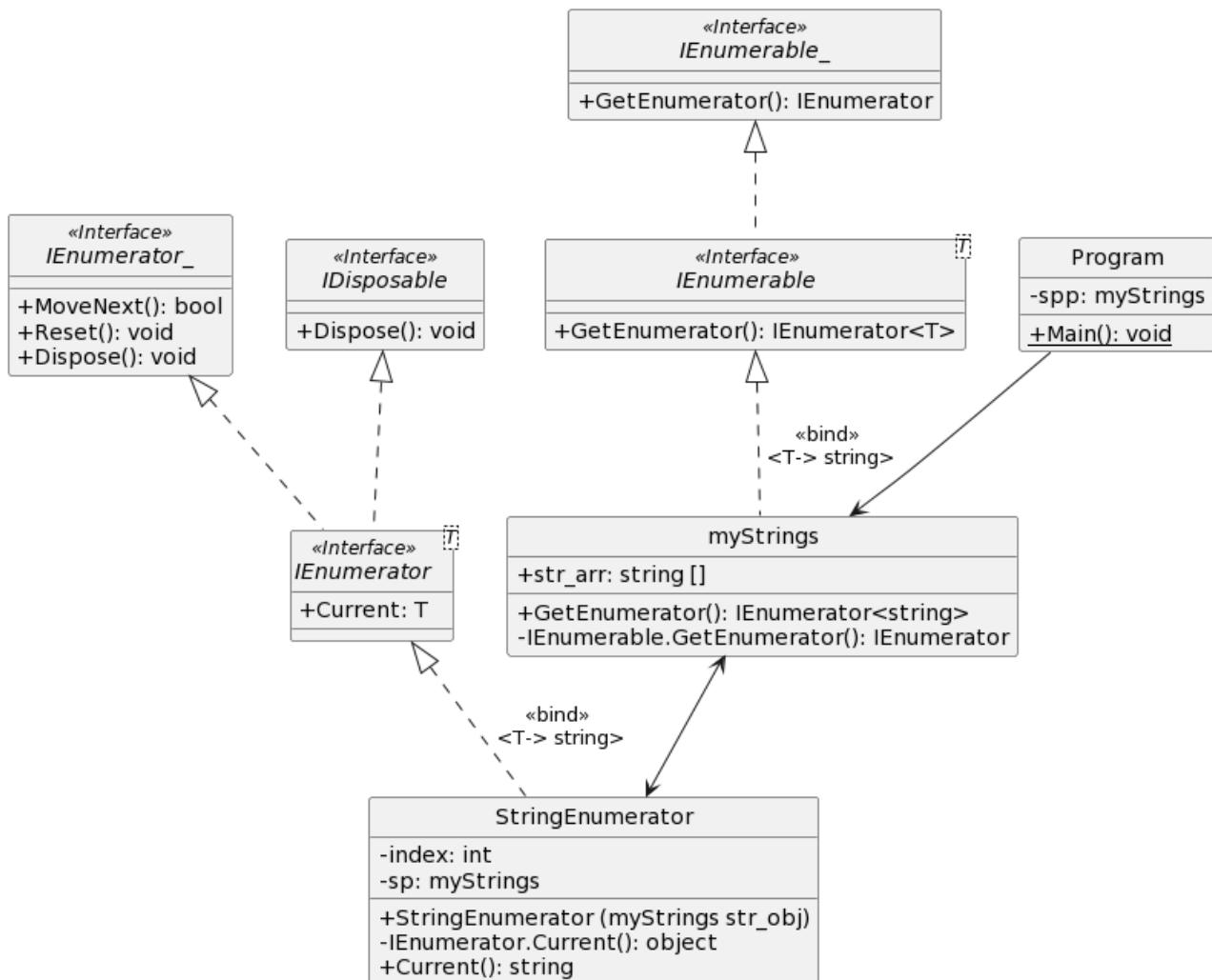


Figure 26.3: Protected

```

class Program {
    public static void Main(string[] args){
        // Spezifikation der Datenquelle
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Definition der Abfrage
        IEnumerable<int> scoreQuery =
            from score in scores      // Bezug zur Datenquelle
            where score > 80          // Filterkriterium
            select score;             // "Projektion" des Rückgabewertes

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.WriteLine(i + " ");
        }
    }
}

```

## Datenquellen

Zugriff	Bedeutung
LINQ to Objects	Zugriff auf Objektlisten und -Hierarchien im Arbeitsspeicher
LINQ to SQL	Abfrage und Bearbeitung von Daten in MS-SQL-Datenbanken
LINQ to Entities	Abfrage und Bearbeitung von Daten im relationalen Modell von ADO.NET
LINQ to XML	Zugriff auf XML-Inhalte
LINQ to DataSet	Zugriff auf ADO.NET-Datensammlungen und -Tabellen
LINQ to SharePoint	Zugriff auf SharePoint-Daten

Im Rahmen dieser Veranstaltung konzentrieren wir uns auf die LINQ to Objects Variante.

## Query Ausdrücke

Insgesamt sind 7 Query-Klauseln vorimplementiert, können aber durch Erweiterungsmethoden ergänzt werden.

Ausdruck	Bedeutung
from	definieren der Laufvariable und einer Datenquelle
where	filtert die Daten nach bestimmten Kriterien
orderby	sortiert die Elemente
select	projiziert die Laufvariable auf die Ergebnisfolge
group	bildet Gruppen innerhalb der Ergebnismenge
join	vereinigt Elemente mehrere Datenquellen
let	definiert eine Hilfsvariable

```

class Student{
    public string Name;
    public int Id;
    public string Subject{get; set;}
    public Student(){}
}

// Collection Initialization
List<Student> students = new List<Student>{
    new Student("Max Müller"){Subject = "Technische Informatik", id = 1},
    new Student("Maria Maier"){Subject = "Softwareentwicklung", id = 2},
    new Student("Martin Morawschek"){Subject = "Höhere Mathematik I", id = 3}
}

// Implizite Typdefinition
var result = from s in students           // Spezifikation der Datenquelle

```

```

        where s.Subject == "Softwareentwicklung"
        orderby s.Name
        select new (s.Name, s.Id) // Projektion der Ausgabe

// explizite Typdefinition
IEnumerable<Student> result = from s in students
    ...

```

Im vorangehenden Beispiel ist `students` die Datenquelle, über der die Abfrage bearbeitet wird. Der List-Datentyp implementiert das Interface `IEnumerable<T>`. Die letzte Zeile bildet das Ergebnis auf die Rückgabe ab, dem Interface entsprechen auf ein `IEnumerable<Student>` mit den Feldern `Name` und `Id`.

Die Berechnung der Folge wird nicht als Ganzes realisiert sondern bei einer Iteration durch den Datentyp `List<Student>`.

Für nicht-generische Typen (die also `IEnumerable` anstatt `IEnumerable<T>` unmittelbar) implementieren, muss zusätzlich der Typ der Laufvariable angegeben werden, da diese nicht aus der Datenquelle ermittelt werden kann.

```

using System;
using System.Collections.Generic;
using System.Linq;

public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int[] Scores { get; set; }
}

class Program {
    public static void Main(string[] args){
        //ArrayList StudentList = new ArrayList(); <-- Nicht mehr benutzen
        List<Student> StudentList = new List<Student>();
        StudentList.Add(
            new Student{
                FirstName = "Svetlana", LastName = "Müller", Scores = new int[] { 98, 92, 81, 60 }
            });
        StudentList.Add(
            new Student {
                FirstName = "Claire", LastName = "O'Donnell", Scores = new int[] { 75, 84, 91, 39 }
            });
        var query = from student in StudentList
                    where student.Scores[0] > 95
                    select student;
        foreach (Student s in query)
            Console.WriteLine(s.LastName + ": " + s.Scores[0]);
    }
}

```

Welche Struktur ergibt sich dabei generell für eine LINQ-Abfrage? Ein Query beginnt immer mit einer `from`-Klausel und endet mit einer `select` oder `group`-Klausel.

Allgemeingültig lässt sich, entsprechend den Ausführungen in Mössenböck folgende Syntax ableiten:

```

QueryExpr =
    "from" [Type] variable "in" SrcExpr
    QueryBody
QueryBody =
    { "from" [Type] variable "in" SrcExpr
    | "where" BoolExpr
    | "orderby" Expr ["ascending" | "descending"] {,,} Expr ["ascending" | "descending"]
    | "join" [Type] variable "in" SrcExpr "on" Expr "equals" Expr ["into" variable]

```

```

| "let" variable "==" Expr
}
( "select" ProjectionExpr ["into" variable QueryBody]
| "group" ProjectionExpr "by" Expr ["into" variable QueryBody]
).

```

Mit der isolierten Definition der Abfragen können diese mehrfach auf die Daten angewandt werden. Man spricht dabei von einer “verzögerten Ausführung” - jeder Aufruf der Ausgabe generiert eine neue Abfrage.

```

using System;
using System.Threading;
using System.Collections.Generic;
using System.Linq;

class Program
{
    public static void Main(string[] args){
        var numbers = new List<int>() {1,2,3,4};
        // Spezifikation der Anfrage
        var query = from x in numbers
                    select x;
        Console.WriteLine(query.GetType());
        // Manipulation der Daten
        numbers.Add(5);
        Console.WriteLine(query.Count());
        // Manipulation und erneute Anwendung der Abfrage
        numbers.Add(6);
        Console.WriteLine(query.Count()); // 6
    }
}

```

## Hinter den Kulissen

Der Compiler transformiert LINQ-Anfragen in der Abfragesyntax in Lambda-Ausdrücke, Erweiterungsmethoden, Objektinitializer und anonyme Typen. Dabei sprechen wir von der Methodensyntax. Abfragesyntax und Methodensyntax sind semantisch identisch, aber viele Benutzer finden die Abfragesyntax einfacher und leichter zu lesen. Da aber einige Abfragen nur in der Methodensyntax möglich sind, müssen sie diese bisweilen nutzen. Beispiele dafür sind `Max()`, `Min()`, oder `Take()`.

Nehmen wir also nochmals eine Anzahl von Studenten an, die in einer generischen Liste erfasst wurden:

```

List<Student> students = new List<Student>({
    new Student{
        Id = "123sdf234"
        FirstName = "Svetlana",
        LastName = "Omelchenko",
        Field = "Computer Science",
        Scores = new int[] { 98, 92, 81, 60 }
    };
    //...
});

var result = from s in students
            where s.Field == "Computer Science"
            orderby s.LastName
            select new {s.LastName, s.Id};

```

Der Compiler generiert daraus folgenden Code:

```

IEnumerable<Student> result = students
    .Where(s => s.Field == "Computer Science" )
    .OrderBy(s => s.LastName)
    .Select(s => new {s.LastName, s.Id});

```

Wieso hat meine Klasse `Student` plötzlich eine Methode `where?` Hier nutzen wir eine automatisch generierte Erweiterungsmethoden.

Dabei wird die eigentliche Filterfunktion als Delegat übergeben, dies wiederum kann durch eine Lambdafunktion ausgedrückt werden. <https://docs.microsoft.com/de-de/dotnet/api/system.linq.enumerable.where?view=netframework-4.8>

Dabei beschreiben die Lambdafunktionen sogenannten Prädikate, Funktionen, die eine bestimmte Bedingung prüfen und einen boolschen Wert zurückgeben.

```
using System;
using System.Threading;
using System.Collections.Generic;
using System.Linq;

class Program {
    public static bool filterme(int num){
        bool result = false;
        if (num > 10) result = true;
        return result;
    }

    public static void Main(string[] args){
        int[] numbers = { 0, 30, 20, 15, 90, 85, 40, 75 };
        //Func<int, bool> filter = delegate(int num) { return num > 10; };
        Func<int, bool> filter = filterme;
        IEnumerable<int> query =
            numbers.Where(filter);
        //IEnumerable<int> query =
        //    numbers.Where(s => s > 10);
        foreach (int number in query)
        {
            Console.WriteLine(number);
        }
    }
}
```

## Basisfunktionen von LINQ

Mit LINQ lassen sich Elementaroperationen definieren, die dann im Ganzen die Mächtigkeit des Konzeptes ausmachen.

### Filtern

Das Beispiel zur Filterung einer Customer-Tabelle wurde der C# Dokumentation unter <https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/linq/basic-linq-query-operations> entnommen.

Die üblichste Abfrageoperation ist das Anwenden eines Filters in Form eines booleschen Ausdrucks. Das Filtern bewirkt, dass im Ergebnis nur die Elemente enthalten sind, für die der Ausdruck eine wahre Aussage liefert.

Das Ergebnis wird durch Verwendung der `where`-Klausel erzeugt. Faktisch gibt der Filter an, welche Elemente nicht in die Quellsequenz eingeschlossen werden sollen. In folgendem Beispiel werden nur die customers zurückgegeben, die eine Londoner Adresse haben.

```
var queryLondonCustomers = from customer in customers
                           where customer.City == "London"
                           select customer;
```

Sie können die logischen Operatoren `&&` und `||` verwenden, um so viele Filterausdrücke wie benötigt in der `where`-Klausel anzuwenden.

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

```

using System.Linq;

class Program {
    public static bool even(int value)
    {
        return value % 2 == 0;
    }

    public static void Main(string[] args){
        var numbers = new List<int>() {-1, 7, 11, 21, 32, 42};
        var query = from i in numbers
                    where i < 40 && i > 0
                    select i;
        foreach (var x in query)
            Console.WriteLine(x);
    }
}

```

Die entsprechenden Operatoren können aber auch um eigenständige Methoden ergänzt werden. Versuchen Sie zum Beispiel die Bereichsabfrage um eine Prüfung zu erweitern, ob der Zahlenwert gerade ist.

## Gruppieren

Die group-Klausel ermöglicht es, die Ergebnisse auf der Basis eines Merkmals zusammenzufassen. Die group-Klausel gibt entsprechend eine Sequenz von `IGrouping< TKey, TElement >`-Objekten zurück, die null oder mehr Elemente enthalten, die mit dem Schlüsselwert `TKey` für die Gruppe übereinstimmen. Der Compiler leiten den Typ des Schlüssels anhand der Parameter von `group` her. `IGrouping` selbst implementiert das Interface `IEnumerable` und kann damit iteriert werden.

```

var queryCustomersByCity =
    from customer in customers
    group customer by customer.City;

// customerGroup is an IGrouping<string, Customer> now!
foreach (var customerGroup in queryCustomersByCity) // Iteration 1
{
    Console.WriteLine(customerGroup.Key);
    foreach (Customer customer in customerGroup) // Iteration 2
    {
        Console.WriteLine("    {0}", customer.Name);
    }
}

```

Dabei können die Ergebnisse einer Gruppierung wiederum Ausgangsbasis für eine weitere Abfrage sein, wenn das Resultat mit `into` in einem Zwischenergebnis gespeichert wird.

```

using System;
using System.Threading;
using System.Collections.Generic;
using System.Linq;

class Student{
    public string Name;
    public int id;
    public string Subject{get; set;}
    public Student(){}
    public Student(string name){
        this.Name = name;
    }
}

class Program {

```

```

public static void Main(string[] args){
    List<Student> students = new List<Student>{
        new Student("Max Müller"){Subject = "Technische Informatik", id = 1},
        new Student("Maria Maier"){Subject = "Softwareentwicklung", id = 2},
        new Student("Martin Morawschek"){Subject = "Höhere Mathematik I", id = 3},
        new Student("Katja Schulz"){Subject = "Technische Informatik", id = 4},
        new Student("Karl Tischer"){Subject = "Softwareentwicklung", id = 5},
    };
    var query = from s in students
                group s by s.Subject;
    foreach (var studentGroup in query)
    {
        Console.WriteLine(studentGroup.Key);
        foreach (Student student in studentGroup)
        {
            Console.WriteLine("      {0}", student.Name);
        }
    }
    var query2 = from s in students
                group s by s.Subject into sg
                select new {Subject = sg.Key, Count = sg.Count()};
    Console.WriteLine();
    foreach (var group in query2){
        Console.WriteLine(group.Count + " students attend in " + group.Subject);
    }
}
}

```

## Sortieren

Bei einem Sortiervorgang werden die Elemente einer Sequenz auf Grundlage eines oder mehrerer Attribute sortiert. Mit dem ersten Sortierkriterium wird eine primäre Sortierung der Elemente ausgeführt. Sie können die Elemente innerhalb jeder primären Sortiergruppe sortieren, indem Sie ein zweites Sortierkriterium angeben.

```

var queryLondonCustomers = from customer in customers
                           orderby customer.City, customer.Street descending
                           select customer;

using System;
using System.Threading;
using System.Collections.Generic;
using System.Linq;

class Student{
    public string Name;
    public int id;
    public string Subject{get; set;}
    public Student(){}
    public Student(string name){
        this.Name = name;
    }
}

class Program {
    public static void Main(string[] args){
        List<Student> students = new List<Student>{
            new Student("Max Müller"){Subject = "Technische Informatik", id = 1},
            new Student("Maria Maier"){Subject = "Softwareentwicklung", id = 2},
            new Student("Martin Morawschek"){Subject = "Höhere Mathematik I", id = 3},
            new Student("Katja Schulz"){Subject = "Technische Informatik", id = 4},
            new Student("Karl Tischer"){Subject = "Softwareentwicklung", id = 5},
        };
    }
}

```

```
};

var query = from s in students
            orderby s.Subject descending
            select s;
foreach (var student in query){
    Console.WriteLine("{0,-22} - {1}", student.Subject, student.Name);
}
}
```

Ausgaben

Die select-Klausel generiert aus den Ergebnissen der Abfrage das Resultat und definiert damit das Format jedes zurückgegebenen Elements. Dies kann

- den vollständigen Datensatz umfassen,
  - lediglich eine Teilmenge der Member oder
  - einen völlig neuen Datentypen.

Wenn die select-Klausel etwas anderes als eine Kopie des Quellelements erzeugt, wird dieser Vorgang als Projektion bezeichnet.

```
using System;
using System.Threading;
using System.Collections.Generic;
using System.Linq;

class Student{
    public string Name;
    public int id;
    public string Subject{get; set;}
    public Student(){}
    public Student(string name){
        this.Name = name;
    }
}

class Program {
    public static void Main(string[] args){
        List<Student> students = new List<Student>{
            new Student("Max Müller"){Subject = "Technische Informatik", id = 1},
            new Student("Maria Maier"){Subject = "Softwareentwicklung", id = 2},
            new Student("Martin Morawschek"){Subject = "Höhere Mathematik I", id = 3},
            new Student("Katja Schulz"){Subject = "Technische Informatik", id = 4},
            new Student("Karl Tischer"){Subject = "Softwareentwicklung", id = 5},
        };
        var query = from s in students
                   select new {Surname = s.Name.Split(' ')[0]};
        Console.WriteLine(query.GetType());
        foreach (var student in query){
            Console.WriteLine(student.Surname);
        }
    }
}
```

Einen guten Überblick zu den Konsequenzen einer Projektion gibt die Webseite <https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/concepts/linq/type-relationships-in-linq-query-operations>

## Aufgabe der Woche

Für die Vereinigten Staaten liegen umfangreiche Datensätze zur Namensgebung von Neugeborenen seit 1880 vor. Eine entsprechende csv-Datei (comma separated file) findet sich im Projektordner und /data, sie umfasst 258.000 Einträge. Diese sind wie folgt gegliedert

```
1880,"John",0.081541,"boy"  
1880,"William",0.080511,"boy"  
1880,"James",0.050057,"boy"
```

Die erste Spalte gibt das Geburtsjahr, die zweite den Vornamen, die Dritte den Anteil der mit diesem Vornamen benannten Kinder und die vierte das Geschlecht an.

Der Datensatz steht zum Download unter <https://osf.io/d2vyg/> bereit.

Lesen Sie aus den Daten die jeweils am häufigsten vergebenen Vornamen aus und bestimmen Sie deren Anteil innerhalb des Jahrganges.

[https://github.com/liaScript/CsharpCourse/tree/master/code/25\\_LINQII](https://github.com/liaScript/CsharpCourse/tree/master/code/25_LINQII)

Welche Möglichkeiten der Analyse sehen Sie?

# Chapter 27

## Entwurfsmuster

---

Parameter Kursinformationen

---

**Veranstaltung:** Vorlesung Softwareentwicklung  
**Semester:** Sommersemester 2021  
**Hochschule:** Technische Universität Freiberg  
**Inhalte:** Überblick Entwurfsmuster  
**Link auf den GitHub:** [https://github.com/TUBAF-IfI-LiaScript/VL\\_Softwareentwicklung/blob/master/26\\_DesignPattern.md](https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/26_DesignPattern.md)  
**Autoren:** @author

---

### Wiederholung - Polymorphie

Polymorphie ist ein Konzept in der objektorientierten Programmierung, das ermöglicht, dass ein Bezeichner abhängig von seiner Verwendung Objekte unterschiedlichen Datentyps annimmt.

Polymorphie “Vielgestaltigkeit” bezeichnet die Situation, dass ein und das gleiche “Etwas” je nach “Situation” verschiedene “Gestalt” annehmen kann.

---

Bezeichnung	Bedeutung im Kontext der Objektorierung
Etwas	Instanz einer Objektinstanz aus einer Vererbungshierarchie
Situation	Typ der Instanz beim Aufruf
Gestalt	ausgeführter Code

---

Voraussetzung ist, dass Methoden gleichen Namens für die Basisklasse und die erbende Klasse besteht. Im Beispiel ist dies `Object` und `Shape` bzw `int`.

```
class A {...}
class B: A {...}
class C: A {...}

A a = new A();
a = new B();
a = new C();

//B b = new A();    verboten
```

Wir unterscheiden entsprechend den dynamischen und den statischen Typ der Variablen `A`. Eine Referenz des Typs `A` zeigt auf eine Instanz von `B` oder `C`.

Der dynamische Typ kann zur Laufzeit geprüft (`is`) durch Cast-Operationen (`as`) angepasst werden.

Dieser Mechanismus wird beim sogenannten Boxing und Unboxing deutlich:

```

using System;

public class Shape{
    protected int m_xpos;
    protected int m_ypos;
    public Shape(){}
    public Shape(int x, int y){
        m_xpos = x;
        m_ypos = y;
    }
    public virtual void Draw(){
        Console.WriteLine("Drawing a SHAPE at {0},{1}", m_xpos, m_ypos);
    }
    public override string ToString(){
        return String.Format("Shape at [x,y] {0},{1}", m_xpos, m_ypos);
    }
}

public class Program {
    public static void Main(string[] args){
        int i = 123;
        object o = i;
        //object o = (object)i; // identisches explizites boxing
        //--- Statischer Typ von obj
        //|
        //|           +-- Dynamischer Typ von obj
        //|           /
        Object obj = new Shape();
        if (obj is Object) Console.WriteLine("Objekt Typ");
        if (obj is Shape) Console.WriteLine("Shape Typ");
    }
}

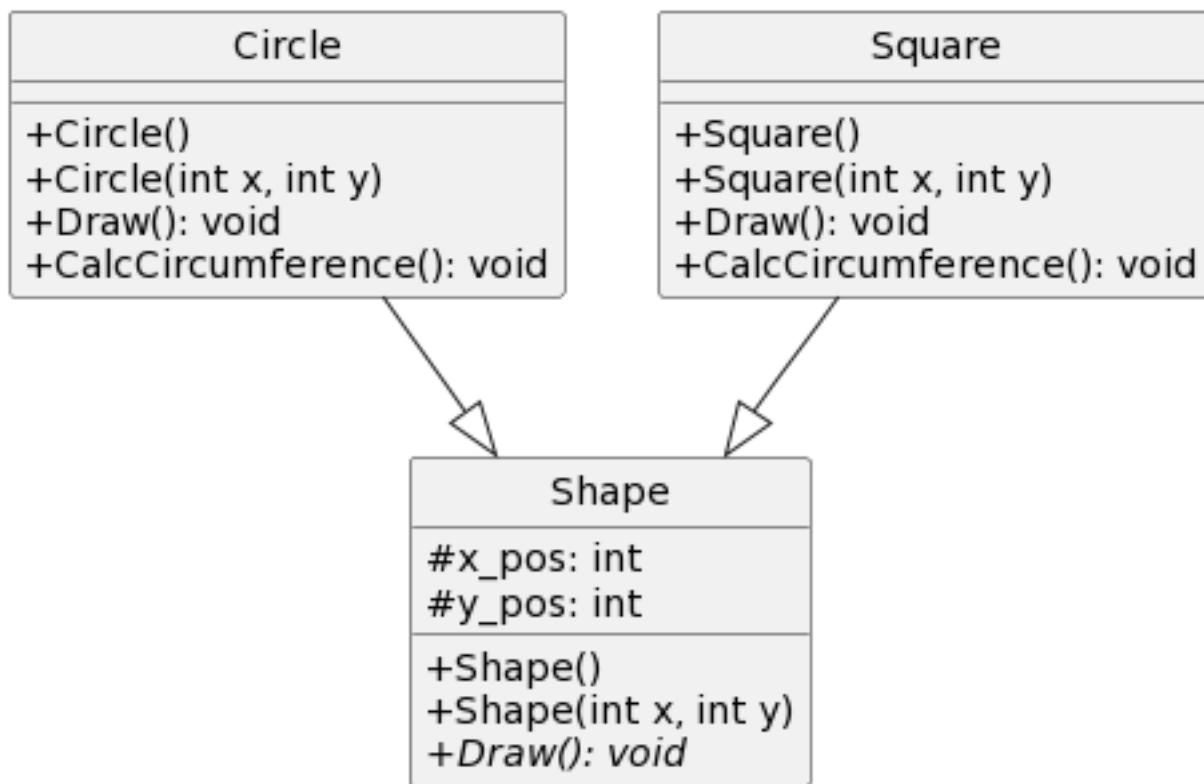
```

Merke: Boxing und Unboxing sind Cast-Operationen die eine Variable in eine Instanz von `object` konvertieren und umgekehrt.

Merke: `(Shape) obj` generiert eine Exception, wenn die Umwandlung fehlschlägt. Bei der Verwendung des Schlüsselwortes `as` in `obj as Shape` wird in diesem Fall ein `null` zurückgegeben.

*Wie wende ich das Ganze an?*

Anwendungsbeispiel:



In folgendem Beispiel wurde allein die Klasse `Circle` implementiert. Es existiert eine Methode `Draw()`, die auf der Ebene der Basisklasse und der erbenden Klasse besteht.

Merke: Polymorphie bezieht sich auf Methoden mit einer gleichen Signatur. Unterscheidet sich diese sprechen wir von Überladen. Die Signatur wird durch die Parameterzahl, -typ, art (`ref`, `out`) und die Sichtbarkeitsattribute bestimmt.

Die Implementierung in `Square` kann die aus `Shape` ignorieren, verdecken oder überschreiben.

```

using System;
using System.Collections.Generic;

public class Shape{
    public void Draw()
    {
        Console.WriteLine("Drawing a SHAPE");
    }
}

public class Square : Shape{
    public void Draw(string output)
    {
        Console.WriteLine($"Drawing a SQUARE with {output}");
    }
}

public class Program {
    public static void Main(string[] args){
        Shape sh = new Shape();
        sh.Draw();
        Square sq = new Square();
        sq.Draw("Tralla");
    }
}

<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
  
```

```
<OutputType>Exe</OutputType>
<TargetFramework>net5.0</TargetFramework>
</PropertyGroup>
</Project>
```

Erklären Sie die Bedeutung der Schlüsselworte `virtual` und `override`.

### | Virtuelle Member |

Schlüsselwort | `virtual` |

Implementierung in der Basisklasse | muss implementiert werden |

Überschreiben in der abgeleiteten Klasse | `override` |

In Bezug auf die Polymorphie bestimmen die Schlüsselworte `new` und `override` das Verhalten:

- `override` realisiert eine “angepasste” Implementierung der Methode der Basisklasse
- `new` implementiert eine völlig neue Methode, die keinen Bezug mehr zur Basisklassenfunktion hat

### | Virtuelle Member | Abstrakte Member |

Schlüsselwort | `virtual` | `abstract` |

Implementierung in der Basisklasse | muss implementiert werden | keine Implementierung |

Überschreiben in der abgeleiteten Klasse | `override` | muss überschrieben werden |

Beide Konzepte können auf Methoden (hier vorrangig betrachtet), Eigenschaften, Ereignisse und Indexer angewandt werden

Und wie war das noch mal mit den abstrakten Klassen, wie hängt deren Konzept mit abstrakten Elementen zusammen?

- Sie können keine Instanzen einer abstrakten Klasse erstellen.
- Eine abstrakte Klasse kann abstrakte oder normale, nicht abstrakte Mitglieder enthalten.
- Eine abstrakte Klasse kann selbst von einer anderen abstrakten Klasse abgeleitet werden
- Jede von einer abstrakten Klasse abgeleitete Klasse muss alle abstrakten Mitglieder der Klasse mithilfe des Schlüsselworts `override` implementieren, es sei denn, die abgeleitete Klasse ist selbst abstrakt.

## Design Pattern

Design Pattern sind spezielle Muster für Interaktionen und Zusammenhänge der Bestandteile einer Softwarelösung. Sie präsentieren Implementierungsmodelle, die für häufig wiederkehrende Abläufe (Generierung und Maskierung von Objekten) eine flexible und gut wartbare Realisierung sicherstellen. Dafür werden die Abläufe abstrahiert und auf generisch anwendbare Muster reduziert, die dann mit domänenspezifische Bezeichnern versehen nicht nur für die vereinfachte Umsetzung sondern auch für die Kommunikation dazu genutzt werden. Dies vereinfacht die Interaktion zwischen Softwarearchitekten, Programmierer und anderen Projektmitglieder.

Design Pattern sind Strukturen, Modelle, Schablonen und Muster, die sich zur Entwicklung stabiler Softwaremodelle nutzen lassen.

Entwurfsmuster für Software orientieren sich eng an den grundlegenden Prinzipien der objektorientierten Programmierung:

- Vererbung
- Kapselung
- Polymorphie

Dabei sollte ein Muster:

- ein oder mehrere Standardprobleme lösen,
- die Lesbarkeit und Wartbarkeit des Codes erhöhen
- auf die Nutzung sprachspezifischer Feature verzichten, um eine Übertragbarkeit sicherzustellen
- ein eindeutiges Set von Begriffen definieren
- Denkanstöße für den eigenen Entwurf liefern

## Kategorien

In welchen Kategorien werden Design Pattern üblicherweise strukturiert:

1. Erzeugungsmuster (englisch creational patterns)

Realisieren die Erzeugung von Objekten und entkoppeln die Konstruktion eines Objekts von seiner Repräsentation.

## 2. Strukturmuster (englisch structural patterns)

Erleichtern den Entwurf von Software durch vorgefertigte Schablonen für Beziehungen zwischen Klassen.

## 3. Verhaltensmuster (englisch behavioral patterns)

Modellieren komplexes Verhalten der Software und erhöhen damit die Flexibilität der Software hinsichtlich ihres Verhaltens.

**ACHTUNG:** Entwurfsmuster sind keine Wunderwaffe und kein Garant für gutes Design! Möglichst viele Design Pattern zu nutzen verbaut mitunter den Blick auf elegantere Lösungen.

## Erzeugungsmuster - Singleton Pattern

Das Singleton ist ein in der Softwareentwicklung eingesetztes Entwurfsmuster und gehört zur Kategorie der Erzeugungsmuster. Es stellt sicher, dass von einer Klasse genau ein Objekt existiert. Dieses Singleton ist darüber hinaus üblicherweise global verfügbar. Es soll sicher gestellt werden, dass ein Resourcenzugriff kanalisiert wird.



Figure 27.1: Singleton

## Beispiel

Ausgangspunkt der Überlegungen ist die Implementierung einer Klasse `PrinterDriver`. Über die entsprechenden Hashcodes kann gezeigt werden, dass es sich um unterschiedliche Instanzen der Klasse handelt.

Welche Möglichkeiten sehen Sie diese Implementierung anzupassen, so dass das Singleton-Pattern realisiert wird?

```

using System;

public class PrinterDriver{
    public void print(string text){
        Console.WriteLine("!PRINT {0}", text);
    }
}

public class Program {
    public static void Main(string[] args){
        // zwei Instanzen von PrinterDriver
        PrinterDriver MyPrinter = new PrinterDriver();
        PrinterDriver FaultyPrinterInstance = new PrinterDriver();
        Console.WriteLine(MyPrinter.GetHashCode());
        Console.WriteLine(FaultyPrinterInstance.GetHashCode());
    }
}
  
```

Offenbar kann der Druckertreiber mehrfach instantiiert werden. Welche Möglichkeiten sehen Sie?

\*\* Variante 1\*\*

**ACHTUNG:** Auf den ersten Blick mag die folgende Lösung plausibel erscheinen, sie hat aber einen zentralen Markel! Welche Einschränkung sehen Sie?

```

using System;

public class PrinterDriver{
    private PrinterDriver(){}
    private static PrinterDriver printerDriverInstance;

    public static PrinterDriver getInstance(){
        if (printerDriverInstance == null){
            printerDriverInstance = new PrinterDriver();
        }
        return printerDriverInstance;
    }
    public void print(string text){
        Console.WriteLine("!PRINT {0}", text);
    }
}

public class Program {
    public static void Main(string[] args){
        PrinterDriver MyPrinter = PrinterDriver.getInstance();
        PrinterDriver FaultyPrinterInstance = PrinterDriver.getInstance();
        Console.WriteLine(MyPrinter.GetHashCode());
        Console.WriteLine(FaultyPrinterInstance.GetHashCode());
    }
}

```

Von *Lazy Creation* spricht man, wenn das einzige Objekt der Klasse erst erzeugt wird, wenn es benötigt wird. Ziel ist, dass der Speicherbedarf und die Rechenzeit für die Instantiierung des Objektes nur dann aufgewendet werden, wenn das Objekt wirklich benötigt wird. Hierzu wird der Konstruktor ausschließlich beim ersten Aufruf der Funktion `getInstance()` aufgerufen.

Unter Rextester gelingt es leider nicht die Threads so zu konfigurieren, dass mehrere Instanzen der Klasse entstehen. Wenn Sie aber den nachfolgenden Code in Ihre Entwicklungsumgebung kopieren, können Sie den Effekt gut beobachten.

```

using System;
using System.Threading;

public sealed class PrinterDriver{
    private PrinterDriver(){}
    private static PrinterDriver printerDriverInstance;
    public static int InstanceCount = 0;
    public static PrinterDriver getInstance(){
        Thread.Sleep(10);
        if (printerDriverInstance == null){
            printerDriverInstance = new PrinterDriver();
            InstanceCount++;
            System.Console.WriteLine("New Driver instantiated!");
        }
        return printerDriverInstance;
    }
    public void print(string text){
        Console.WriteLine("!PRINT {0}", text);
    }
}

public class Program {
    public static void CheckInitialization() {
        PrinterDriver localInstance = PrinterDriver.getInstance();
    }
    public static void Main(string[] args){
        for (int i = 0; i < 10; i++){

```

```

        new Thread(CheckInitialization).Start();
    }
    Thread.Sleep(1000);
    Console.WriteLine("{0} Instances of PrinterDriver established!", arg0: PrinterDriver.InstanceCount)
}
}

```

Welche Lösung sehen Sie?

Als Lösungsansatz können die Synchronisationsmethoden aus der Laufzeitumgebung nutzen. `lock` garantiert, dass lediglich ein Thread einen bestimmten Codeabschnitt betreten hat und blockiert alle anderen. Eine mögliche Lösung könnte wie folgt aussehen:

```

using System;
using System.Threading;

public sealed class PrinterDriver{
    private PrinterDriver(){}
    private static PrinterDriver printerDriverInstance;
    // Zusätzliches Feld "padlock"
    private static readonly object padlock = new object();
    public static int InstanceCount = 0;
    public static PrinterDriver getInstance(){
        Thread.Sleep(100);
        lock (padlock)
        {
            if (printerDriverInstance == null){
                printerDriverInstance = new PrinterDriver();
                InstanceCount++;
                System.Console.WriteLine("New Driver instantiated!");
            }
        }
        return printerDriverInstance;
    }
    public void print(string text){
        Console.WriteLine("!PRINT {0}", text);
    }
}

public class Program {
    public static void CheckInitialization() {
        PrinterDriver localInstance = PrinterDriver.getInstance();
    }
    public static void Main(string[] args){
        for (int i = 0; i < 10; i++){
            new Thread(CheckInitialization).Start();
        }
        Thread.Sleep(1000);
        Console.WriteLine("{0} Instances of PrinterDriver established!", arg0: PrinterDriver.InstanceCount)
    }
}

```

Die einfachste Form der Realisierung kann aber mit `System.Lazy<T>` ab .NET 4 umgesetzt werden. Alles was man dabei braucht ist ein Delegate auf den Konstruktor des Singletons.

```

using System;

public sealed class PrinterDriver{
    //private static readonly Lazy<PrinterDriver> lazy = new Lazy<PrinterDriver> () => new PrinterDriver();
    private static readonly Lazy<PrinterDriver> lazy = new Lazy<PrinterDriver> (getInstance);
    static PrinterDriver getInstance(){
        return new PrinterDriver();
    }
}

```

```

public static PrinterDriver Instance {
    get { return lazy.Value; }
}
private PrinterDriver(){}
public void print(string text){
    Console.WriteLine("!PRINT {0}", text);
}
}

public class Program {
    public static void Main(string[] args){
        PrinterDriver MyPrinter = PrinterDriver.Instance;
        PrinterDriver FaultyPrinterInstance = PrinterDriver.Instance;
        Console.WriteLine(MyPrinter.GetHashCode());
        Console.WriteLine(FaultyPrinterInstance.GetHashCode());
        MyPrinter.print("Singleton - Aus die Maus");
    }
}

```

## Strukturmuster Adapter Pattern

Ausgangspunkt für das Beispiel ist die Notwendigkeit eine externes Buchungssystem mit einer Mitarbeiterdatenbank zu verknüpfen. Dabei sind Sie als Entwickler mit zwei Formen der Datenhaltung konfrontiert. Während Ihr Managementsystem für die Mitarbeiter HRSysteM auf ein Array von strings setzt, erwartet das einzubindende Buchungssystem eine (generische) Liste von strings.

```

public class HRSysteM{
    public string[][] GetEmployees(){
        string[][] employees = new string[4][];
        employees[0] = new string[] { "100", "Deepak", "Team Leader" };
        employees[1] = new string[] { "101", "Rohit", "Developer" };
        employees[2] = new string[] { "102", "Gautam", "Developer" };
        employees[3] = new string[] { "103", "Dev", "Tester" };

        return employees;
    }
}

public class ThirdPartyBillingSystem
{
    ...
    public void ShowEmployeeList(){
        List<string> employee = employeeSource.GetEmployeeList();
        ...
    }
    ...
}

```

Der Adapter (englisch adapter pattern) – auch als Wrapper bezeichnet – ist ein Entwurfsmuster das zur Übersetzung einer Schnittstelle in eine andere dient. Dadurch wird die Kommunikation von Klassen mit zueinander inkompatiblen Schnittstellen ermöglicht.

```

// Das Beispiel ist motiviert durch den Code auf der Seite
// https://www.dotnettricks.com/learn/designpatterns/adapter-design-pattern-dotnet

using System;
using System.Collections.Generic;

public interface ITarget{
    List<string> GetEmployeeList();
}

```

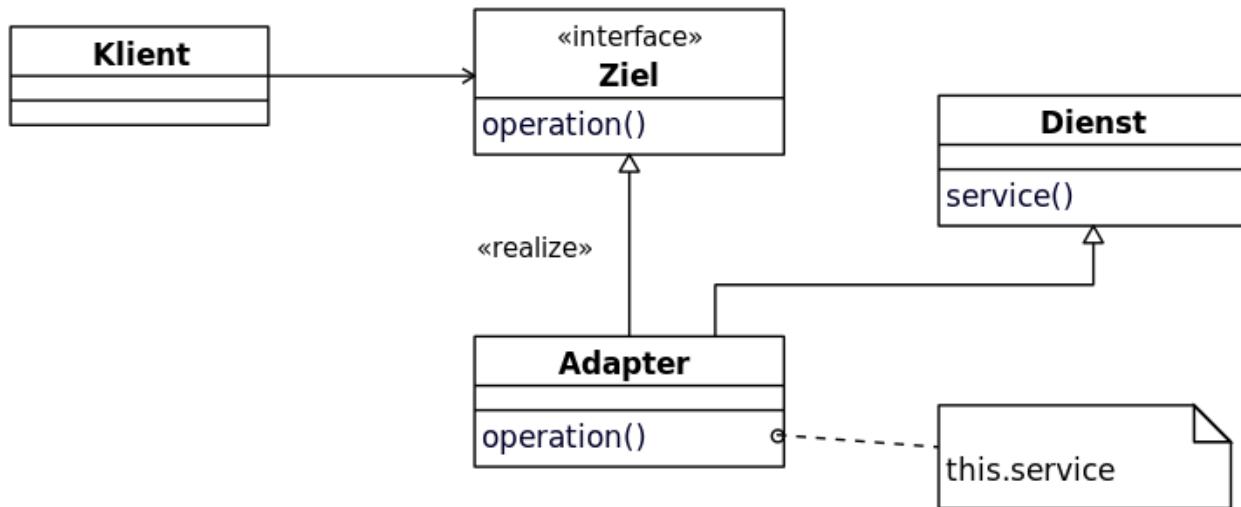


Figure 27.2: Adapter

```

public class ThirdPartyBillingSystem
{
    private ITarget employeeSource;
    public ThirdPartyBillingSystem(ITarget employeeSource){
        this.employeeSource = employeeSource;
    }
    public void ShowEmployeeList(){
        List<string> employee = employeeSource.GetEmployeeList();
        //To DO: Implement your business logic
        Console.WriteLine("##### Employee List #####");
        foreach (var item in employee){
            Console.Write(item);
        }
    }
}

public class HRSystem{
    public string[][] GetEmployees(){
        string[][] employees = new string[4] [];
        employees[0] = new string[] { "100", "Deepak", "Team Leader" };
        employees[1] = new string[] { "101", "Rohit", "Developer" };
        employees[2] = new string[] { "102", "Gautam", "Developer" };
        employees[3] = new string[] { "103", "Dev", "Tester" };
        return employees;
    }
}

public class EmployeeAdapter : HRSystem, ITarget{
    public List<string> GetEmployeeList(){
        List<string> employeeList = new List<string>();
        string[][] employees = GetEmployees();
        foreach (string[] employee in employees)
        {
            employeeList.Add(employee[0]);
            employeeList.Add(",");
            employeeList.Add(employee[1]);
            employeeList.Add(",");
            employeeList.Add(employee[2]);
            employeeList.Add("\n");
        }
    }
}
  
```

```

        return employeeList;
    }

}

public class Program {
    public static void Main(string[] args){
        ITarget Itarget = new EmployeeAdapter();
        ThirdPartyBillingSystem client = new ThirdPartyBillingSystem(Itarget);
        client.ShowEmployeeList();
    }
}

```

## Erzeugungsmuster (Abstract) Factory Pattern

Der Begriff Factory Pattern bezeichnet ein Entwurfsmuster, das beschreibt, wie ein Objekt durch Aufruf einer Methode anstatt durch direkten Aufruf eines Konstruktors erzeugt wird.

Eine abstrakte „Fabrikmethode“ dient dabei als Schnittstelle zur Erstellung eines Objektes. Die konkrete Implementierung der Erzeugung neuer Objekte findet jedoch nicht in der Oberklasse statt, sondern in von ihr abgeleiteten Unterklassen, die die besagte abstrakte Methode implementieren.

Das Muster beschreibt somit die Erzeugung von Produktobjekten, deren konkreter Typ ein Untertyp einer abstrakten Produktklasse ist, welcher von Unterklassen einer Erzeugerklasse bestimmt wird. Es wird manchmal auch als „virtueller Konstruktor“ bezeichnet.

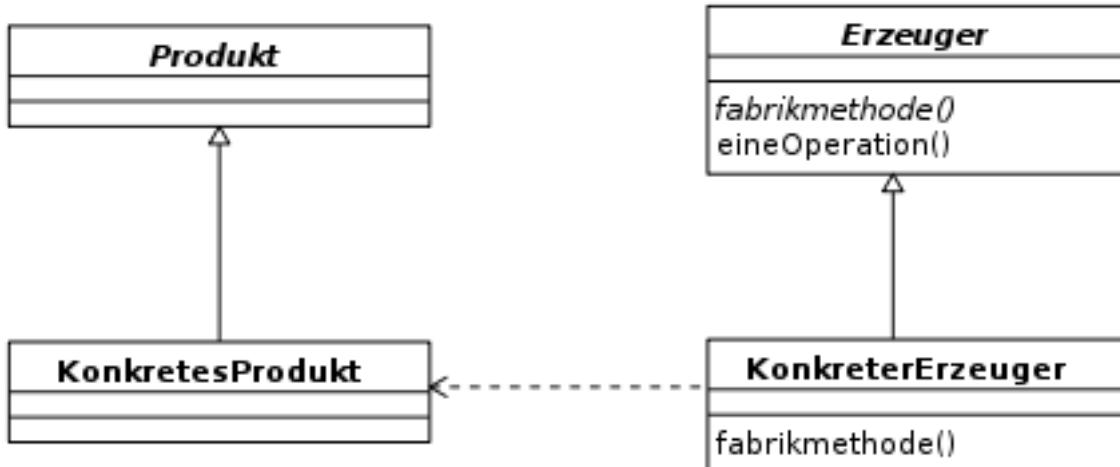


Figure 27.3: Adapter

Der Begriff Fabrikmethode wird in der Praxis auch oft einfach nur für eine statische Methode verwendet, die ein neues Objekt erzeugt, vergleichbar einem Konstruktor.

```

// SomeObject o = new SomeObject();
// Aufruf einer Fabrikmethode statt des Konstruktors
SomeObject o = SomeObjectFactory.createNewInstance();

```

Fabrikmethoden entkoppeln ihre Aufrufer von Implementierungen konkreter Produkt-Klassen:

- leichtere Anpassbarkeit bei Veränderungen
- der fest definierte Name des Konstruktors kann durch eine aussagekräftigere Methode ersetzt werden.

```
using System;
```

```

public interface IVehicle {}
public class MonoWheel : IVehicle {}
public class Car : IVehicle {}
public class Motorbike : IVehicle {}
public class Truck : IVehicle {}

```

```

public static class VehicleFactory
{
    public static IVehicle Build(int numberOfWheels)
    {
        switch (numberOfWheels)
        {
            case 1:
                return new MonoWheel();
            case 2:
            case 3:
                return new Motorbike();
            case 4:
                return new Car();
            default :
                return new Truck();
        }
    }
}

public class Program {
    public static void Main(string[] args){
        var vehicle = VehicleFactory.Build(2);
        Console.WriteLine($" You built a {vehicle.GetType().Name}");
    }
}

```

Dabei können zwei grundsätzliche Varianten unterschieden werden:

Pattern	Bedeutung
Factory pattern	Implementiert den Zugriff auf eine Familie von Produkten über einen Konkreten Erzeuger.
Abstract Factory pattern	Implementiert den Zugriff auf unterschiedliche Produktklassen über eine Familie von Erzeugern.

## Verhaltensmuster State Pattern

Die Abbildung von Zustandsmaschinen ist ein häufig wiederkehrendes Motiv. Nehmen wir an, das wir eine Rollenspielfigur modellieren wollen. Dabei bestehen lediglich drei emotionale Zustände, die Figur kann eine neutrale, eine aggressive oder eine freundliche Position einnehmen. Üblicherweise würde sich diese Einschätzung auf den Gegenüber beziehen. Bei komplexeren Sozialstrukturen müsste eine Zuordnung zu einzelnen Charakteren gewährleistet sein.

In einer tabellarischen State-Maschine Darstellung ergibt sich dann folgendes Bild:

	Happy	Neutral	Aggressive
Happy	DealingWith, Addressed		
Neutral		Addressed	DealingWith, Addresssed
Aggressive	Provoked	Provoked	Provoked

Und wie implementieren wir das Ganze? Zunächst intuitiv mittels einer einzigen Klasse und switch-case Statements.

```

using System;

public enum Feeling {Happy, Aggressive, Neutral};
public class Character{
    public byte state;
    private string name;
    public Character(string name,

```

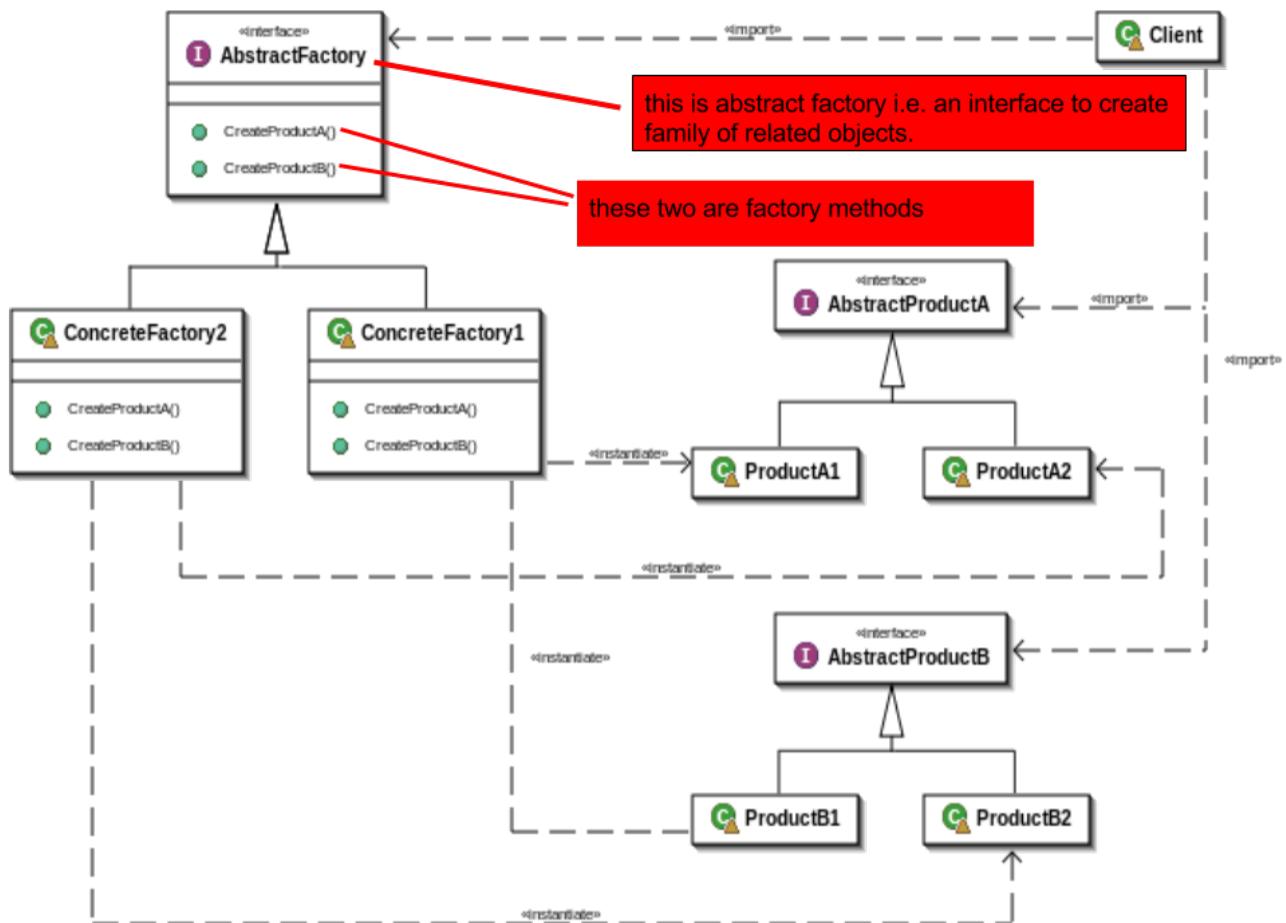


Figure 27.4: Adapter

```
        byte state =(byte)Feeling.Happy )
{
    this.name = name;
    this.state = state;
}

public void PrintState(){
    Console.WriteLine("{0} ", (Feeling)state);
}

public void Provoked(){
    switch ((Feeling)this.state)
    {
        case Feeling.Happy:
        case Feeling.Neutral:
            Console.WriteLine("{0}: Jetzt hast Du meine Laune verdorben!", name);
            this.state = (byte)Feeling.Aggressive;
            break;
        case Feeling.Aggressive:
            Console.WriteLine("{0}: Mich kannst Du nicht wütender machen", name);
            break;
        default:
            Console.WriteLine("NOT IMPLEMENTED!");
            break;
    }
    PrintState();
}

public void Addressed(){
    switch ((Feeling)this.state)
    {
        case Feeling.Happy:
        case Feeling.Neutral:
            Console.WriteLine("{0}: Dein Geschnack ändert meine Stimmung nicht", name);
            break;
        case Feeling.Aggressive:
            Console.WriteLine("{0}: Na gut, bin Dir nicht mehr böse!", name);
            this.state = (byte)Feeling.Neutral;
            break;
        default:
            Console.WriteLine("NOT IMPLEMENTED!");
            break;
    }
    PrintState();
}

public void DealingWith(){
    switch ((Feeling)this.state)
    {
        case Feeling.Happy:
            Console.WriteLine("{0}: Nett mit Dir Geschäfte zu machen!", name);
            break;
        case Feeling.Aggressive:
            Console.WriteLine("{0}: Jup, ein gutes Geschäft, fühle mich besser!", name);
            this.state = (byte)Feeling.Neutral;
            break;
        case Feeling.Neutral:
            Console.WriteLine("{0}: Super Deal, wir sind jetzt dicke Kumpels!", name);
            this.state = (byte)Feeling.Happy;
            break;
        default:
```

```

        Console.WriteLine("NOT IMPLEMENTED!");
        break;
    }
    PrintState();
}
}

public class Program {
    public static void Main(string[] args){
        Character Golum = new Character("Golum");
        Golum.Provoked();
        Golum.DealingWith();
        Golum.Addressed();
    }
}

```

Welche Probleme sehen Sie?

Wie wäre es mit folgender neuen Anforderung: Um die Modellierung "spielbar" zu sollen Wahrscheinlichkeiten beim Übergang eingeführt werden. Damit muss jeder Transition eine eigene Zahl zugeordnet werden, die vom aktuellen Zustand, der Transition und einer Zufallsvariablen abhängt.

- Unleserlichkeit und Unübersichtlichkeit.

Der Code bildet den Zustandsautomaten nicht aus Zustands- sondern aus Transitionssicht ab. Entsprechend ist das Verhalten für jeden Zustand über die Methoden `DealingWith`, `Addressed` und `Provoked` verteilt. Man stelle sich nur den Entwurf mit 10 oder mehr Zuständen und 15 Methoden vor.

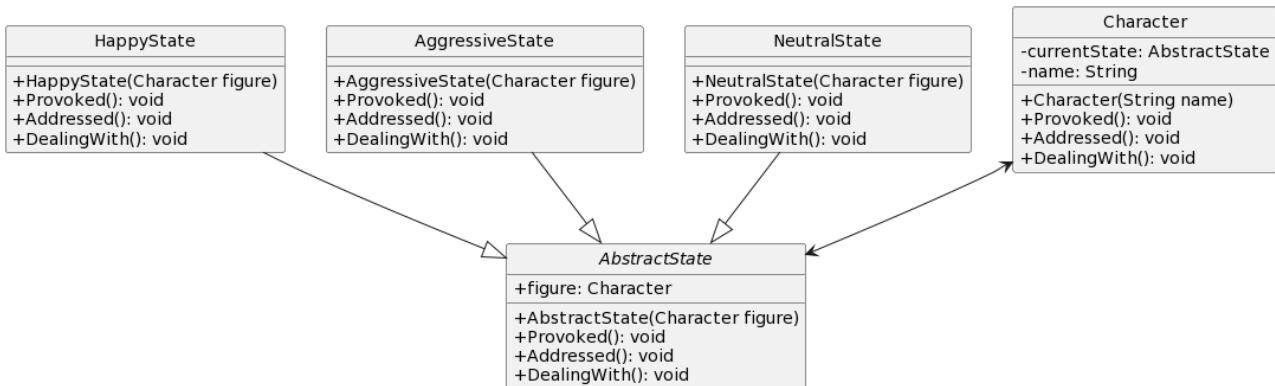
- Schlechte Wartbarkeit und Erweiterbarkeit.

Sollen neue Zustände eingeführt werden, so muss umständlich JEDER Operation um einen weiteren Fall erweitert werden. Die Fehlerträchtigkeit hierbei ist enorm.

- In jedem Zustand sind alle Transitionsfunktionen möglich

Wenn wir annehmen, dass nur im `Feeling.Happy` Fall eine bestimmte Interaktion stattfinden kann, sollten wir in allen anderen Zuständen deren Aufruf auch nicht ermöglichen.

Der State Pattern ist ein Entwurfsmuster, das zur Kapselung unterschiedlicher, zustandsabhängiger Verhaltensweisen eines Objektes eingesetzt. Grundsätzlich gilt, dass das Verhalten eines Objekts abhängig von seinem Zustand dargestellt wird. Entsprechend wird hier jeder Fall der switch-Anweisung in einer eigenen Klasse implementiert, so dass der Zustand des Objektes selbst wieder ein Objekt ist, das unabhängig von anderen Objekten ist. Gleichzeitig realisieren wir eine Abstraktionsebene durch ein Interface oder eine Basisklasse. Gegen diese wird die Anwendung, hier die Klasse `Character` entwickelt.



```

using System;

public abstract class AbstractState{
    public Character figure;
    public AbstractState(Character figure){
        this.figure = figure;
    }
}

```

```

    }
    public abstract void Provoked();
    public abstract void Addressed();
    public abstract void DealingWith();
}

public class HappyState : AbstractState{
    public HappyState(Character figure): base(figure) {}
    public override void Provoked(){
        Console.WriteLine("{0} is Happy but switches to Aggressive", figure.name);
        figure.setState(new AggressiveState(figure));
    }

    public override void Addressed(){
        Console.WriteLine("{0} is Happy", figure.name);
    }

    public override void DealingWith(){
        Console.WriteLine("{0} is DealingWith", figure.name);
    }
}

public class AggressiveState : AbstractState{
    public AggressiveState(Character figure): base(figure) {}
    public override void Provoked(){
        Console.WriteLine("{0} is Aggressive", figure.name);
    }

    public override void Addressed(){
        Console.WriteLine("{0} is Aggressive but switches to Neutral", figure.name);
        figure.setState(new NeutralState(figure));
    }

    public override void DealingWith(){
        Console.WriteLine("{0} is Aggressive but switches to Neutral", figure.name);
        figure.setState(new NeutralState(figure));
    }
}

public class NeutralState : AbstractState{
    public NeutralState(Character figure): base(figure) {}
    public override void Provoked(){
        Console.WriteLine("{0} is Neutral but switches to Aggressive", figure.name);
        figure.setState(new AggressiveState(figure));
    }

    public override void Addressed(){
        Console.WriteLine("{0} is Neutral", figure.name);
    }

    public override void DealingWith(){
        Console.WriteLine("{0} is Aggressive but switches to Neutral", figure.name);
        figure.setState(new HappyState(figure));
    }
}

public class Character{
    private AbstractState currentState;
    public string name;
    public Character(string name){
        this.name = name;
    }
}

```

```

        currentState = new HappyState(this);
    }

    public void setState(AbstractState newState){
        currentState = newState;
    }

    public void Addressed(){
        currentState.Addressed();
    }

    public void Provoked(){
        currentState.Provoked();
    }

    public void DealingWith(){
        currentState.DealingWith();
    }
}

public class Program {
    public static void Main(string[] args){
        Character Golum = new Character("Golum");
        Golum.Provoked();
        Golum.DealingWith();
    }
}

```

1. Was müssen Sie tun, um einen weitere Zustand hinzuzufügen?
2. Wie können wir eine zusätzliche Transition integrieren?
3. Wie lassen sich Methoden einbetten, die nur von bestimmten Zuständen realisiert werden?

## Aufgabe der Woche

Bitte geben Sie uns ein Feedback!

<https://panel.ovgu.de/s/c9845e6d/de.html>

## Ausblick

Und nun ... Wie geht es weiter?

Im nächsten Semester mit der Veranstaltung Robotikprojekt bestehend aus zwei Teilen:

- theoretische Vorbereitung im Wintersemester (abstrakte Einführung in C++ mit der Perspektive Softwareentwurf, Einführung in ROS, Robotikanwendungen)
- praktische Umsetzung im Sommersemester 2022



Figure 27.5: Husky