

# Organisatorisches

## **Lernziele I**

Die Studierenden kennen die Methoden der objektorientierten Programmierung und können diese anwenden.

Sie sind in der Lage, mittelgrosse, vollständig objektorientierte, grafische Anwendungen zu implementieren, testen und dokumentieren.

## Lernziele II

### Die Studierenden

- kennen die Konzepte Kapselung, Vererbung, Polymorphie, dynamisches Binden, abstrakte Klassen und generische Programmierung und können diese in einfachen Beispielen anwenden.
- können den Kontrollfluss eines Programmes mit Ausnahmebehandlung verstehen und die Vorteile erläutern.
- kennen die SOLID - Prinzipien und können sie in eigenen Worten erklären.
- kennen die verschiedenen Testarten und können einfache Unit-Tests selber schreiben.

## Lernziele II

### Die Studierenden

- kennen das Vorgehen beim Test Driven Development und kennen die Bedeutung von Refactoring und Testing als integralen Teil der Softwareentwicklung.
- kennen das Vorgehen sowie Vor- und Nachteile des Pair-Programming.
- können eine GUI-Applikation entwickeln. Sie können dabei gängige objektorientierte Konzepte anwenden und den Code sinnvoll strukturieren.
- wissen, worauf sie bei der Auswahl eines Frameworks achten müssen.

# Unterlagen

- [github.com/fhirter](https://github.com/fhirter)
- Literatur.pdf
- OneNote Klassennotizbuch

## Ratschlag

- Wenn du etwas nicht verstehst, frage! Dumme Fragen sind nur die, die nicht gestellt werden.

# Einstieg

# **Design**

"Any sufficiently advanced technology is indistinguishable from magic."

-- Arthur C. Clarke

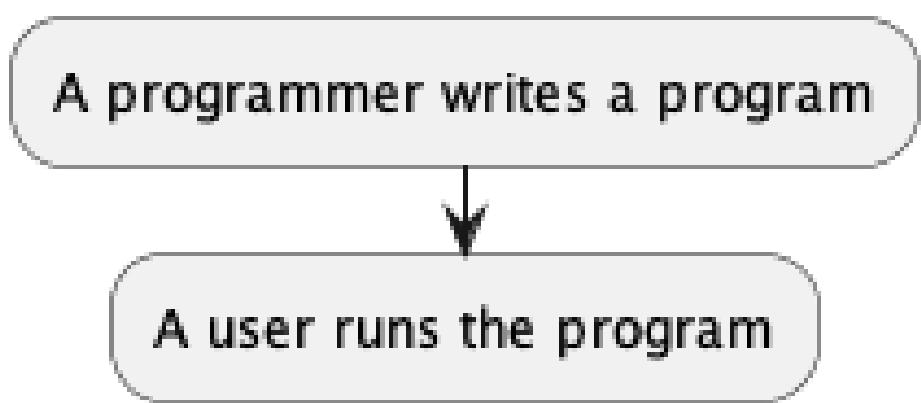
## **Softwareentwickler:innen bauen Maschinen**

- Unsere Maschinen können nicht angefasst werden: Sie sind nicht materiell
- Wir sprechen von Programmen oder Systemen (Software)
- Um eine Softwaremaschine laufen zu lassen brauchen sie eine physische Maschine: den Computer (Hardware)

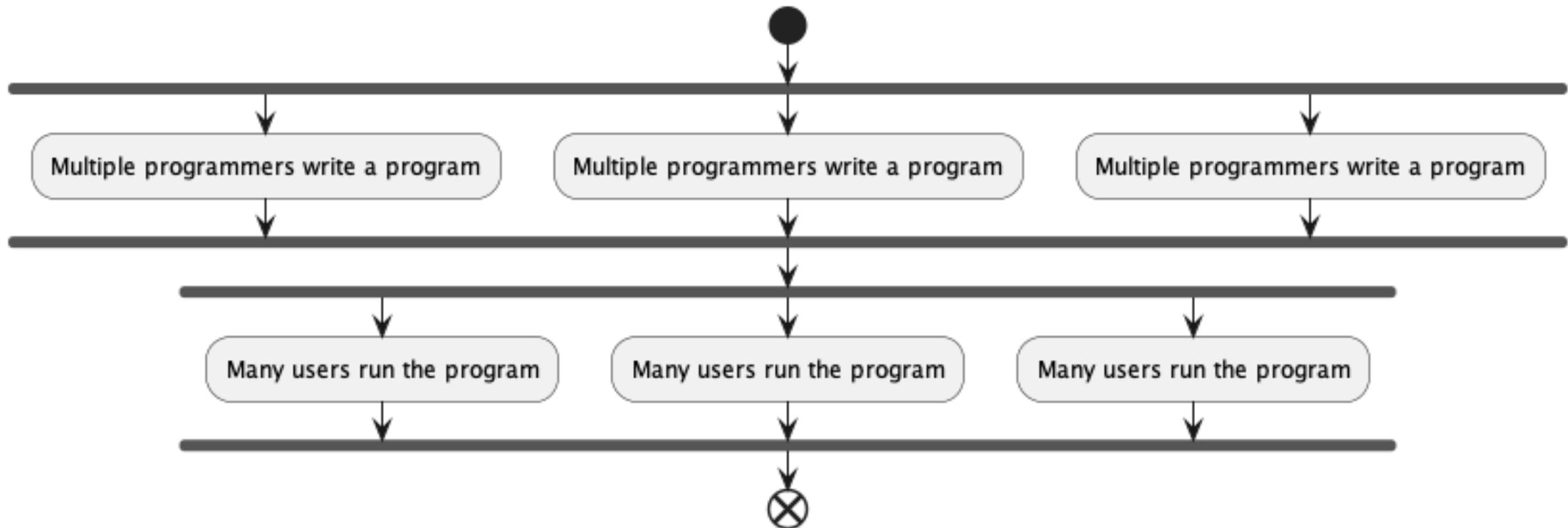
# Computer

- Computer sind universelle Maschinen. Sie führen die Programme aus, die wir ihnen füttern.
- Die einzigen Grenzen sind unsere Vorstellungskraft
- Gute Nachricht
  - Dein Computer macht genau das, was man ihm sagt.
  - Er macht es sehr schnell.
- Schlechte Nachricht
  - Dein Computer macht genau das, was man ihm sagt.
  - Er macht es sehr schnell.

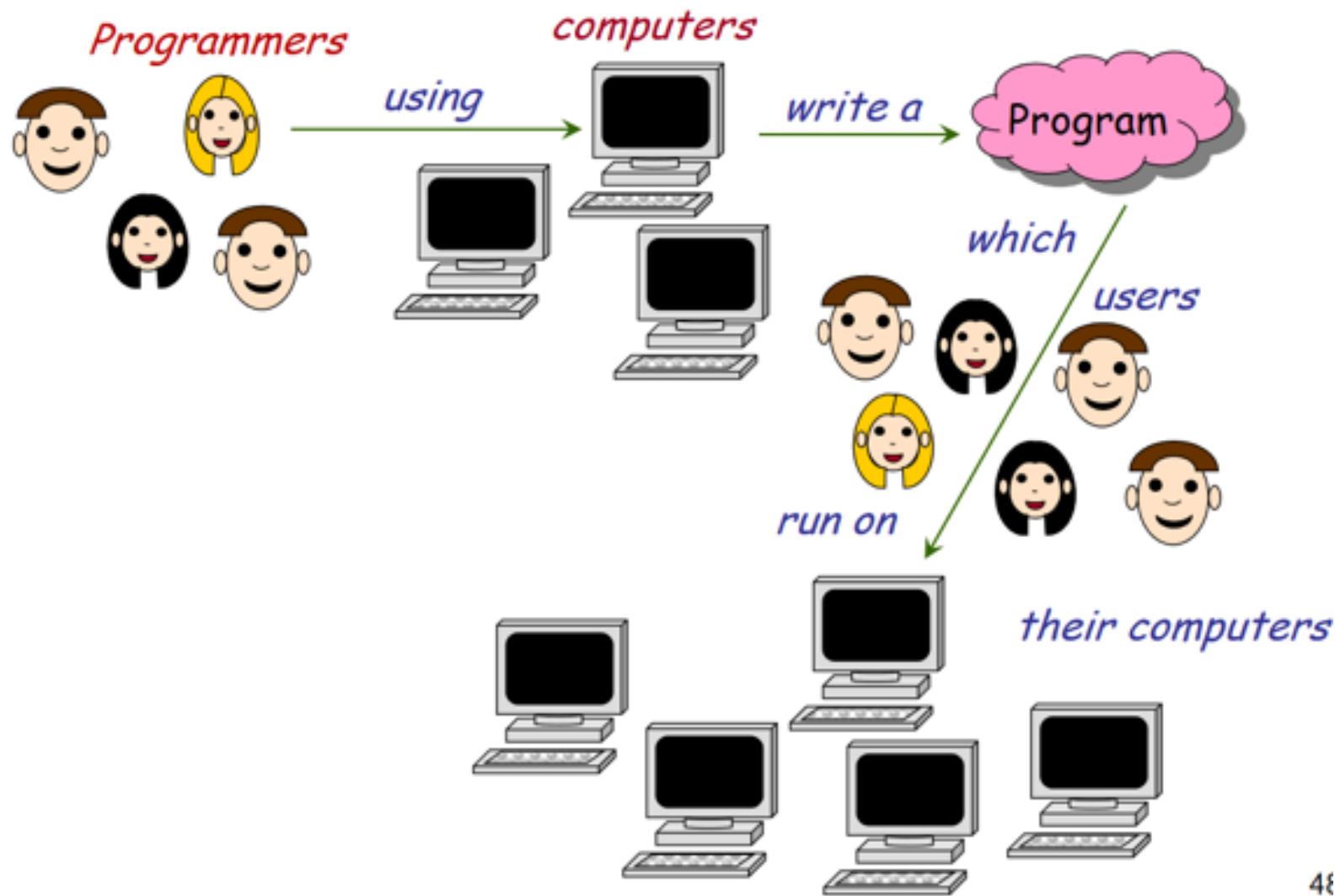
# Programme erstellen und laufen lassen



# Programme erstellen und laufen lassen



# Programme erstellen und laufen lassen



# Verbreitete Mythen und Entschuldigungen

- «Computer sind intelligent»
  - Fakt: Computer sind weder intelligent noch dumm. Sie führen Programme aus, die von Menschen entwickelt wurden.
  - Diese Programme bilden die Intelligenz ihrer Autoren ab.
  - Die grundlegenden Computeroperationen sind elementar (Speichere diesen Wert, Addiere diese beiden Zahlen...)
- «Der Computer ist abgestürzt»
- «Der Computer erlaubt das nicht»
- «Der Computer hat ihren Datensatz verloren»
- [how to never write bug - FireShip.io](#)

# Software schreiben ist herausfordernd

- Programme können «abstürzen»
- Programme, die nicht «abstürzen» funktionieren nicht zwangsläufig richtig.
- Fehlerhafte Programme können Menschen töten, (medizinische Geräte, Luftfahrt) → Boeing 737 MCAS
- Ariane5 Rakete, 1996: \$10 Milliarden verloren aufgrund eines Programmfehlers.
- Programmierer sind verantwortlich für das korrekte Funktionieren der Programme
- Das Ziel dieses Fachs ist, nicht nur programmieren zu lernen, sondern gut programmieren zu lernen.

# Grundsätzliche Organisation

# Computer

- Computer sind universelle Maschinen, sie führen Programme aus, die wir ihnen "füttern"

# Informationen und Daten

- Information ist das, was wir Menschen wollen und verstehen, z.B. ein Lied oder einen Text
  - Interpretation von Daten für Menschen
- Daten bezeichnet, wie dies im Computer gespeichert wird, z.B. als MP3 Datei.
  - Ansammlung von Symbolen in einem Computer

## Informationen und Daten

- Daten werden gespeichert
- Eingabegeräte produzieren Daten aus Informationen
- Ausgabegeräte produzieren Informationen aus Daten

# Wo ist das Programm?

- Stored-program computer: Das Programm ist im Speicher
  - „ausführbare Daten“
- Ein Programm kann in verschiedenen Formen im Speicher auftreten:
  - Quellcode / Sourcecode: durch Menschen lesbare Form (Programmiersprache)
  - Maschinencode: Ausführbar durch Computer
- Compiler / Interpreter transformieren von Sourcecode zu Maschinencode
- Der Computer findet das Programm im Speicher und führt es aus.

# Software Engineering

Software sollte folgende Merkmale haben:

- Korrekt: Machen, was es sollte!
- Erweiterbar: Einfach zu ändern sein!
- Lesbar: durch Menschen!
- Wiederverwertbar: Das Rad nicht neu erfinden!
- Robust: Korrekt auf Fehler reagieren!
- Sicher: Angreifer abwehren!

## **Software schreiben ist herausfordernd**

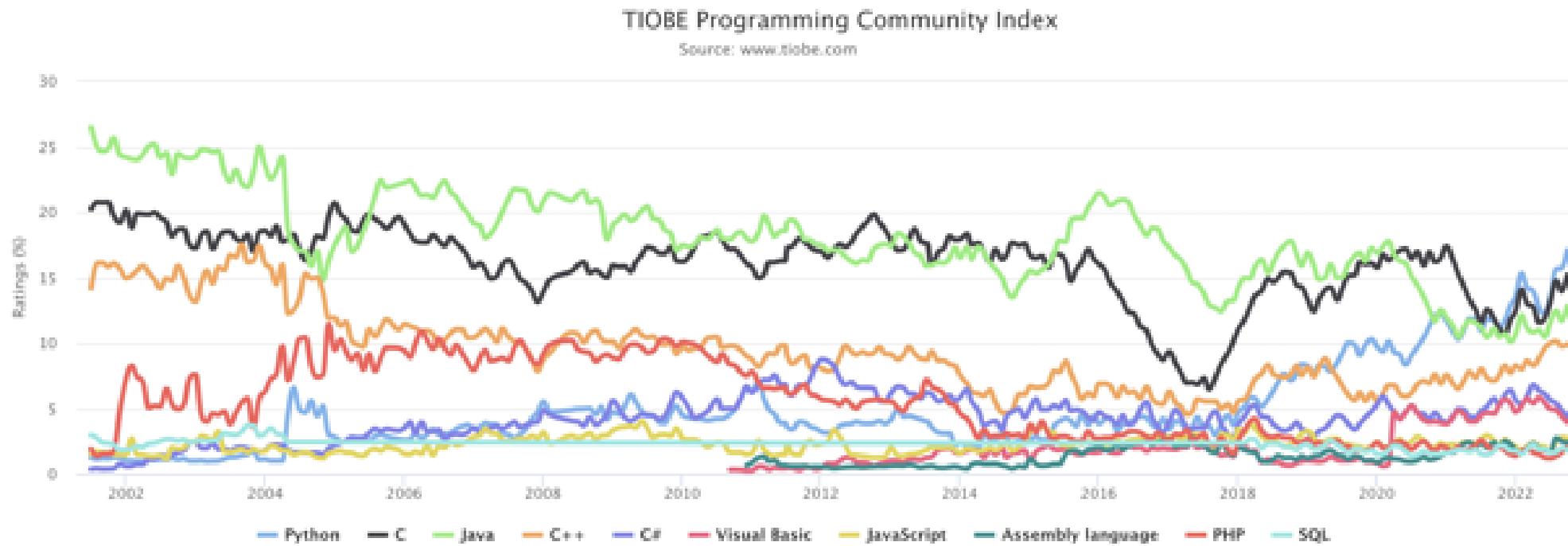
- Es ist schwierig, das Programm korrekt zu schreiben
- Trial-and-error ist ineffizient

## **Software schreiben macht Spass**

- Entwickle deine eigene Maschine!
- Kreativität und Vorstellungsvermögen kann ausgelebt werden!
- Programme retten leben und machen die Welt besser!

# Entwicklungswerkzeuge

# Programmiersprachen



Tiobe Index

God-Tier Developer Roadmap

# C

- 1972, Dennis Ritchie, Bell Labs
- Kompiliert
- Imperativ, Strukturiert
- statisch Typisiert
- Grosse Verbreitung in Betriebssystemen und Embedded
- Sehr schnell und effizient

# C++

- 1985, Bjarne Stroustrup, Bell Labs
- Kompiliert
- Objektorientiert
- Erweiterung von C
- Schnell und effizient
- Hochkomplex
- Grosse Verbreitung in Betriebssystemen, Desktop Applikationen, Games, Datenbanken, Interpreter

# Java

- 1995, James Gosling, Sun Microsystems
- Kompiliert / Virtuelle Maschine (Plattformunabhängigkeit)
- Objektorientiert
- statisch Typisiert
- Grosses Angebot an Bibliotheken und Werkzeugen
- Einfache Syntax

# C#

- 2000, Anders Hejlsberg, Microsoft
- Kompiliert
- Objektorientiert
- statisch Typisiert
- Game Entwicklung (Unity), Microsoft Ökosystem

# Python

- 1991, Guido van Rossum, Centrum Wiskunde & Informatica
- Interpretiert
- Objektorientiert
- dynamische Typisierung
- Einfache Syntax, schlanke Programme, wenig Ballast
- Grosses Angebot an Bibliotheken und Werkzeugen

# PHP

- 1995, Rasmus Lerdorf
- Interpretiert
- Objektorientiert
- dynamische Typisierung
- Im Web weit verbreitet (Backend)

# JavaScript / TypeScript

- 1995, Brendan Eich, Netscape
- Interpretiert
- Objektorientiert (Prototypenbasiert)
- dynamische Typisierung
- statische Typisierung mit TypeScript, 2014, Microsoft
- Hohe Verbreitung im Web (Frontend und Backend)

# Rust

- 2015, Graydon Hoare, Mozilla
- Kompiliert
- Objektorientiert, nebenläufig
- statische Typisierung
- Keine Garbage Collection
- Sicher, Nebenläufig
- Seit 2022 im Linux Kernel verwendet

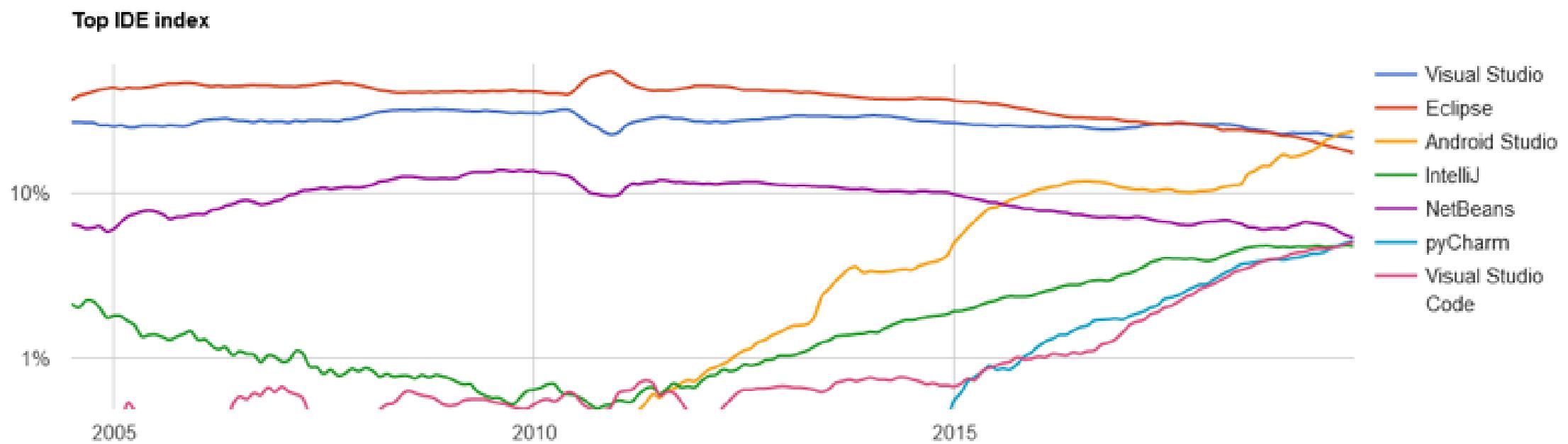
# Go

- 2012, Rob Pike / Ken Thompson / Robert Griesemer, Google
- Kompiliert
- Objektorientiert, nebenläufig
- statische Typisierung
- Keine Vererbung
- Effizienz, Lesbarkeit / DX, Networking, Multiprocessing

# Energy, Time, Memory Comparison

Total				
	Energy	Time	Mb	
(c) C	1.00	1.00	(c) Pascal	1.00
(c) Rust	1.03	1.04	(c) Go	1.05
(c) C++	1.34	1.56	(c) C	1.17
(c) Ada	1.70	1.85	(c) Fortran	1.24
(v) Java	1.98	1.89	(c) C++	1.34
(c) Pascal	2.14	2.14	(c) Ada	1.47
(c) Chapel	2.18	2.83	(c) Rust	1.54
(v) Lisp	2.27	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	3.09	(c) Haskell	2.45
(c) Fortran	2.52	3.14	(i) PHP	2.57
(c) Swift	2.79	3.40	(c) Swift	2.71
(c) Haskell	3.10	3.55	(i) Python	2.80
(v) C#	3.14	4.20	(c) Ocaml	2.82
(c) Go	3.23	4.20	(v) C#	2.85
(i) Dart	3.83	6.30	(i) Hack	3.34
(v) F#	4.13	6.52	(v) Racket	3.52
(i) JavaScript	4.45	6.67	(i) Ruby	3.97
(v) Racket	7.91	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	26.99	(v) F#	4.25
(i) Hack	24.02	27.64	(i) JavaScript	4.59
(i) PHP	29.30	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	43.44	(v) Java	6.01
(i) Lua	45.98	46.20	(i) Perl	6.62
(i) Jruby	46.54	59.34	(i) Lua	6.72
(i) Ruby	69.91	65.79	(v) Erlang	7.20
(i) Python	75.88	71.90	(i) Dart	8.64
(i) Perl	79.58	82.91	(i) Jruby	19.84

# Entwicklungsumgebungen



# Entwicklungsumgebungen

## Eclipse

- JavaScript/TypeScript, C/C++, PHP, Rust etc
- Open Source

## Microsoft Visual Studio

- VB, C, C++, C##, SQL, TypeScript, Python, HTML, JavaScript, CSS
- Closed Source

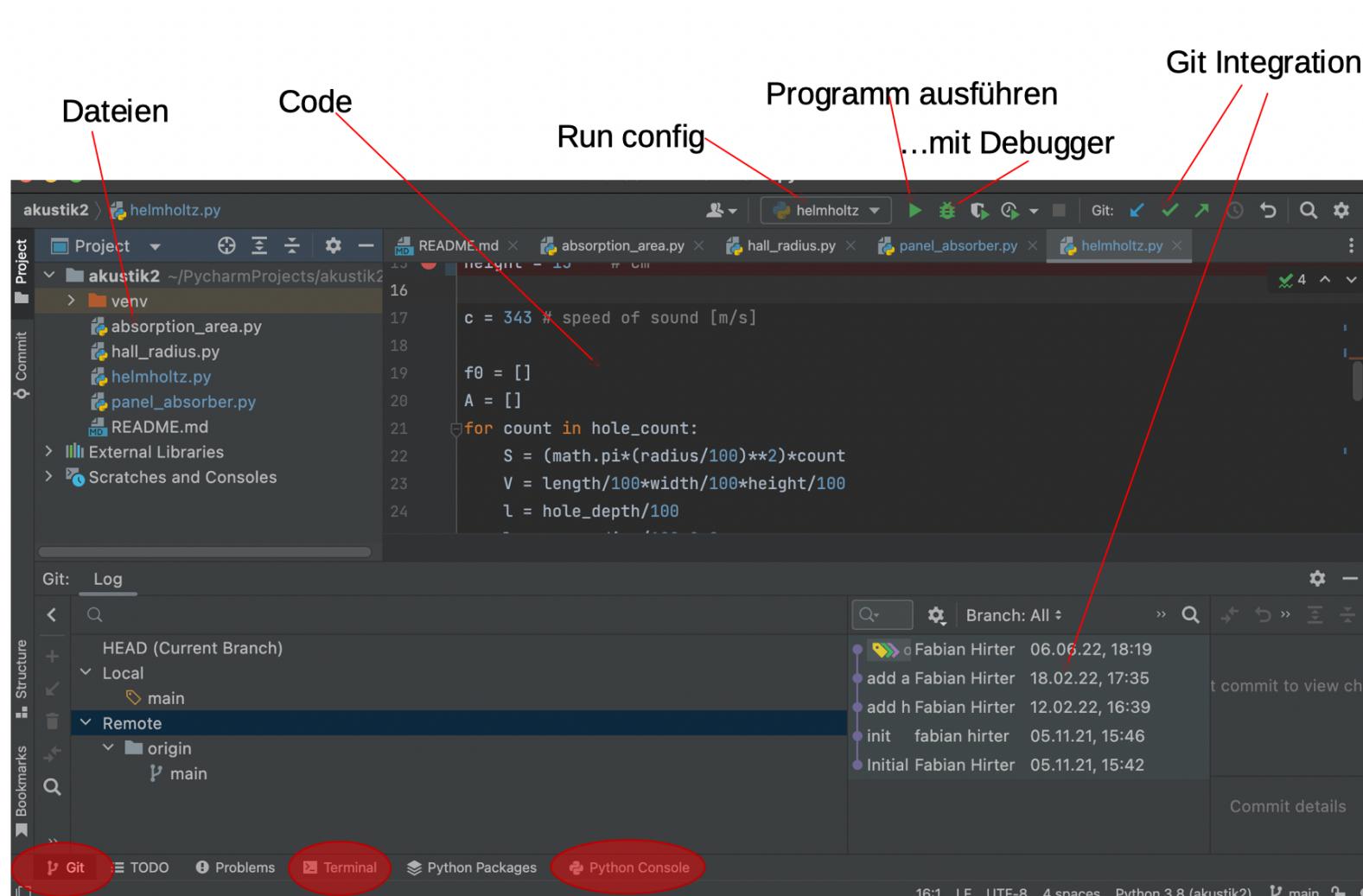
## **Microsoft Visual Studio Code**

- JavaScript, TypeScript, HTML, CSS, etc
- Open Source, Proprietär, frei

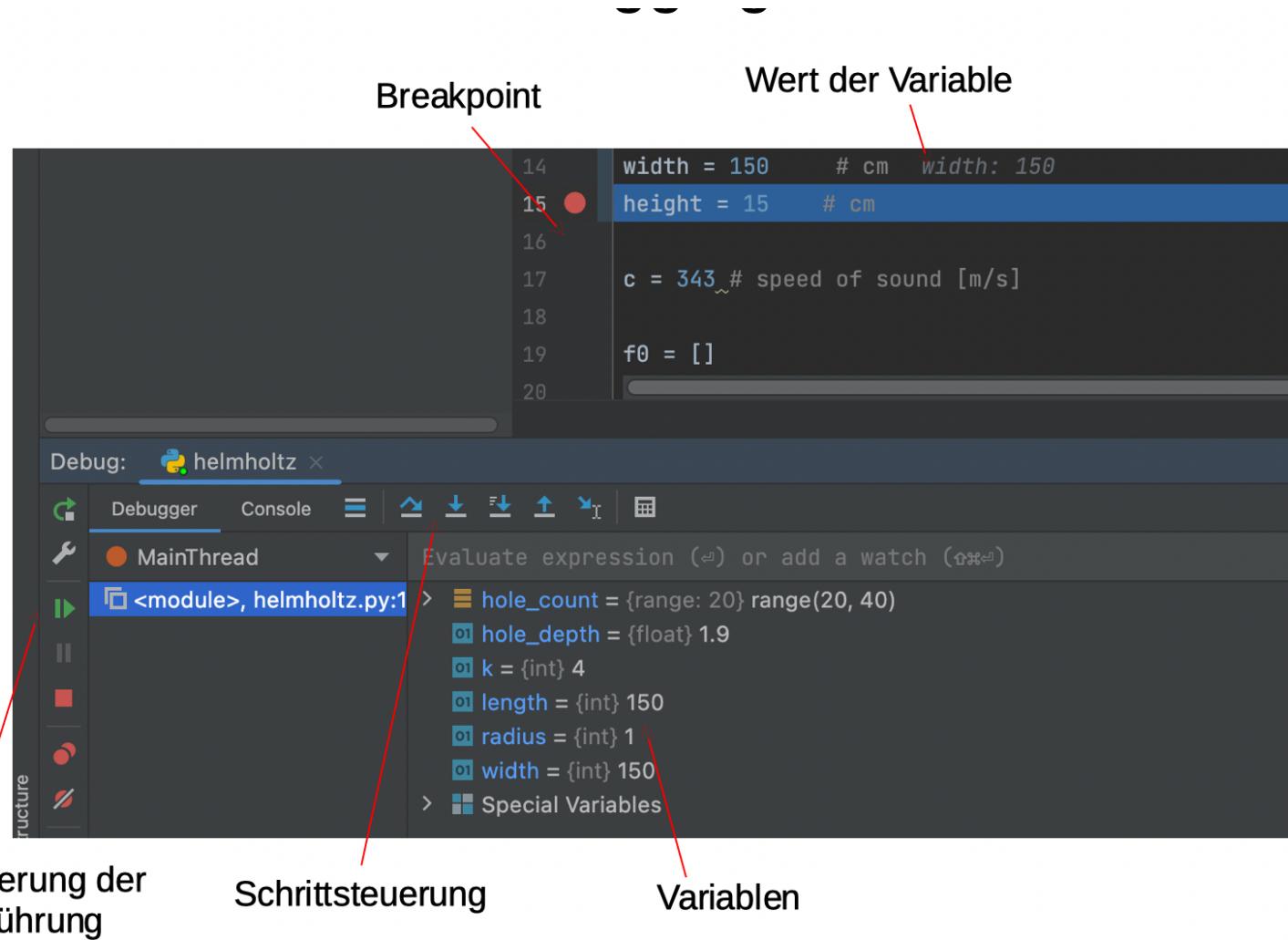
## **JetBrains**

- Java, Kotlin, Groovy, Scala, JavaScript, TypeScript, C (CLion), PHP (PHPStorm), Ruby (RubyMine), Python (PyCharm), iOS (AppCode), Android (AndroidStudio), C## (Rider)
- Teilweise OpenSource (Community Version)

# Jetbrains PyCharm



# Debugging



# Versionsverwaltung Basics

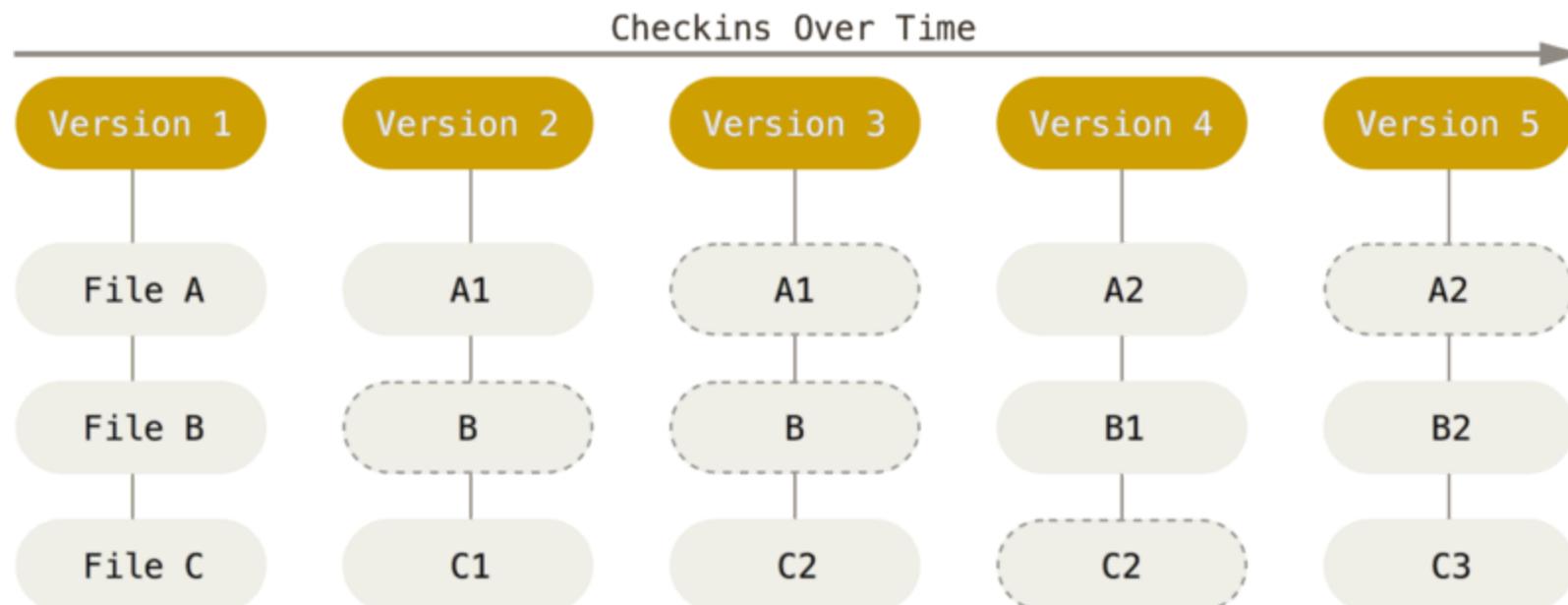
- Protokollierung von Änderungen
- Wiederherstellung von alten Ständen
- Archivierung
- Koordinierung des gemeinsamen Zugriffs
- Entwicklungszweige (Branches) -> **Don't Branch!**

# Moderne Versionsverwaltung

- CI/CD
- GitOps
- Infrastructure as Code
- Documentation as Code
  - Markdown
  - MKDocs
  - PlantUML
- Everything as Code

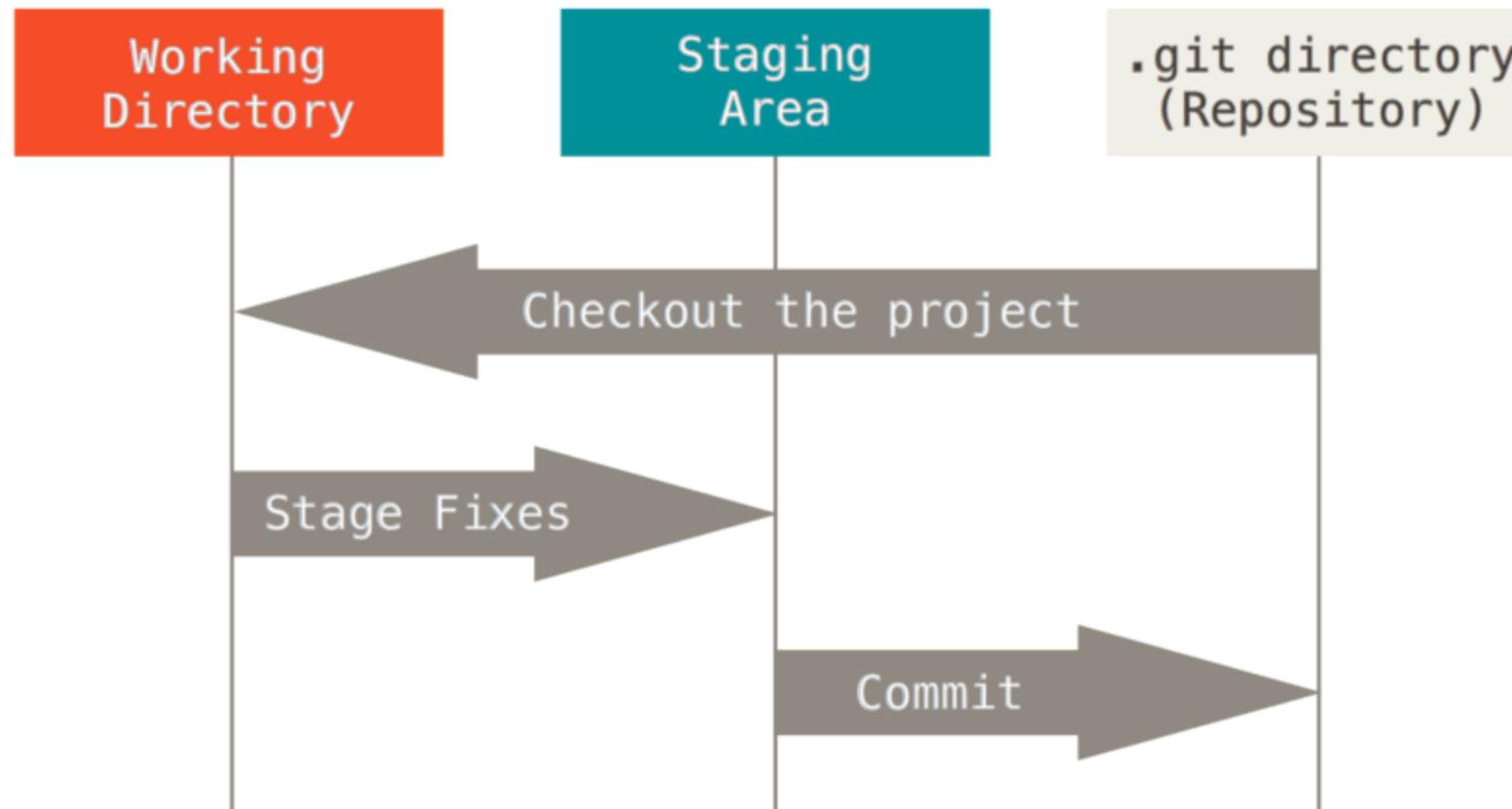
# Git

- Fast jede Funktion arbeitet lokal -> Repository wird repliziert
- Optimistic Locking
- Git stellt Integrität sicher
- Git fügt im Regelfall nur Daten hinzu



# Die drei Zustände

- Modified
- Staged
- Committed



# Arbeiten mit Git

## Initialisieren

- Auf Github oder Gitlab ein leeres Projekt erstellen
- Dieses Projekt lokal klonen `git clone`
- User name setzen: `git config user.name <name>`

## Arbeitsablauf

- Lokales Repository aktualisieren `git pull origin`
- Source Dateien erstellen oder editieren
- Änderungen zum Staging Area hinzufügen `git add <directory>` (z.B. ".")
- Änderungen im Repository festhalten `git commit -m "<message>"` (z.B. "change data type")
- Lokales Repository aktualisieren `git pull <remote>` (z.B. "origin")
  - Mit Rebase bleibt die History aufgeräumter: `git pull --rebase`
- Änderungen auf Github/Gitlab/Bitbucket laden `git push <remote> <branch>` (z.B. "origin main")

## CI/CD mit Git

- Tests und Linter werden bei Commit automatisch ausgeführt und Commit ggf. abgelehnt.
- Mit Tags werden Releases markiert. [semantic versioning](#).
- Das neuste Release wird automatisch deployed.
- [Changelogs werden automatisiert anhand der Git Messages generiert](#)

# Commit Messages

- Your Git Commit History Should Read Like a History Book

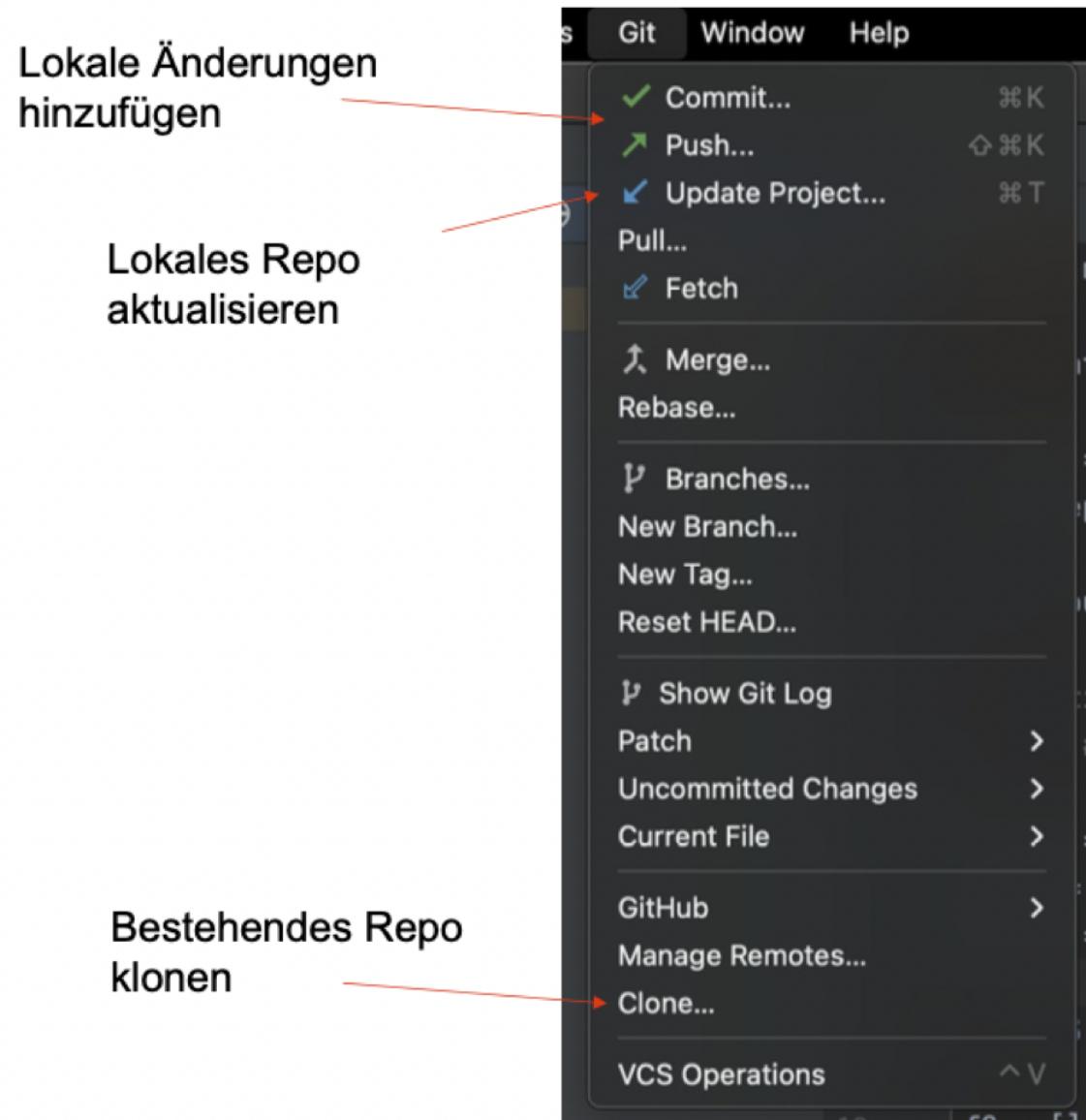
```
feat(logging): added logs for failed signups
```

```
fix(homepage): fixed image gallery
```

```
test(homepage): updated tests
```

```
docs(readme): added new logging table information
```

# PyCharm Git Integration



## Commit Messages

- add helmholtz calculation
- add absorption coefficient
- add hall radius calculation
- init
- Initial commit

## Position der branches in der historie

 origin & main	Fabian Hirter	06.06.22, 18:19
	Fabian Hirter	18.02.22, 17:35
	Fabian Hirter	12.02.22, 16:39
	fabian hirter	05.11.21, 15:46
	Fabian Hirter	05.11.21, 15:42

Autor und Datum des Commits

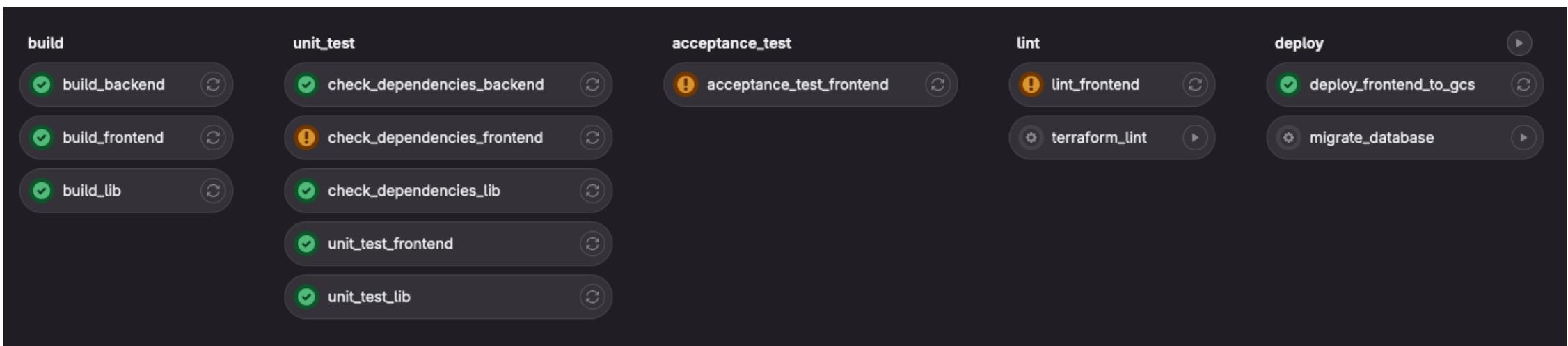
# Ressourcen

- [Cheatsheet](#)
- [Atlassian Tutorials](#)
- [Git Tutorials](#)
- [Simulationstool](#)

# Github / Gitlab

- Git Server
- CI/CD Plattform
- Issue Tracking / Projektmanagement
- Dokumentation
- Webhosting
- Release Management

# CI/CD Plattform



# Issue Tracking

The image shows a digital issue tracking board with the following structure:

- Open (27 issues):**
  - Parcel visualization (#95)
  - Tags as out of ordinary visualization (#94)
  - Payment integration (#45)
  - Set Security Headers (#91)
  - lane editing (#6)
- Next (6 issues):**
  - Authentication
    - Security (#39)
  - Create REST API
    - Feature (#51)
  - Customer & Product integration
    - Feature (#31)
  - After reload entity version is off
    - Bug (#88)
  - input validation
- In Progress (2 issues):**
  - Multitenancy
    - Feature (#46)
  - compact mode (#8)
- Closed (30 issues):**
  - deploy backend code using gsutils (DX) (#84)
  - accept partial shipment in updateShipment (#82)
  - do not loose parcel on validation error (Bug) (#90)
  - Deploy Frontend in GCP (#83)
  - add feature flags (#37)
  - MVVC Pattern (#80)

**Multitenancy**

Open Issue created 5 months ago by Fabian Hirter

Edit ⋮

Like 0 Dislike 0 Comment

Create merge request ⋮

Drag your designs here or [click to upload](#).

**Tasks** 2

Show labels ✓ Add ⋮

<span>Green</span> Add org id to event when persisting	<span>Clock</span> We have a Prod... <span>X</span>
<span>Green</span> use raccoon lib in backend	<span>Clock</span> We have a Prod... <span>X</span>

**Linked items** 0

Add ⋮

Link issues together to show that they're related. [Learn more](#).

**Activity**

Sort or filter ⋮

Preview B I ♂ ≡ </> ♂ ≡ ≡ ↶ ↷ ↶ ↷ ↶ ↷ ↶ ↷

Write a comment or drag your files here...

Switch to rich text editing [M+]

Make this an internal note ?

Comment ⋮ Close issue

**0 Assignees** None - assign yourself Edit

**Labels** Feature X In Progress X Edit

**Milestone** We have a Product Edit

**Weight** This feature is locked. [Learn more](#) ⋮

**Due date** None Edit

**Time tracking** ⌚ + No estimate or time spent

**Confidentiality** ⌚ Not confidential Edit

**1 Participant** 

Move issue

# Klassen und Objekte

**Eine Klasse: eine Software Maschine**

# Was ist ein Objekt?

Es gibt verschiedene Arten von Objekten:

- “physische Objekte”: bilden physische Objekte ab, z.B. eine Ampel oder ein Auto
- “abstrakte Objekte”: Beschreiben abstrakte Dinge aus der modellierten Welt, z.B. eine Route oder eine Himmelsrichtung
- “Softwareobjekte”: Reine Softwareelemente, z.B. Datenstrukturen wie Arrays oder Listen
- Ein grosser Vorteil der objektorientierten Programmierung ist, dass die Software anhand der «echten» Welt modelliert werden kann.

## Was ist ein Objekt?

- Ein Objekt besitzt Daten → Eigenschaften / Felder
- Ein Objekt kann Operationen ausführen → Funktionen / Methoden

Ein Objekt kann Operationen ausführen und dazu auf seine Daten zugreifen und diese ändern.

# Methoden

- Entspricht dem Begriff "Funktion" der strukturierten Programmierung
- Eine Operation, die von Objekten ausgeführt werden kann.
- Abfragen, Befehle
- Name der Methode kann, mit Einschränkungen, frei gewählt werden.
- Eine Methode von einem Objekt wird in den meisten Sprachen mit dem `.` aufgerufen:  
``<objekt>.<methode>`

# Methoden

- Methoden können Argumente haben:
  - `primaryStage.setTitle("Ampelsteuerung");`
- Mehrere Argumente werden durch Komma getrennt:
  - `primaryStage.add(crossroadController, 1100, 900);`
- Weniger Argumente sind übersichtlicher! (Faustregel: Max. 3)

# Ein Objekt hat eine Schnittstelle (Interface)



# Ein Objekt hat eine Implementierung



# Abstraktion

- Ein Objekt kann verwendet werden, ohne die Implementierung der Methoden zu kennen.
- Die Implementierungsdetails sind abstrahiert

# Kapselung

- Grundsätzlich sind alle Felder (Daten) privat, d.h. nur für das eigene Objekt zugänglich.
- Diese Felder stellen den Zustand (State) des Objekts dar.
- Zugriff auf Felder wird mit Methoden gewährt (sogenannte Setter und Getter, z.B. `setColor()`, `getSize()`)
- Es werden nur die nötigen Methoden zugänglich gemacht.
- Die Programmiersprache stellt den Zugriffsschutz sicher.

## Objektorientierung: Geschichte

- Untergruppe der imperativen Programmierung (Abfolge von Befehlen)
- Ursprung: Simula67, Oslo, 60er Jahre
- Kaum verbreitet in den 70er Jahren
- Smalltalk (Xerox PARC, 1970s) machte OOP Populärer

- Grosser Verbreitung in den 90er Jahren
- Die meisten heute verbreiteten Sprachen sind objektorientiert: Objective C, C++, Java, C#, Python, Kotlin, Go, JavaScript, uvm
- Heute das meistverbreitete Konzept der Softwareentwicklung
- Andere Programmierparadigmen:
  - Imperative Programmierung
    - Strukturierte Programmierung
    - Prozedurale Programmierung
  - Deklarative Programmierung
    - funktionale Programmierung

# Syntaktische Struktur einer Klasse

```
class Vehicle:                                # Class Name
    def __init__(self, brand, model, type):   # Constructor
        self.brand = brand
        self.model = model
        self.type = type
        self.gas_tank_size = 14
        self.fuel_level = 0

    def fuel_up():                           # Method declaration
        self.fuel_level = self.gas_tank_size # Method implementation
        print('Gas tank is now full.')

    def drive(self):
        print(f'The {self.model} is now driving.')
```

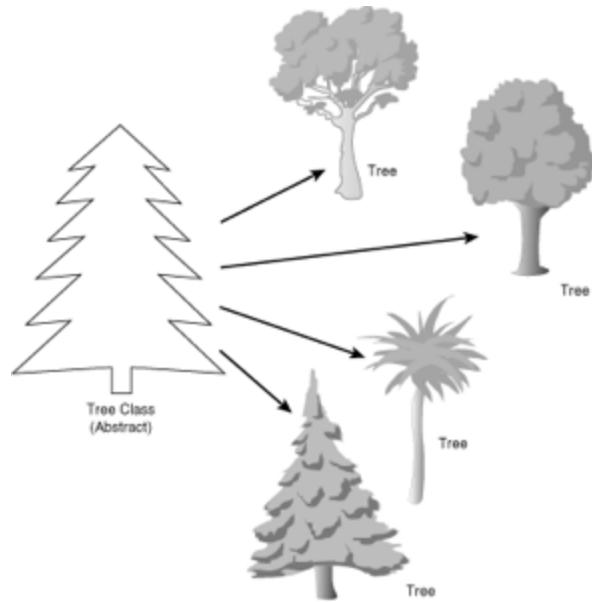
# Klasse

- Jedes Objekt gehört zu einer Klasse, welche die zur Verfügung stehenden Methoden und Felder definiert.
- Eine Klasse ist eine Beschreibung von Laufzeit-Objekten, welche dieselben Eigenschaften und Methoden besitzen.
- Eine Klasse ist eine Kategorie von Dingen
- Ein Objekt ist eines von diesen Dingen

# Objekte

Wenn ein Objekt O ein Objekt der Klasse C ist:

- O ist ein Exemplar von C
- O ist eine Instanz von C



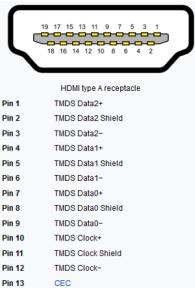
# Objekte und Klassen

- Klassen existieren nur im Source Code
  - Bauplan für konkrete Objekte
- Objekte existieren nur zur Laufzeit

- Ein zentraler Aspekt der Softwareentwicklung ist das Bilden von sinnvollen Klassen für die Aufgabenstellung (Softwarearchitektur, OOD)
- Das Schreiben der Details wird Implementierung genannt.

# Interface (de: Schnittstelle)

- Die Schnittstelle (engl. interface) ist der Teil eines Systems, welcher der Kommunikation dient.
- Der Begriff stammt ursprünglich aus der Naturwissenschaft [...]. Er beschreibt bildhaft die Eigenschaft eines Systems als Black Box, von der nur die „Oberfläche“ sichtbar ist, und daher auch nur darüber eine Kommunikation möglich ist. [...]
- Daneben bedeutet das Wort „Zwischenschicht“: Für die beiden beteiligten Boxes ist es ohne Belang, wie die jeweils andere intern mit den Botschaften umgeht, und wie die Antworten darauf zustande kommen.



# Interfaces

- User interface: Wenn die Clients Menschen sind
  - GUI: Graphical User Interface
  - Text interfaces, command line interfaces.
- Program interface: Wenn die Clients Software sind
  - API: Application Program Interface

# API

- Eine Schnittstelle gibt an, welche Methoden vorhanden sind oder vorhanden sein müssen.
- Zusätzlich zu dieser syntaktischen Definition sollten Vorbedingungen und Nachbedingungen der verschiedenen Methoden definiert werden.
- Heute werden dazu in der Regel automatisierte Tests geschrieben.
- Es kann auch in der Dokumentation festgehalten werden.

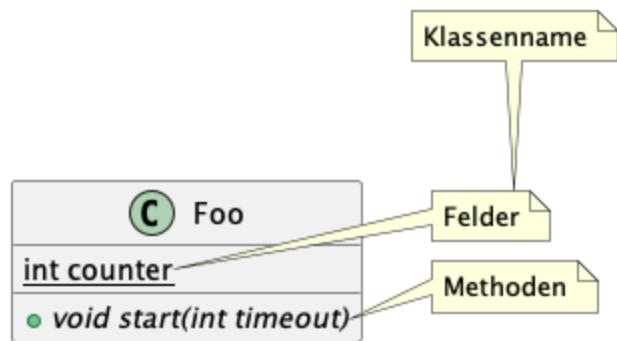
# Schnittstellen definieren

- Nicht jede Methode ist für jeden möglichen Parameter geeignet
- Lösungen:
  - immer: gute Wahl der Bezeichner
  - möglichst immer: Tests
  - Einschränkung durch Datentyp
  - wenn nötig: Kommentare: JavaDoc
  - falls erforderlich: Exceptions

# Javadoc

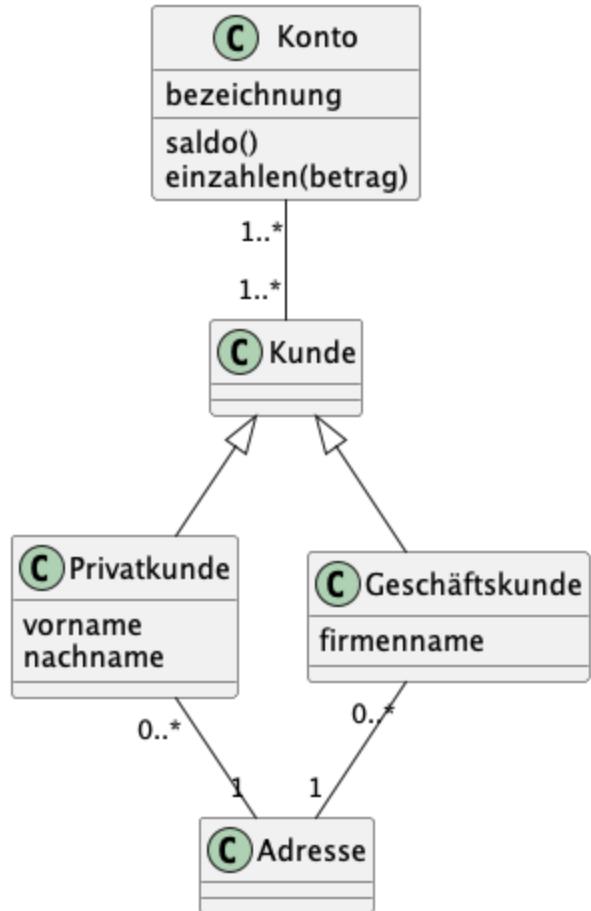
```
/**  
 * Returns an Image object that can then be painted on the screen.  
 * The url argument must specify an absolute {@link URL}. The name  
 * argument is a specifier that is relative to the url argument.  
 * <p>  
 * This method always returns immediately, whether or not the  
 * image exists. When this applet attempts to draw the image on  
 * the screen, the data will be loaded. The graphics primitives  
 * that draw the image will incrementally paint on the screen.  
 *  
 * @param url an absolute URL giving the base location of the image  
 * @param name the location of the image, relative to the url argument  
 * @return the image at the specified URL  
 * @see Image  
 */  
public Image getImage(URL url, String name) {  
    try {  
        return getImage(new URL(url, name));  
    } catch (MalformedURLException e) {  
        return null;  
    }  
}
```

# UML Klassendiagramm



PlantUML

# UML Klassendiagramm



# PlantUML

```
@startuml
class Konto {
    bezeichnung
    saldo()
    einzahlen(betrag)
}

class Kunde {}

class Privatkunde {
    vorname
    nachname
}

class Geschäftskunde {
    firmenname
}

class Adresse {}

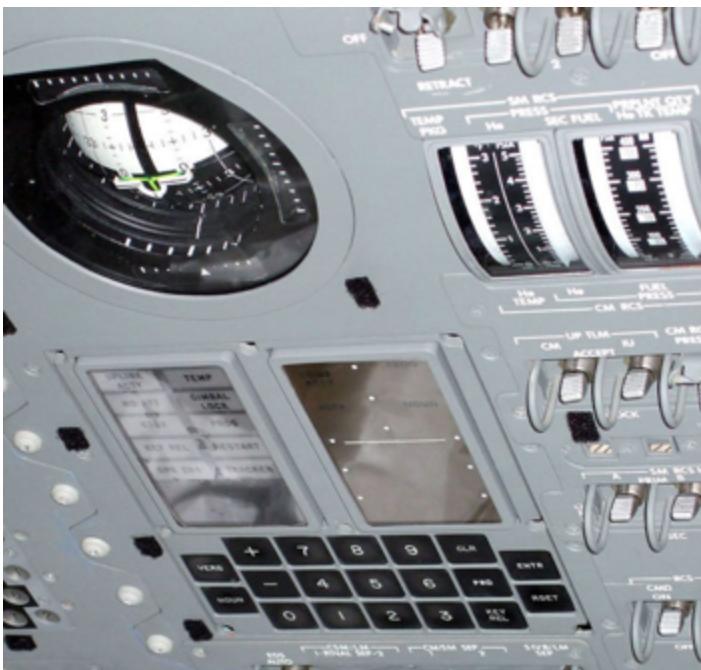
Kunde <|-- Privatkunde
Kunde <|-- Geschäftskunde

Privatkunde "0..*" -- "1" Adresse
Geschäftskunde "0..*" -- "1" Adresse

Konto "1..*" -- "1..*" Kunde
@enduml
```

# Moderne Softwareentwicklung

# Software Engineering



# Kundenorientierung

**Software soll den Kunden Mehrwert bringen**

- Software soll stabil laufen
- Neue Features sollten schnell umgesetzt und nutzbar sein
- Softwaresysteme werden immer komplexer

## Teamarbeit

- Mehrere Personen arbeiten am selben Softwareprojekt
- Versionsverwaltung wird verwendet (Git, SVN)
- Konflikte entstehen und sind aufwendig

# Lösungen

- Kleine Arbeitspakete iterativ und inkrementell
- Kurzer Feedbackloop
- Komplexität reduzieren
- Hohe Qualität

# Alles hängt zusammen

- Hohe Qualität reduziert Komplexität
- Hohe Qualität kürzt den Feedbackloop durch schnelle Entwicklung
- Kleine Arbeitspakete kürzen den Feedbackloop
- Kleine Arbeitspakete reduzieren Komplexität
- ...

# Manifesto for Agile Software Development

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

<https://agilemanifesto.org/>

# Iteratives und inkrementelles Arbeiten

Iterative



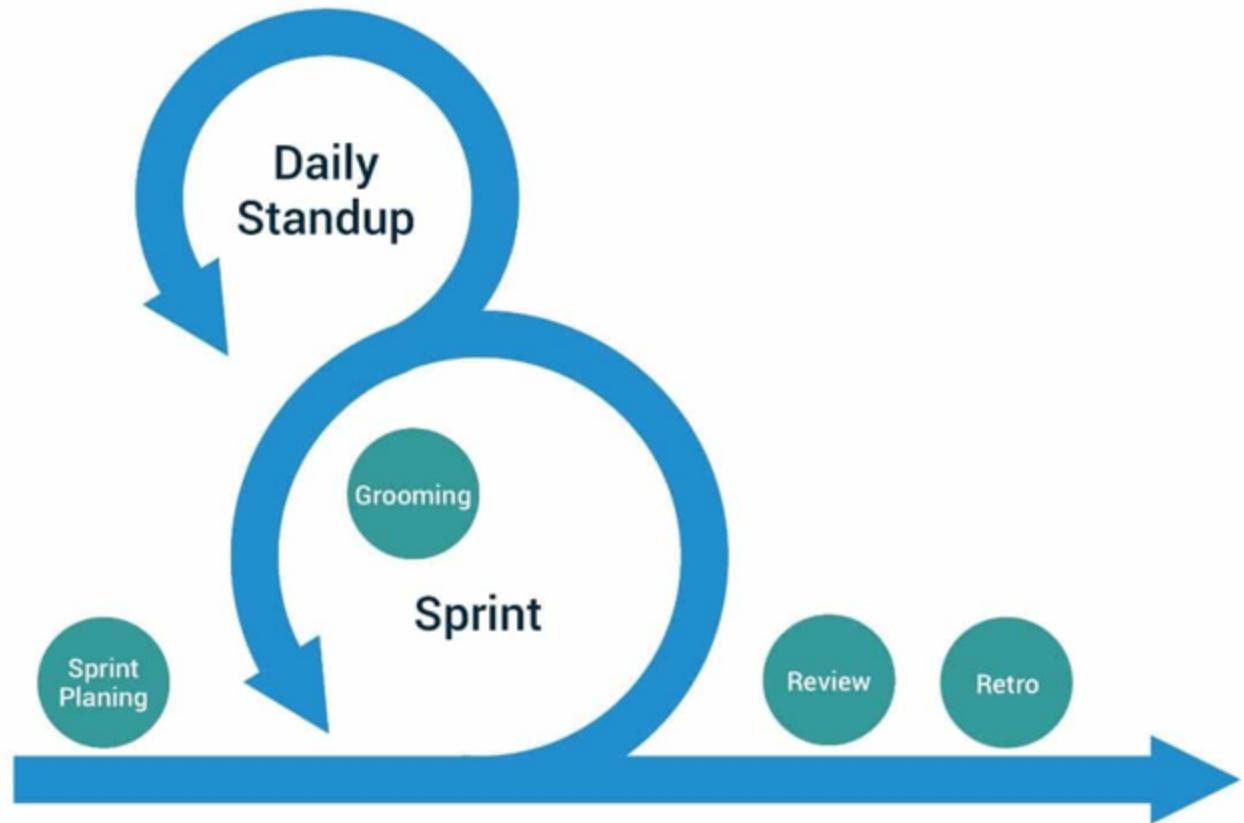
Incremental



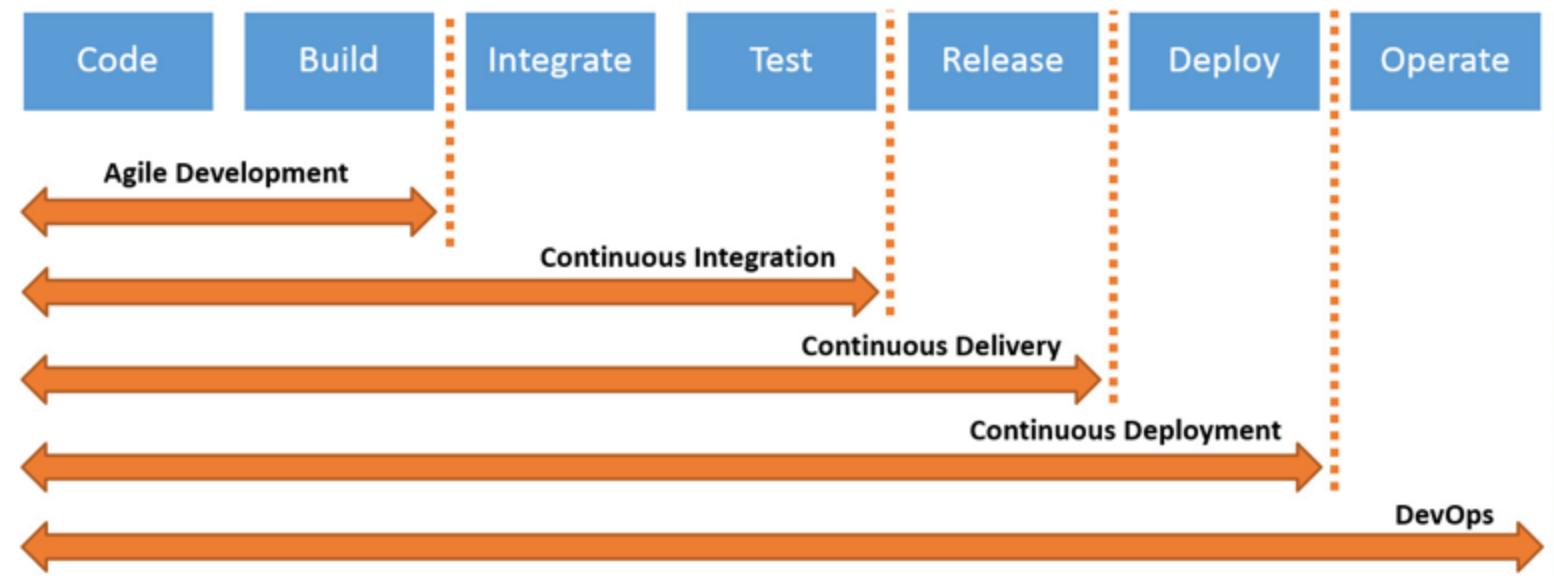
Iterative &  
Incremental



# Iterationen



# Kurzer Feedbackloop: CI/CD

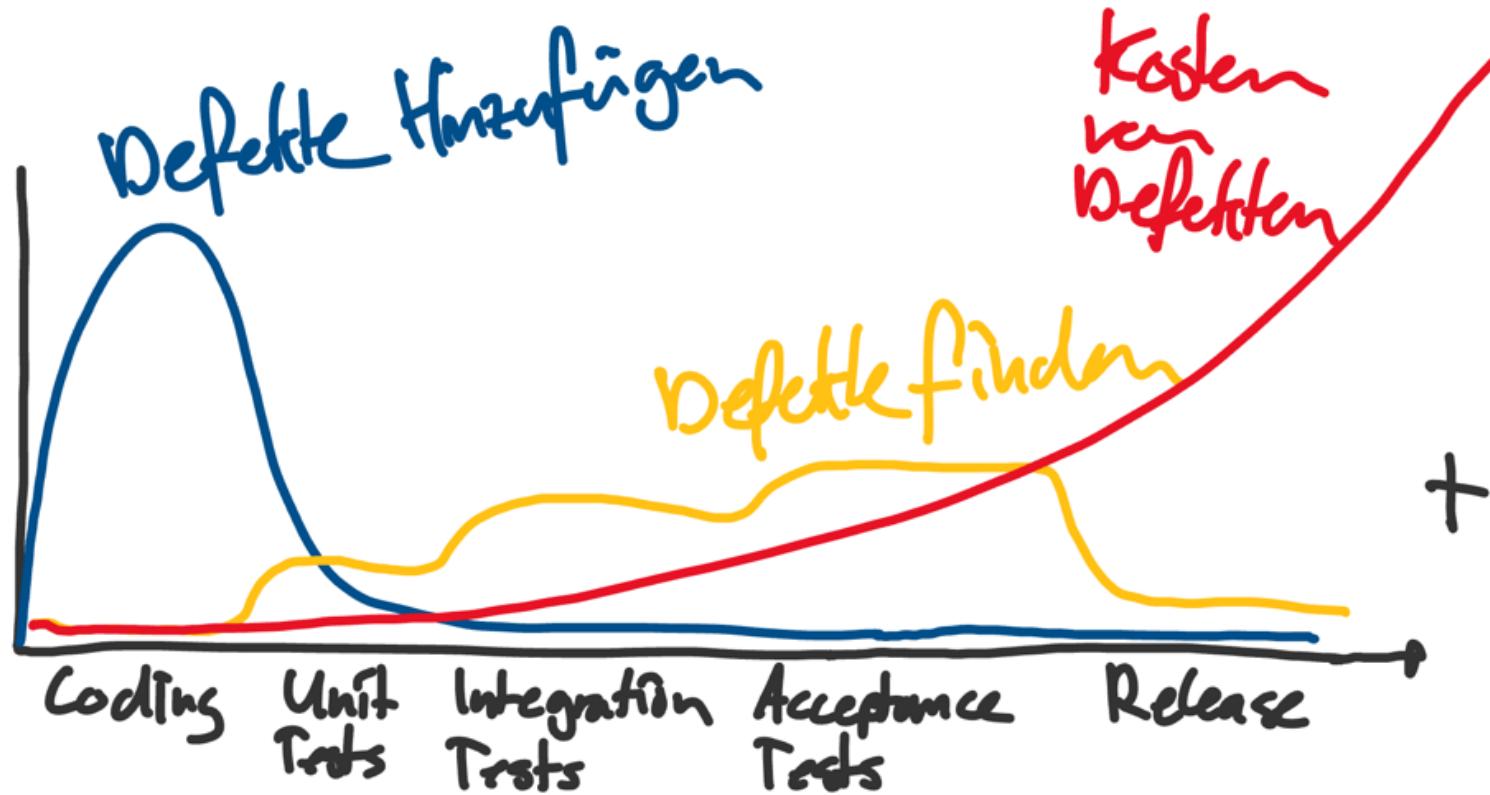


# CI / CD

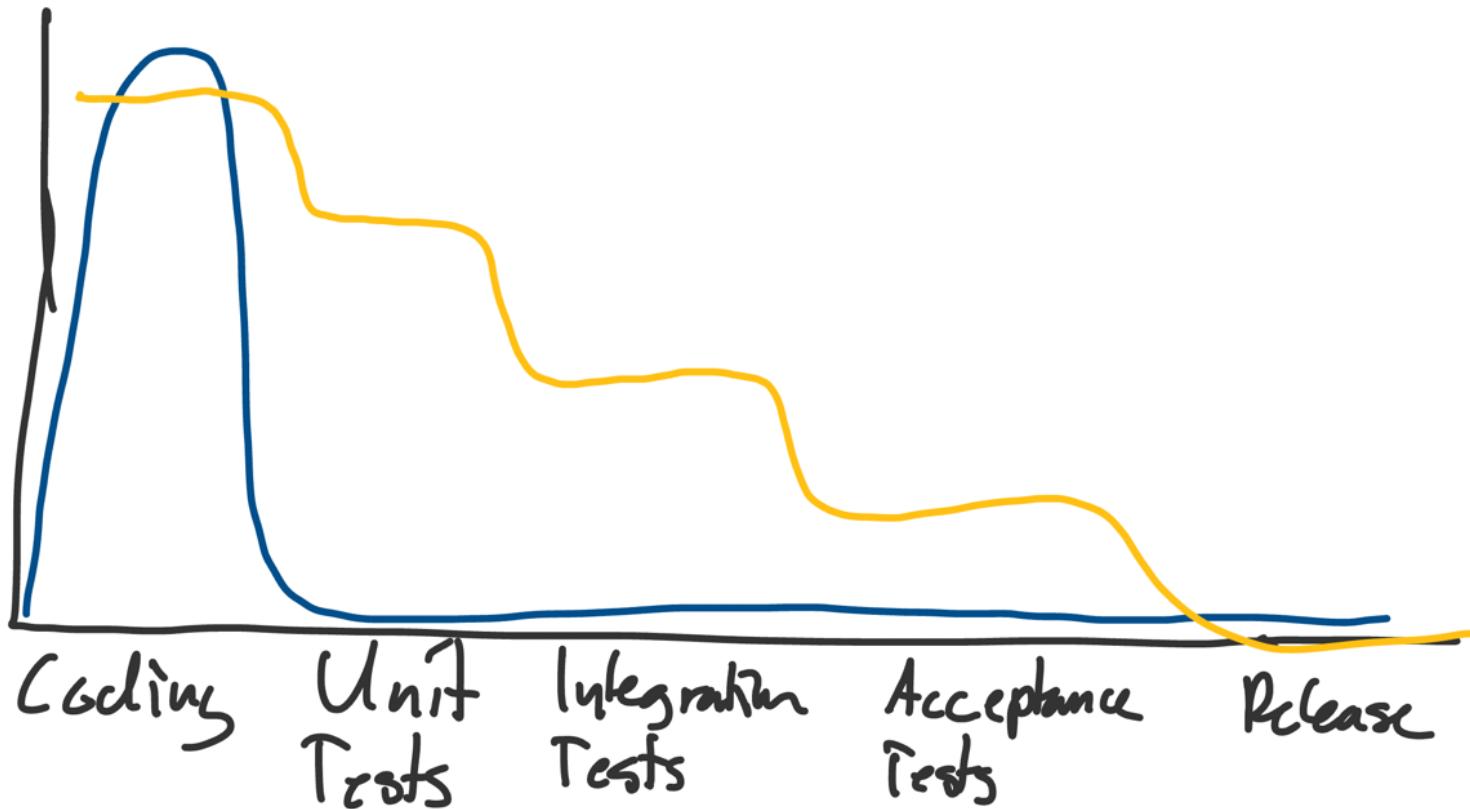
- Ziel: Releases werden vereinfacht
- Time to market ist kürzer, neue Features sind sofort verfügbar
- Durch automatisierte deployments ist der Aufwand initial höher, anschliessend jedoch sehr klein
- Nur möglich mit automatisierten Tests

# Testing

# Kosten von Defekten

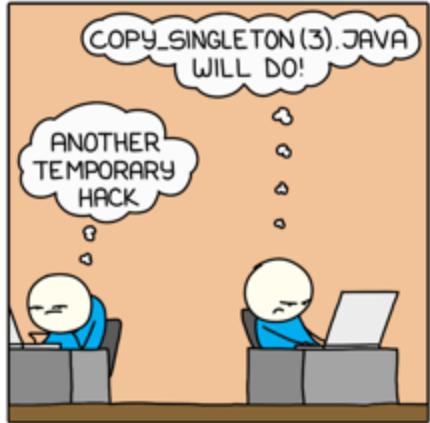


# Kosten von Defekten

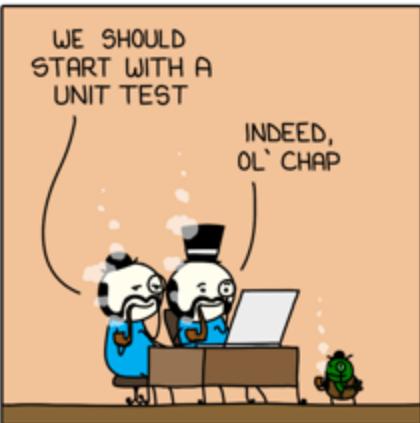
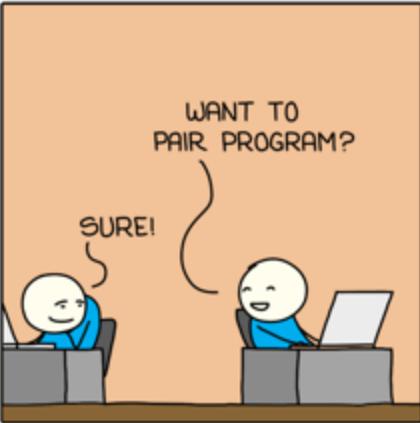


# Pair Programming

PAIR PROGRAMMING



MONKEYUSER.COM



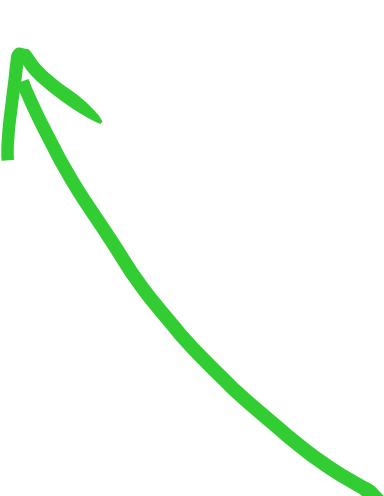
# **Test Driven Development (TDD)**

Following 3 Drawings from Growing Object-Oriented Software by Nat Pryce and Steve Freeman.

Write a  
failing  
test

Make the  
test pass

Refactor

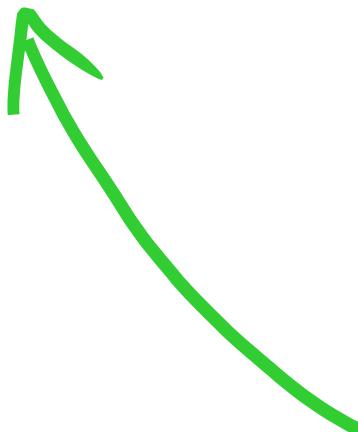
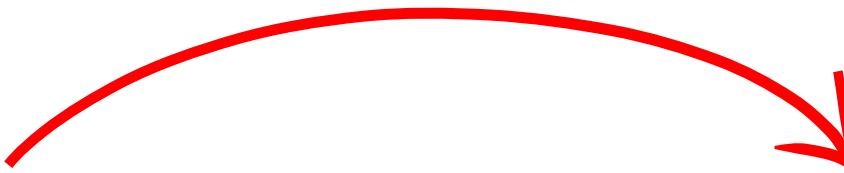


Write a  
failing  
test

Make the  
test pass

Refactor

Hard to write a test?

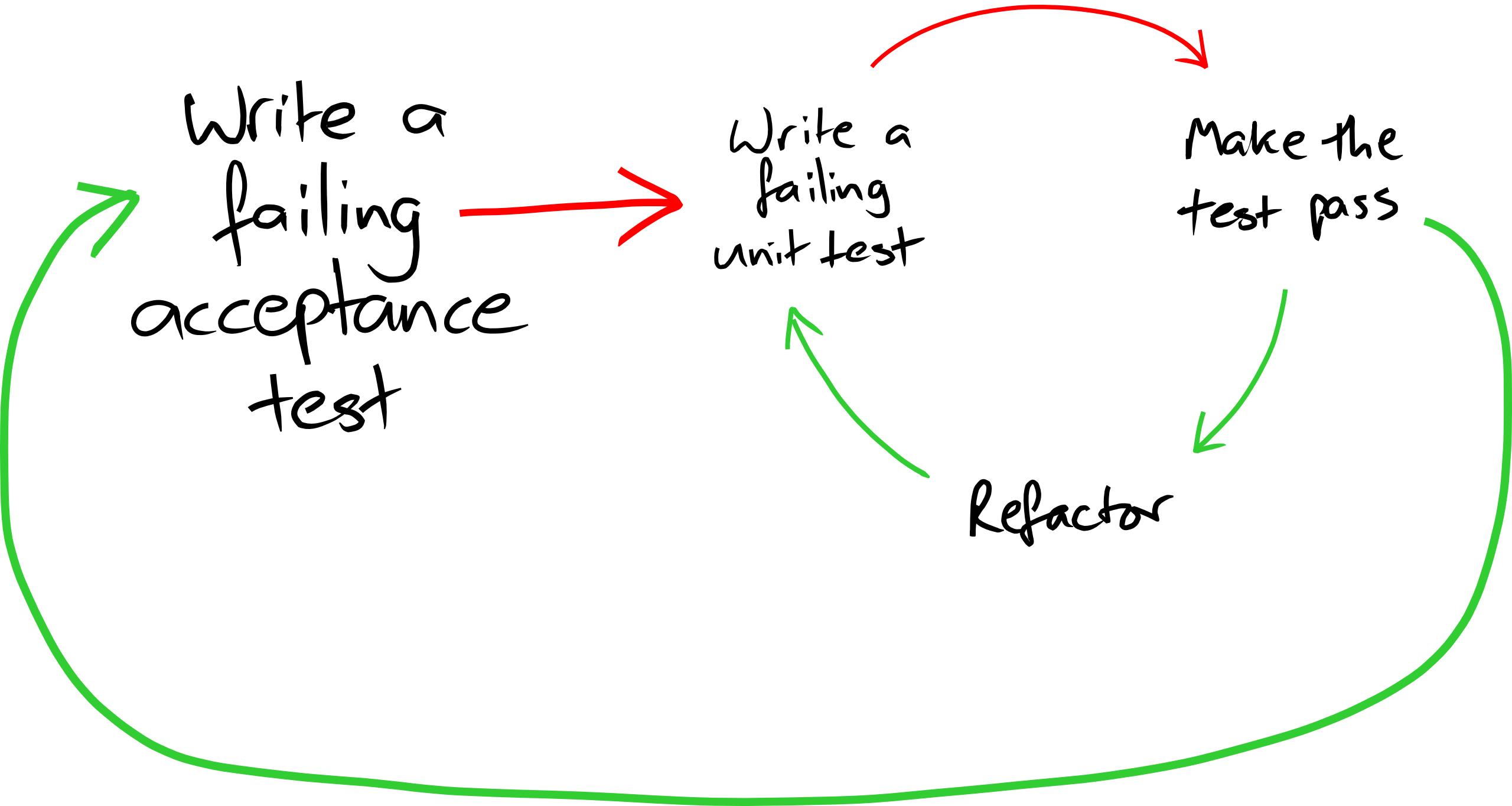


Write a  
failing  
acceptance  
test

Write a  
failing  
unit test

Make the  
test pass

Refactor

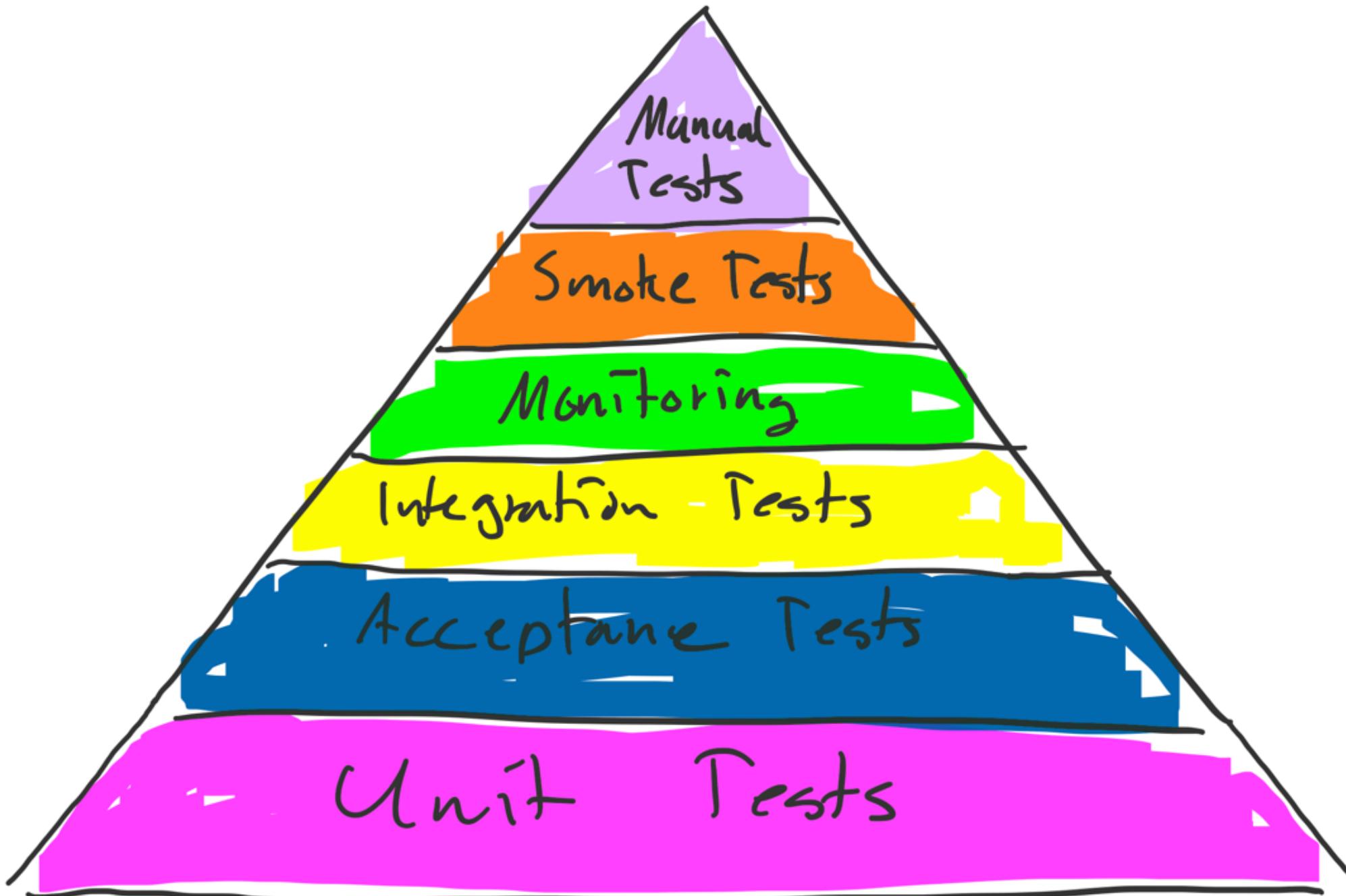


## Why Should You Refactor?

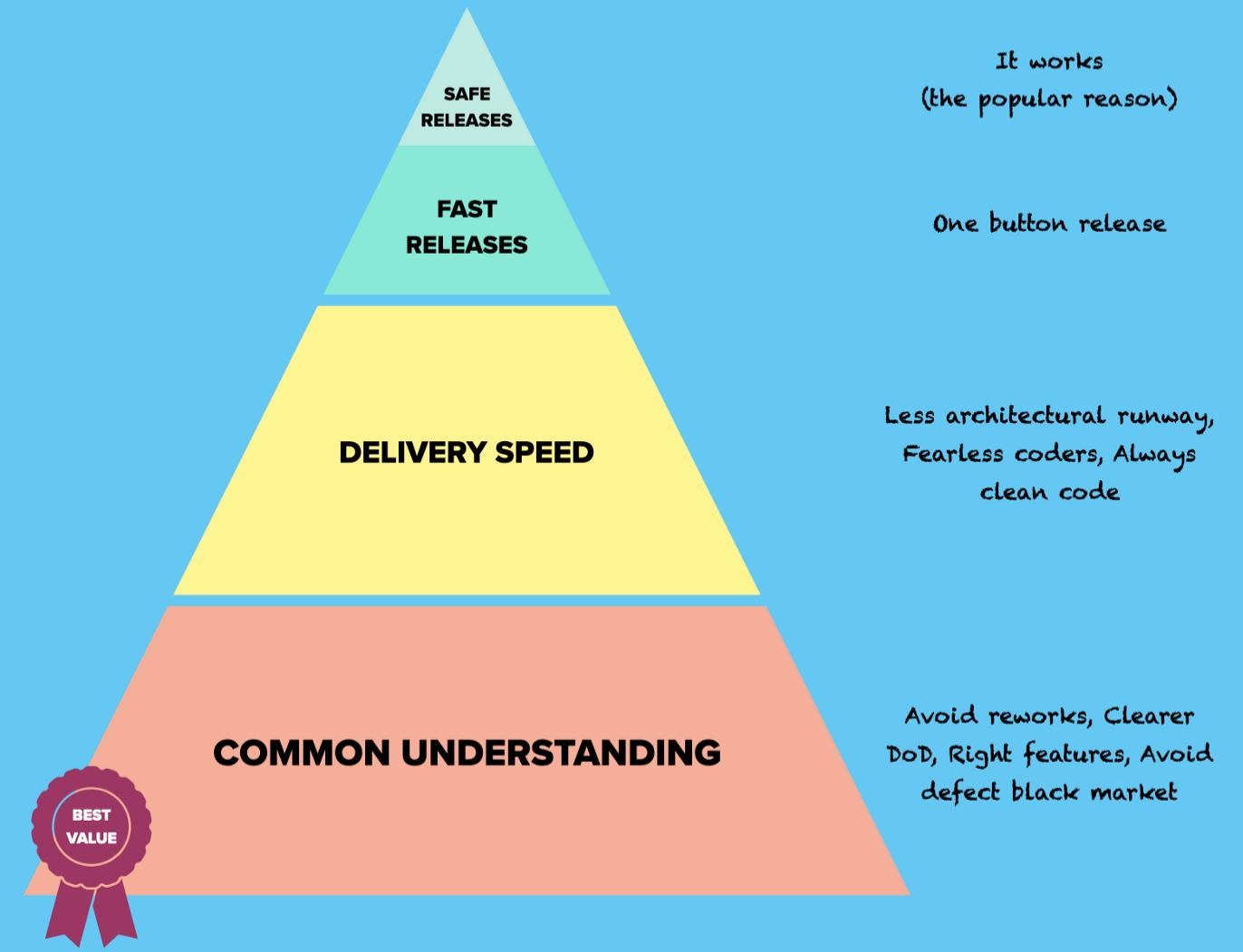
- Refactoring Improves the Design of Software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You Find Bugs
- Refactoring Helps You Program Faster

## When Should You Refactor?

- [The Rule of Three](#)
- Refactor When You Add Functionality
- Refactor When You Need to Fix a Bug
- Refactor As You Do a Code Review



# WHY TO TEST PYRAMID



## Testing: AAA

- Arrange: Set up your data
- Act: Execute code under Test
- Assert: Verify that the result ist correct

## Testing: Further Reading

- How to write clear and robust unit tests: the dos and don'ts
- The Real Value of Testing

# Extreme Programming



# Embrace Change



# Algorithmen und Datenstrukturen

# Containerdatenstrukturen

- Enthalten andere Objekte («items»)
- Grundsätzliche Operationen:
  - Elemente hinzufügen
  - Elemente entfernen
  - Ein Element suchen
  - Über alle Elemente iterieren
- Verschiedene Implementationen unterscheiden sich
  - Welche Operationen möglich sind
  - Wie schnell diese sind
  - Wie der Speicher ausgenutzt wird

# Record

- Einfachste Anordnung von Daten
- Zeile in Datenbank / Tabelle

```
struct date {  
    int year;  
    int month;  
    int day;
```

- struct in C   };
- Datenobjekte
- Python: tup1 = ('physics', 'chemistry', 1997, 2000)

# Set

- Anordnung von Elementen
- keine Duplikate
- keine definierte Ordnung
- testen, ob Teil des Sets
- Python: `thisset = {"apple", "banana", "cherry"}`

# List

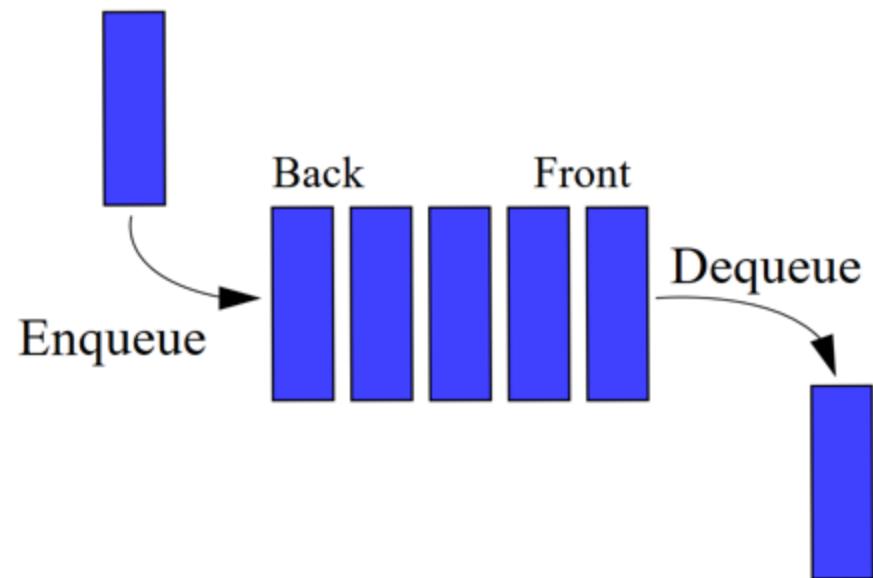
- Definierte Ordnung
- Elemente hinzufügen und entfernen
- Element mit einem Index abrufen
- Duplikate möglich
- Python:
  - `list2 = [1, 2, 3, 4, 5, 6, 7];`
  - Zugriff: `list1[0]` , `list1[1:5]`

# Map

- Schlüssel / Wert Paare
- Hinzufügen, Entfernen, Ändern, Abrufen
- Assoziatives Array, Lookup Table, Dictionary
- Python: Dictionaries (hash map)
  - Erstellen: `dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}`
  - Zugriff: `dict['Name']`

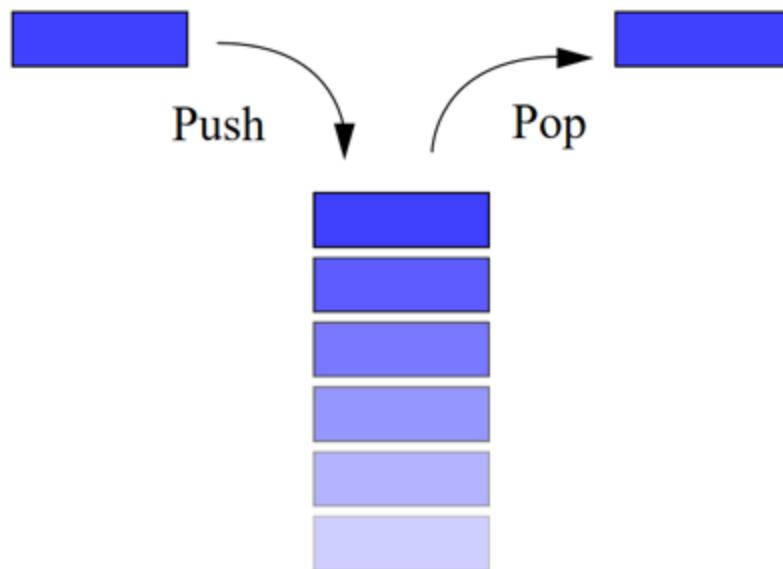
# Queue

- FIFO: First In, First Out
- Warteschlange, Pipe

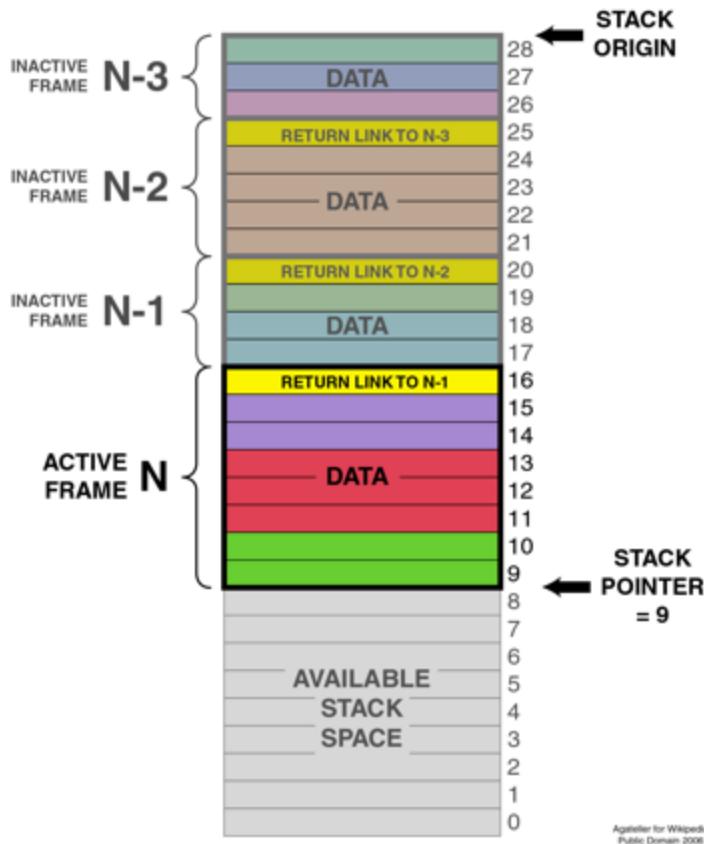


# Stack

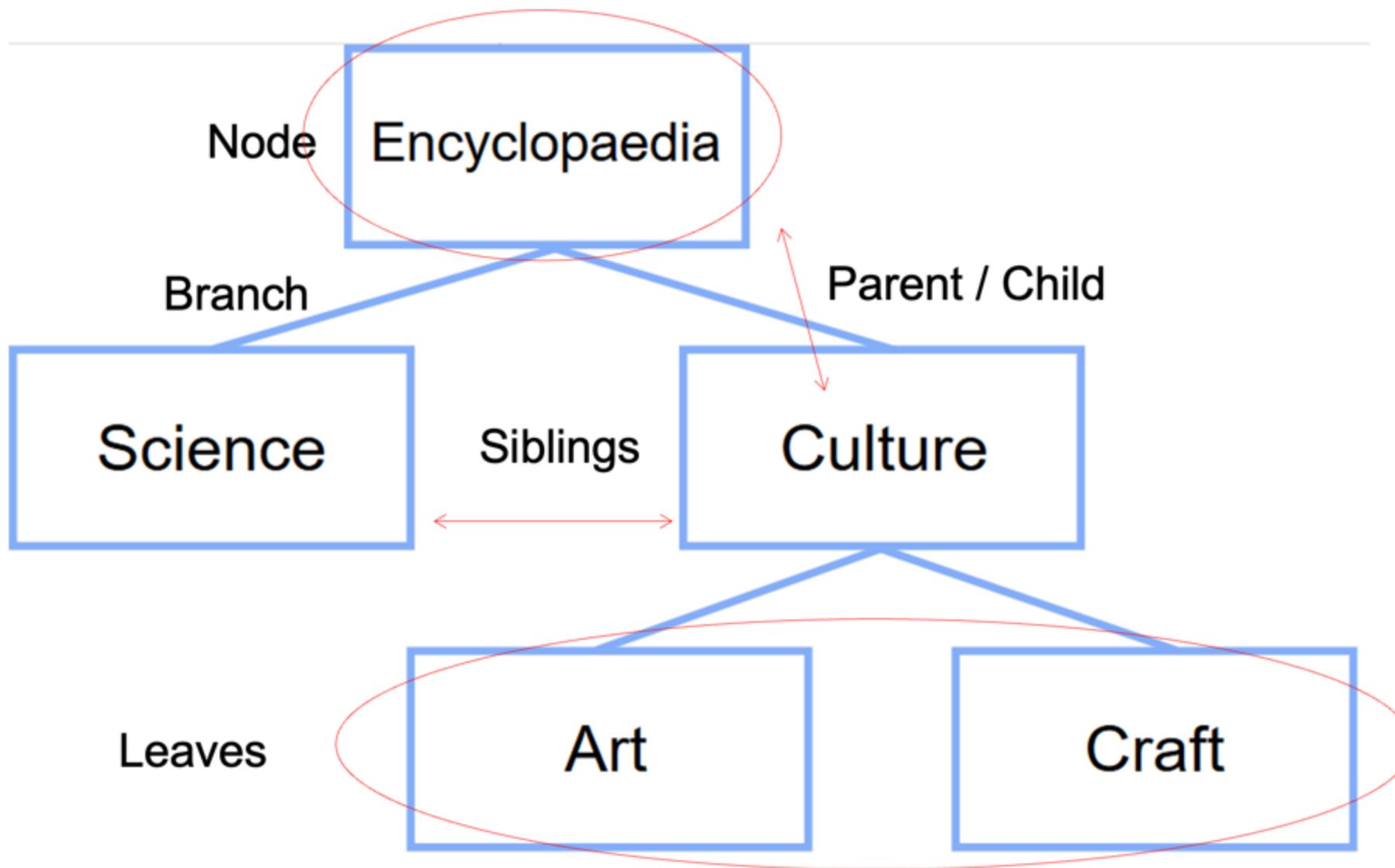
- LIFO: Last In, First Out
- push: Neues Element speichern
- pop: Letztes Element abrufen und entfernen
- Stapelspeicher, Kellerspeicher



# Stack



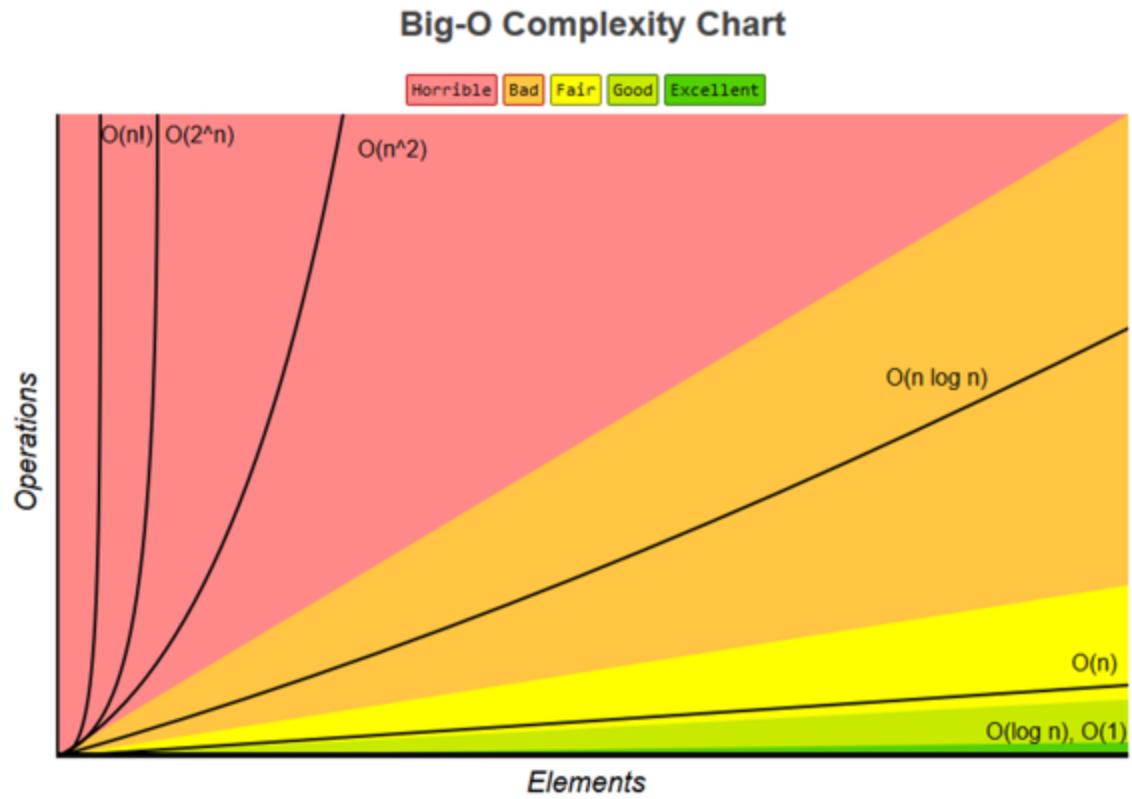
# Tree



# Konkrete Datenstrukturen

- Array
- Graph

# Big O Notation



# Komplexität von Algorithmen

- $O(1)$ : Operation dauert immer gleich lange, unabhängig von der Anzahl der Elemente
- $O(n)$ : Operation ist linear abhängig von der Anzahl der Elemente (Je mehr Elemente in der Liste, desto länger dauert die Operation)
- [Big O Cheatsheet](#)

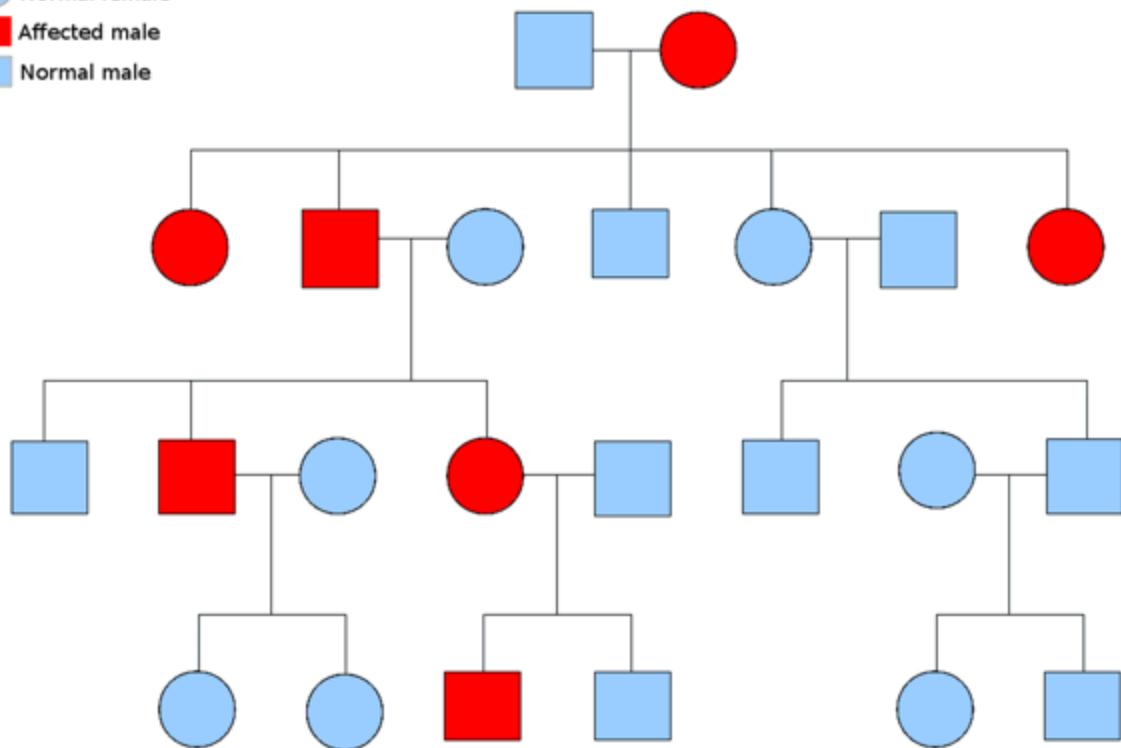
# Speicher und Rechenaufwand

Operation	Array	Linked List	Binary Tree	Hashtable
Einfügen	$O(n)$	$O(1)$		
Löschen	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$
Suche	$O(n)$	$O(n)$		
Zugriff auf beliebiges Element	$O(1)$	$O(n)$		-

# Objektorientierte Prinzipien

# Vererbung

- Affected female
- Normal female
- Affected male
- Normal male



# Vererbung

- Generalisierung / Spezialisierung
- Polymorphismus
- Dynamisches Binden

# Vererbung

Eine Klasse ist ein Modul:

- Eine Sammlung von Funktionalität (Methoden)
- Kapselung (nicht alle Funktionalität ist sichtbar)

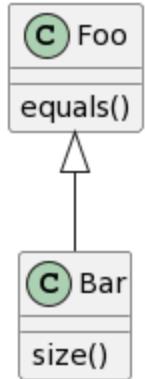
Eine Klasse ist ein Datentyp:

- Beschreibt die Art einer Objektinstanz
- Kann bei Variablen, Methoden oder Feldern verwendet werden

# Vererbung

- Eine neue Klasse kann als Erweiterung oder Spezialisierung einer existierenden Klasse beschrieben werden.
- Bar erbt von Foo
  - Modul: Alle Funktionalität von Foo steht in Bar zur Verfügung
  - Typ: Immer wenn eine Instanz von Foo benötigt wird, wird eine Instanz von Bar akzeptiert
- Oder umgekehrt: Eine neue Klasse kann eine existierende Klasse generalisieren

# Vererbung



```
@startuml  
Foo : equals()  
Bar : size()  
  
Foo <|-- Bar  
@enduml
```

# Vererbung: Terminologie

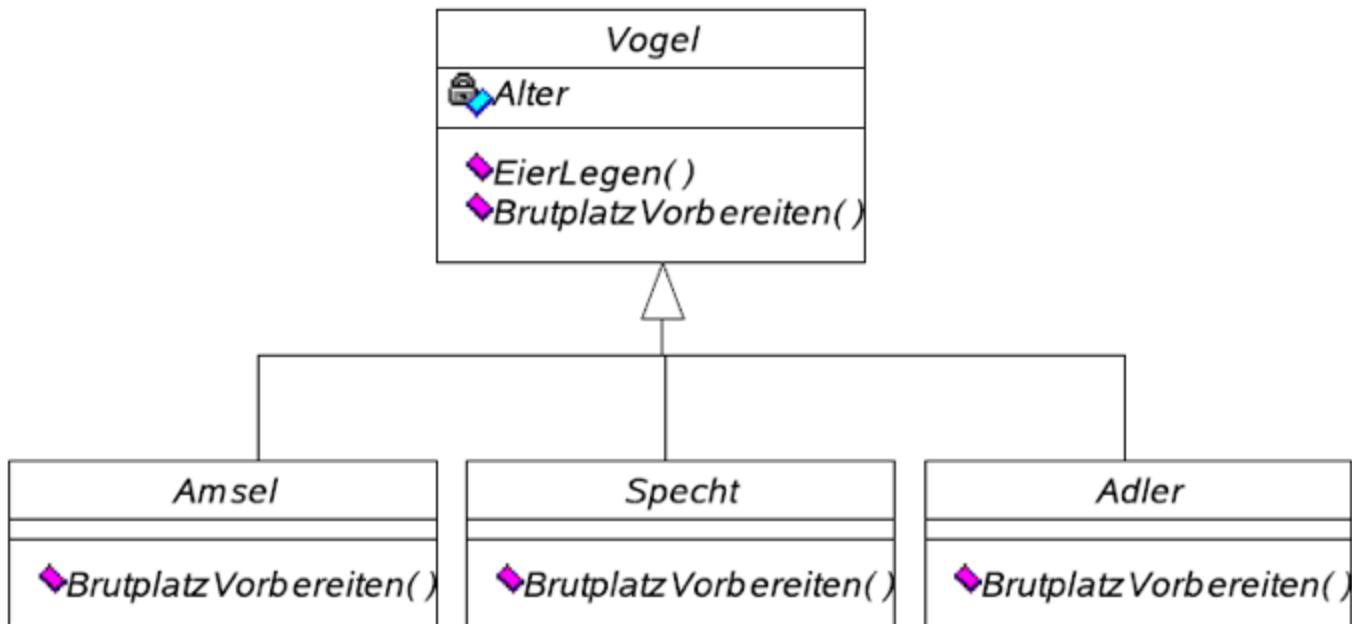
Bar erbt von Foo

- Bar ist eine Kindklasse/'child' von Foo
- Bar ist eine Unterklasse/'Subclass' von Foo
- Bar ist eine von Foo abgeleitete Klasse
- Foo ist die Elternklasse/'parent' von Bar
- Foo ist die 'superclass' von Bar
- Foo ist die 'base class' von Bar

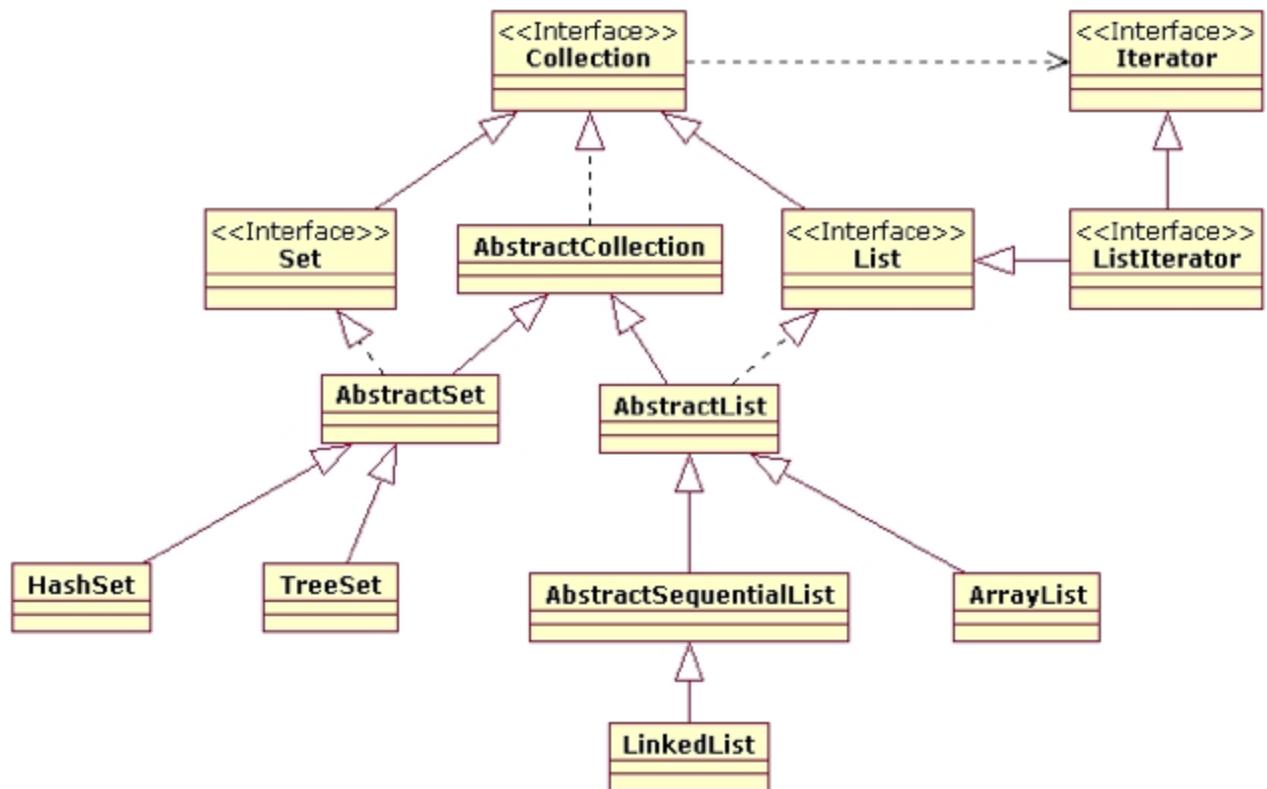
# Vererbung in Python

```
class Robot:  
    def __init__(self, name):  
        self.name = name  
  
    def say_hi(self):  
        print("Hi, I am " + self.name)  
  
class PhysicianRobot(Robot):  
    def say_hi(self):  
        print("Everything will be okay!")  
        print(self.name + " takes care of you!")  
  
r2d2 = Robot("r2d2")  
james = PhysicianRobot("James")  
james.say_hi()  
r2d2.say_hi()
```

# Vererbung: Beispiel I



# Vererbung: Beispiel II



## Liskovsches Substitutionsprinzip

"Subtype Requirement: Let  $\phi(x)$  be a property provable about objects  $x$  of type  $T$ . Then  $\phi(y)$  should be true for objects  $y$  of type  $S$  where  $S$  is a subtype of  $T$ ."

$S$  ist ein Untertyp von  $T$ . Ein Objekt des Typs  $S$  sollte sich, wo ein Objekt vom Typ  $T$  erwartet wird, gleich verhalten wie ein Objekt des Typs  $T$ .

# Polymorphismus

- Bis jetzt war bei einer Zuweisung der Ausdruck rechts immer vom gleichen Typ wie das Ziel links: `ziel = ausdruck`
- Mit Polymorphismus kann der Ausdruck rechts auch eine Unterklasse vom Typ des Ziels sein.
- Das gilt auch bei Argumenten von Methoden
- Variablen, Felder und Parameter von Methoden sollten möglichst einen allgemeinen Datentyp haben (Interface)

# Dynamisches Binden

Es können mehrere Methoden mit demselben Namen existieren:

- Durch Vererbung
- Durch verschiedene Methodesignaturen (unterschiedliche Anzahl oder Typen der Argumente)

Bei einem Methodenaufruf wird immer die bestgeeignete Methode ausgewählt.

# Bindung und Typen

Für einen Methodenaufruf `x.f()` :

- Statische Typisierung: Es gibt mindestens eine Version der Methode f
- Dynamische Typisierung: Während der Laufzeit wird geprüft ob f existiert
- Dynamische Bindung: Jeder Aufruf verwendet die best passende Version von f  
→ Methode des Objekts, nicht des Typs

## Vererbung: Coupling

Durch Vererbung werden zwei Klassen sehr eng gekoppelt (coupling). Sie sind dadurch stark voneinander abhängig. Das kann bei Änderungen zu Problemen führen.

## Vererbung: Zusammenfassung

- Datentypen können gruppiert und geordnet werden
- Neue Klassen können Bestehende erweitern
- Dynamisches Binden: Automatische Auswahl der korrekten Methode

# Grundlegende O-O Prinzipien

- Vererbung
- Polymorphismus
- Dynamische / statische Bindung
- Dynamische / statische Typisierung
- Generische Programmierung

# SOLID Principles

# Single Responsibility Principle

- "A module should be responsible to one, and only one, actor." The term actor refers to a group (consisting of one or more stakeholders or users) that requires a change in the module.
- "A class should have only one reason to change"

[wikipedia](#)

## **Open Closed Principle**

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

[wikipedia](#)

# Liskov's Substitution Principle

# Interface Segregation Principle

# **Dependency Inversion Principle**

# Dependency Inversion

```
const faker = new Faker();
let board = new BoardObject();
const logEventGateway = new LogEventGateway();
const loopbackEventGateway = new LoopbackEventGateway();
const ablyEventGateway = new AblyEventGateway();
const eventHandler = new EventHandler(loopbackEventGateway);
const boardEventFactory = new EventFactory(config['Board']);
board = new EventDispatcher(board, eventHandler, boardEventFactory);
faker.populateBoard(board);
```

# Clean Code

<https://cleancoders.com/>

Clean Code: A Handbook of Agile Software Craftsmanship

## Bezeichner

There are only two hard things in Computer Science: cache invalidation and naming things.

-- Phil Karlton

# Bezeichner

- Zweck erkennbar
- Keine Falschinformation
- Unterscheidbar
- Aussprechbar
- Suchbar
- Klassen: Nomen
- Methoden: Verben
- Länge dem Scope entsprechend

# Funktionen

- Kurz!
- Machen nur etwas
- Keine Nebenwirkungen
- Höchstens 3 Parameter
- Don't Repeat Yourself

# Kommentare

- Code sollte selbsterklärend sein
- Informativ
- Absicht erklären
- Erläuterung
- Warnung
- Todo

# Design Patterns

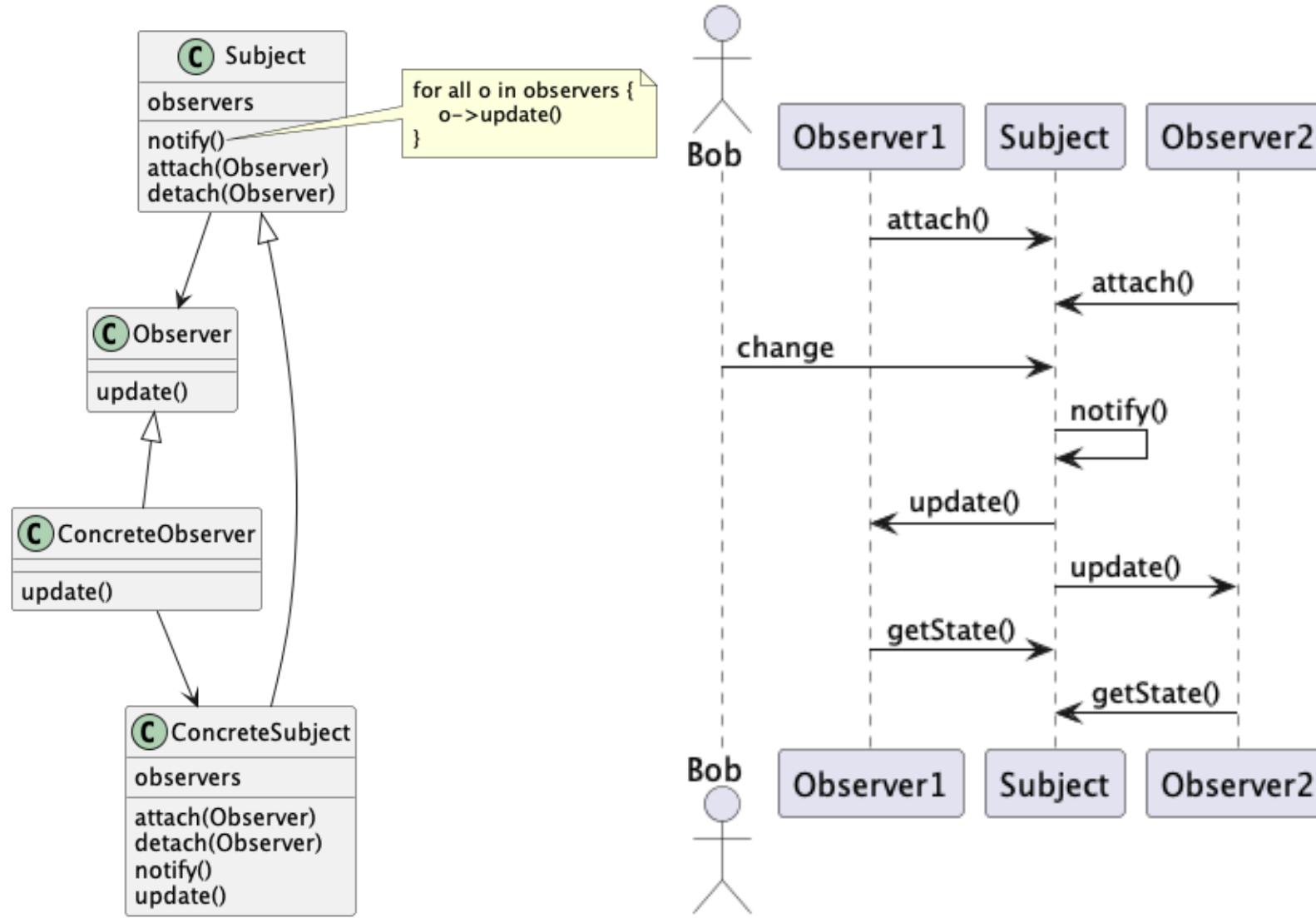
Gang of Four:

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995): Design Patterns,  
Elements of Reusable Object-Oriented Software, Addison-Wesley

Fowler, Martin (2002): Patterns of Enterprise Application Architecture, Addison-Wesley

Hohpe, Gregor; Woolf, Bobby (2003): Enterprise Integration Patterns: Designing, Building,  
and Deploying Messaging Solutions, Addison-Wesley

# Observer



# Fehlerbehandlung

- Es gibt Bedingungen, die erfüllt werden müssen, damit eine Methode überhaupt korrekt funktionieren kann.
- Oftmals gibt es beim Nichterfüllen kein sinnvolles weiteres Vorgehen, es handelt sich um einen Fehler.

# Arten von Fehlerbehandlung

- Fehler über Rückgabewerte zu kommunizieren funktioniert nur sinnvoll, wenn mehrere Rückgabewerte möglich sind: (Go)

```
swagger, err := api.GetSwagger()
if err != nil {
    fmt.Fprintf(os.Stderr, "Error loading swagger spec\n: %s", err)
    os.Exit(1)
}
```

- Viele Sprachen unterstützen das Konzept der "Exceptions":

```
if ( !(new.target) ) {
    throw new Error("Constructor called as a function");
}
```

# Exceptions

- Ausnahmen (Fehler) werden beim Auftreten geworfen (throw) und können gefangen (catch) werden.
- Exceptions werden weitergereicht bis sie gefangen werden.
- Werden sie bis zur `main` Methode nicht gefangen, stürzt das Programm ab.
- Exceptions, die im normalen Programmablauf auftreten können (z.B. Fehlerhafter User Input, Netzwerkverbindung offline) müssen gefangen und behandelt werden.
- Exceptions aufgrund von einem Programmierfehler sollten nicht gefangen werden.
- Code für die Fehlerbehandlung sollte möglichst vom Code der Funktionalität getrennt werden.

# Exceptions in Python

- Werfen von Exceptions: `raise Exception('<error message>')`
- Fangen von Exceptions:

```
try:  
    foo() ## method that might raise an exception  
except:  
    ## handle exception
```

# Exceptions in Go

Exceptions können es schwierig machen, den Programmablauf nachzuvollziehen, weil Exceptions den normalen Programmablauf unterbrechen.

In Go müssen, anders als in anderen Sprachen, Fehler als Rückgabewert explizit angegeben werden:

```
func (p Eurobox) setWeight(weight int) error {
    if weight <= 0 {
        return errors.New("Weight must be greater than zero")
    }
    p.weight = weight
    return nil
}
```

# Computer Hardware

# Moore's Law

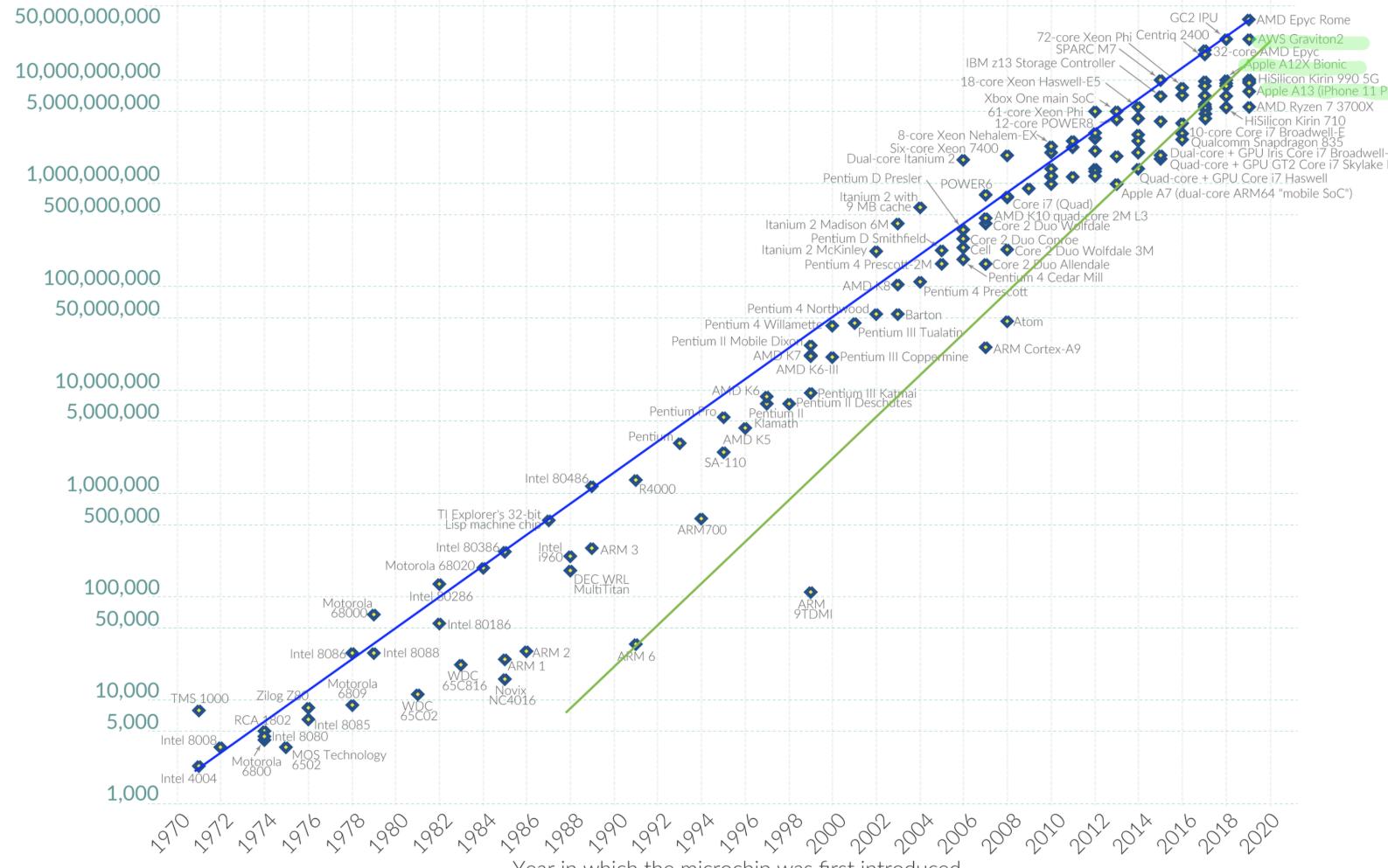
Moore's Law: The number of transistors on microchips doubles every two years

Our World  
in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.

This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

## Transistor count



Data source: Wikipedia ([wikipedia.org/wiki/Transistor\\_count](https://en.wikipedia.org/w/index.php?title=Transistor_count&oldid=910000000))

OurWorldInData.org – Research and data to make progress against the world's largest problems.

Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

# Alan Touring

- \*1912 - †1954
- Begründer der «Computer Science»
- WWII: Massgeblich am Knacken der deutschen Enigma-Verschlüsselung beteiligt
- Verfolgt und «therapiert» wegen Homosexualität
- Vermutlich Selbstmord
- The Imitation Game, Benedict Cumberbatch
-

# Turingmaschine

Die Turingmaschine hat ein Steuerwerk, in dem sich das Programm befindet, und besteht außerdem aus

- einem unendlich langen Speicherband mit unendlich vielen sequentiell angeordneten Feldern.
- einem programmgesteuerten Lese- und Schreibkopf, der sich auf dem Speicherband feldweise bewegen und die Zeichen verändern kann.
- Turing bewies, dass solch ein Gerät in der Lage ist, „jedes vorstellbare mathematische Problem zu lösen, sofern dieses auch durch einen Algorithmus gelöst werden kann“.

# Turingvollständigkeit

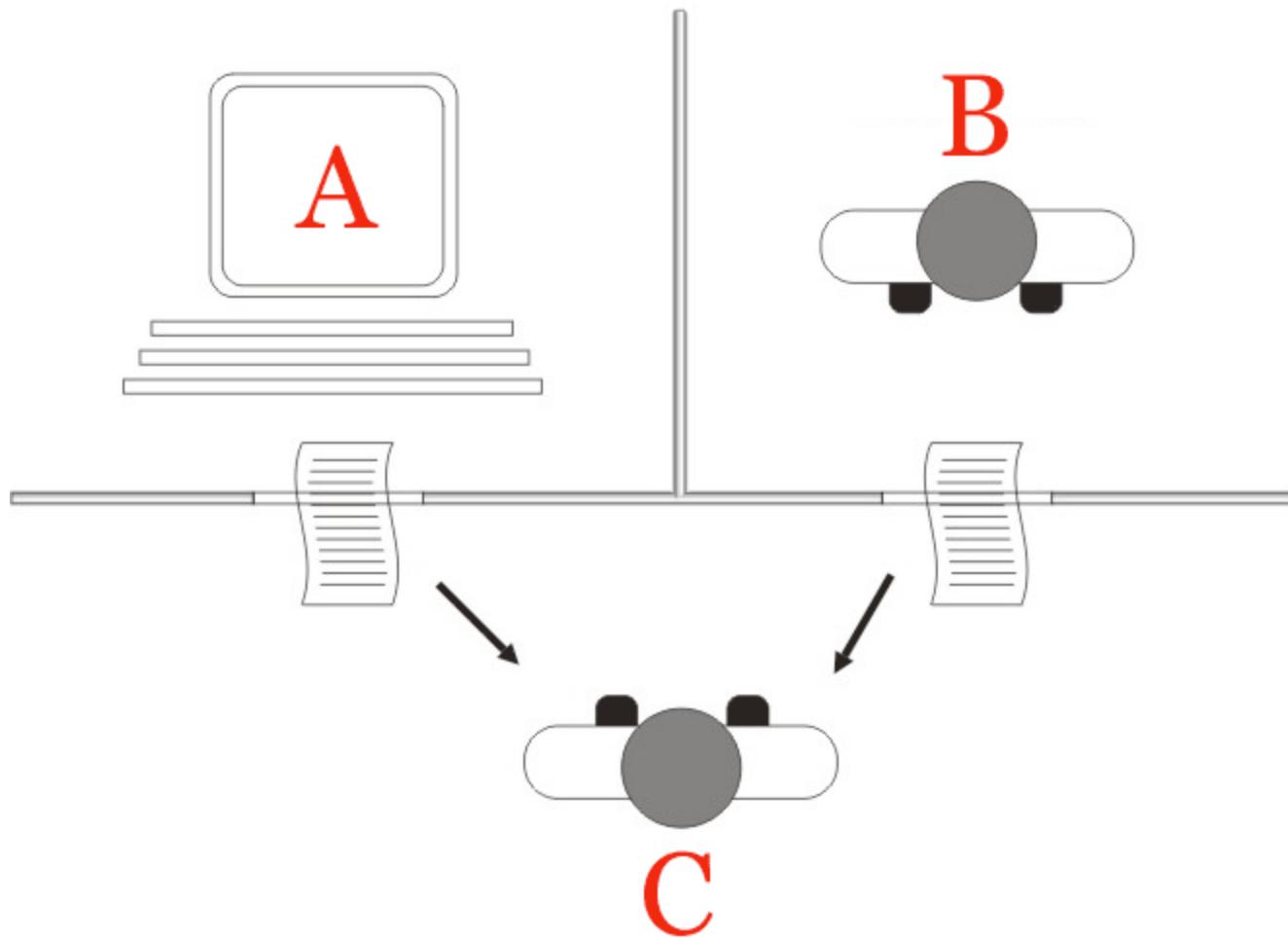
«Exakt ausgedrückt bezeichnet Turing-Vollständigkeit in der Berechenbarkeitstheorie die Eigenschaft einer Programmiersprache oder eines anderen logischen Systems, sämtliche Funktionen berechnen zu können, die eine universelle Turingmaschine berechnen kann.»  
[\(https://de.wikipedia.org/wiki/Turing-Vollständigkeit\)](https://de.wikipedia.org/wiki/Turing-Vollständigkeit)

## Halteproblem

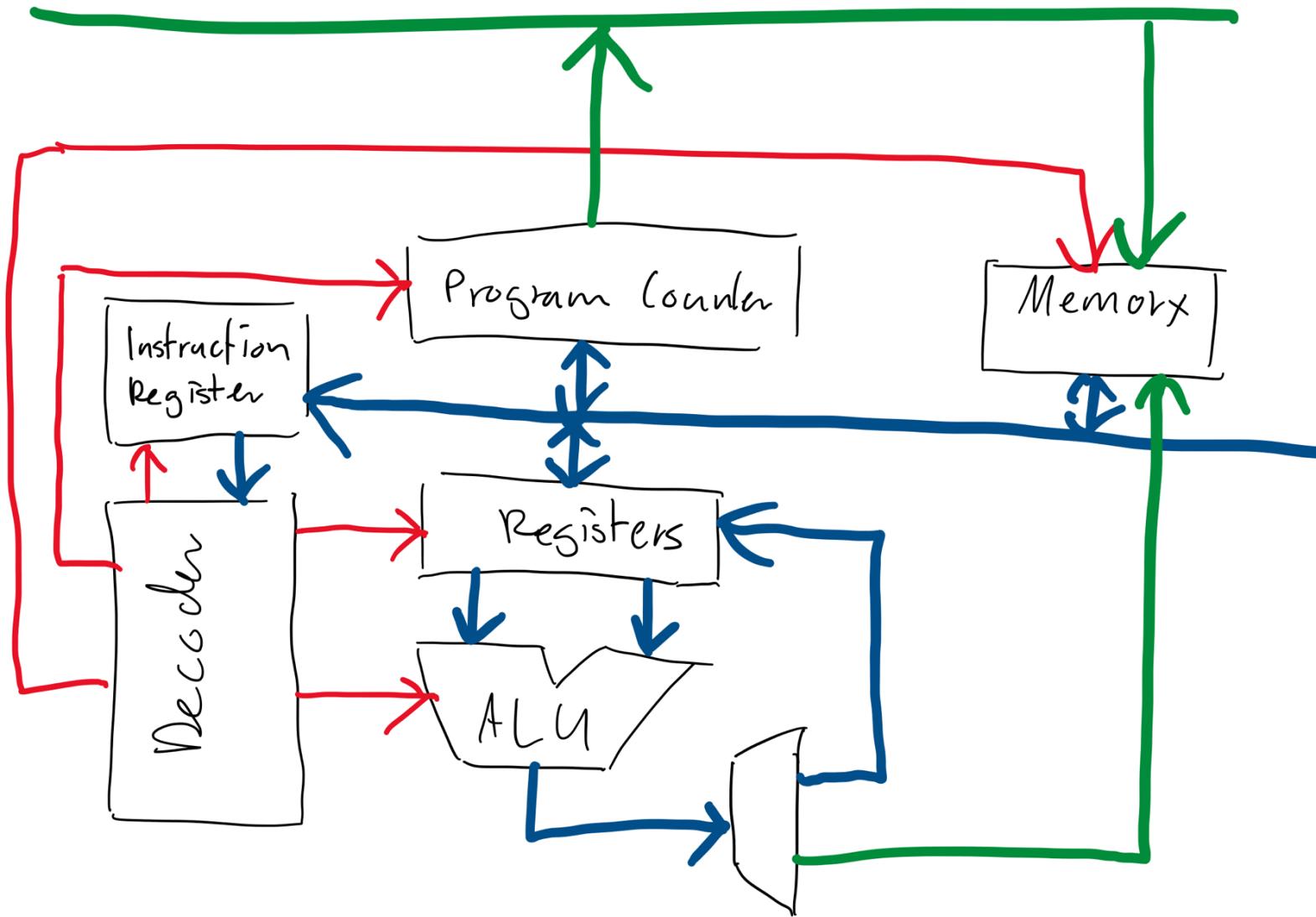
«Das Halteproblem beschreibt die Frage, ob die Ausführung eines Algorithmus zu einem Ende gelangt. Obwohl das für viele Algorithmen leicht beantwortet werden kann, konnte der Mathematiker Alan Turing beweisen, dass es keinen Algorithmus gibt, der diese Frage für alle möglichen Algorithmen und beliebige Eingaben beantwortet.»  
[\(https://de.wikipedia.org/wiki/Halteproblem\)](https://de.wikipedia.org/wiki/Halteproblem)

Wir müssen sicherstellen, dass unsere Programme nicht unabsichtlich endlos weiterlaufen!

# Turing-Test



# Von-Neumann-Architektur



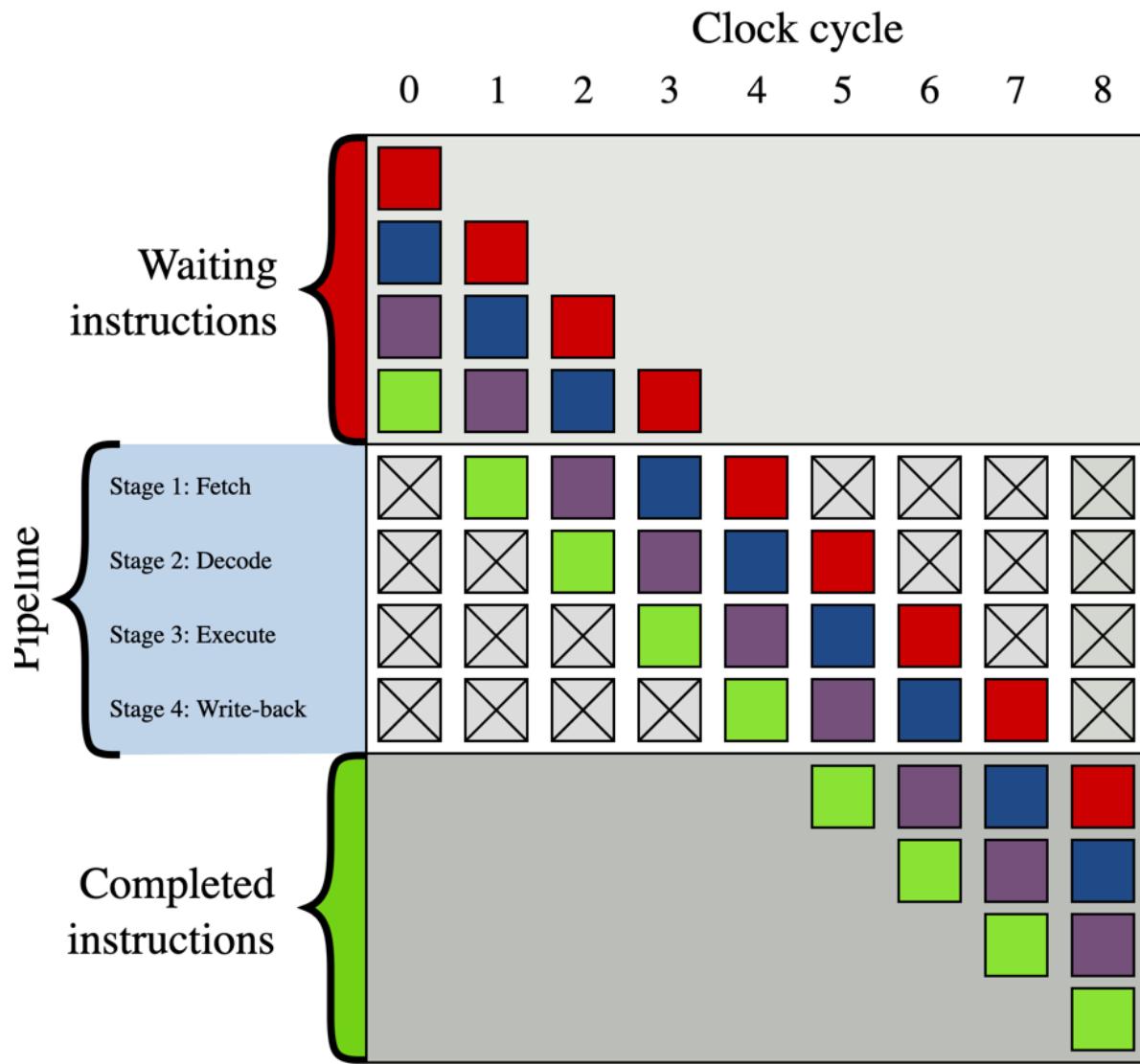
## Von-Neumann-Architektur

- Befehle werden aus einer Zelle des Speichers gelesen und dann ausgeführt.
- Normalerweise wird dann der Inhalt des Befehlszählers um Eins erhöht.
- Es gibt Verzweigungs-Befehle, die in Abhängigkeit vom Wert eines Entscheidungs-Bit den Befehlszähler um Eins erhöhen oder um einen anderen Wert verändern

# Instruction Cycle

- Fetch
- Decode
- Execute

# Instruction Pipelines



# Arithmetic Logic Unit (ALU)

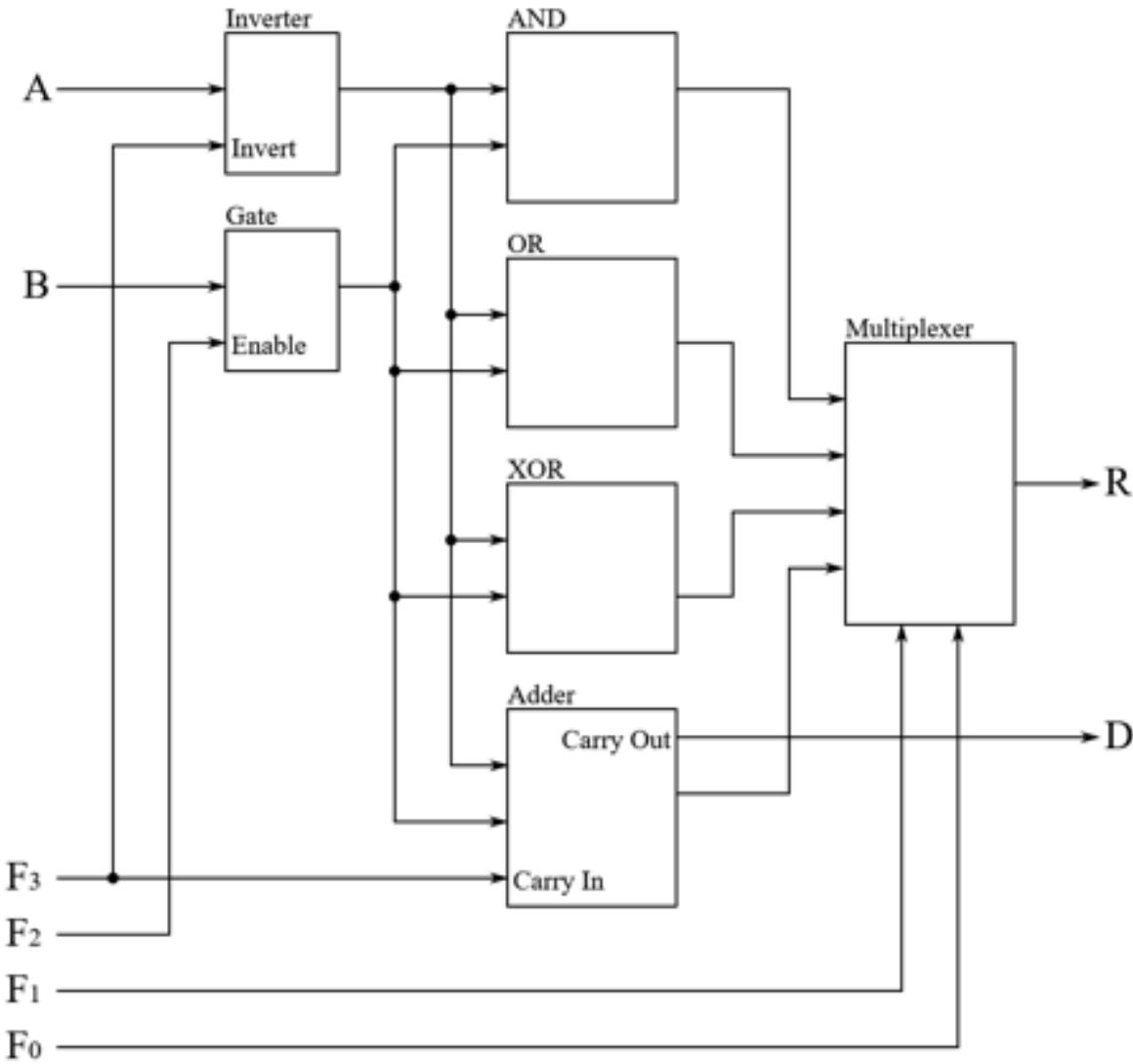
Mindestens:

- Addition
- Negation
- Konjunktion (AND)

Zusätzlich (Auswahl):

- Subtraktion
- Vergleich
- Multiplikationen
- Division
- (Exklusiv-)Oder-Verknüpfung
- Rechts-, Links-Shift

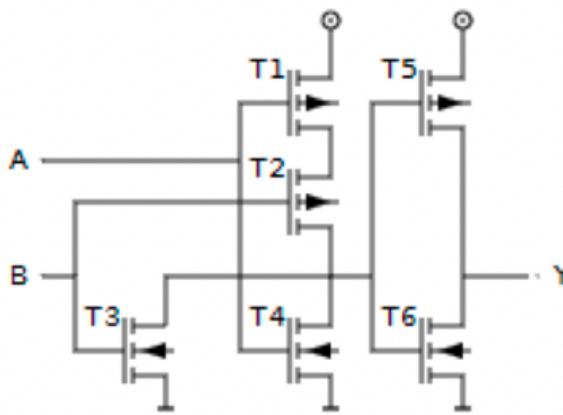
# Einfache n-Bit ALU



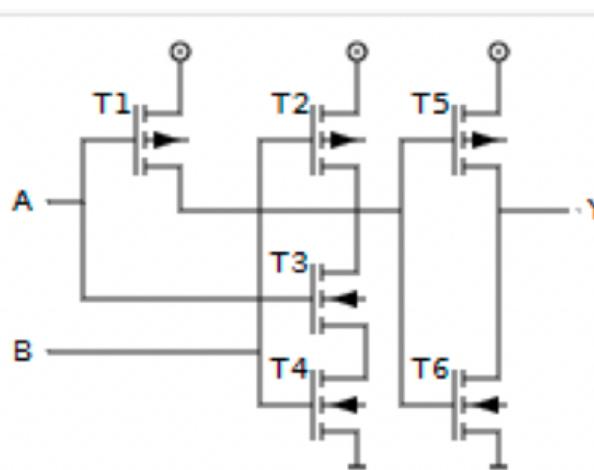
Steuertabelle für n-Bit ALU

$F_3$	$F_2$	$F_1$	$F_0$	R
0	0	0	0	0
0	0	0	1	A
1	0	0	1	NOT A
0	1	0	0	A AND B
0	1	0	1	A OR B
0	1	1	0	A XOR B
0	1	1	1	A + B
1	1	1	1	B - A

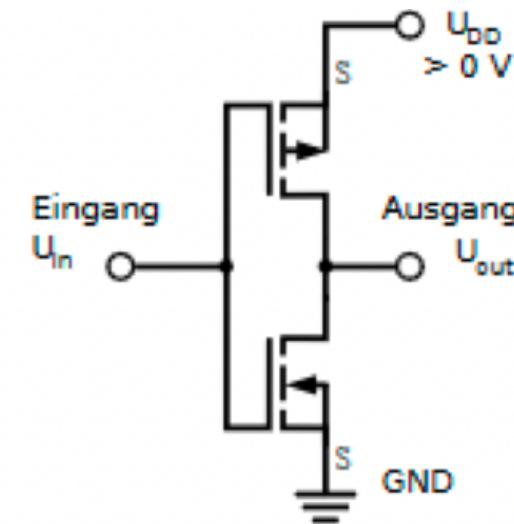
# CMOS Gatter



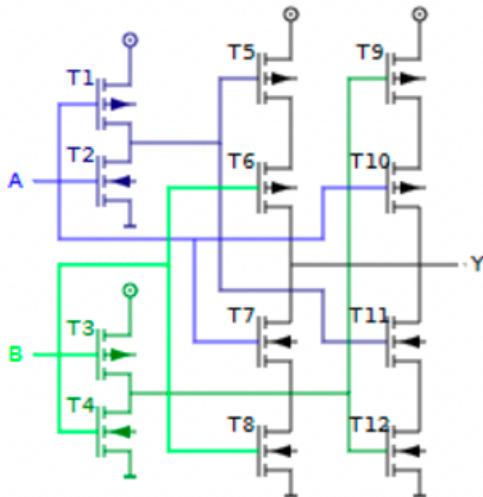
OR



AND



NOT



XOR

# AMD Zen 3

## “ZEN 3” OVERVIEW

2 THREADS PER CORE (SMT)

STATE-OF-THE-ART BRANCH PREDICTOR

### CACHES

- I-cache 32k, 8-way
- Op-cache, 4K instructions
- D-cache 32k, 8-way
- L2 cache 512k, 8-way

### DECODE

- 4 instructions / cycle from decode or 8 ops from Op-cache
- 6 ops / cycle dispatched to Integer or Floating Point

### EXECUTION CAPABILITIES

- 4 integer units
- Dedicated branch and store data units
- 3 address generations per cycle

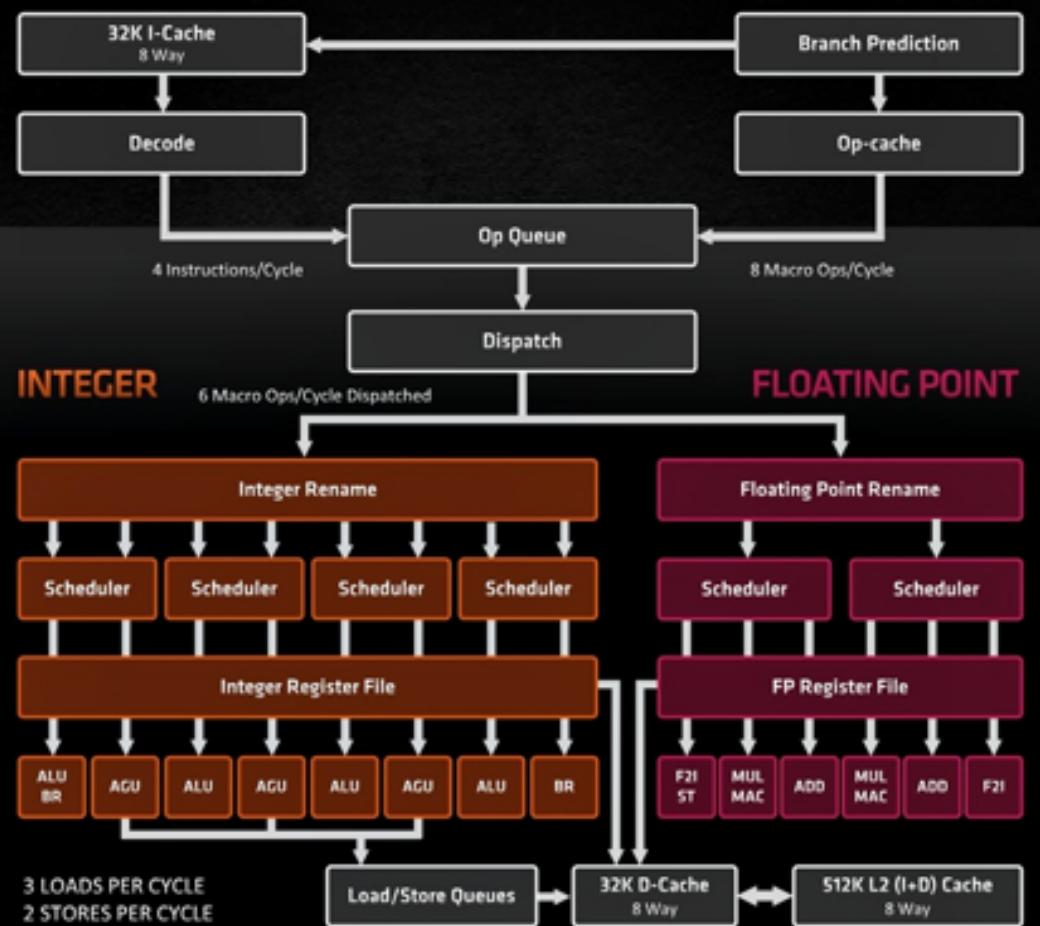
### 3 MEMORY OPS PER CYCLE

- Max 2 can be stores

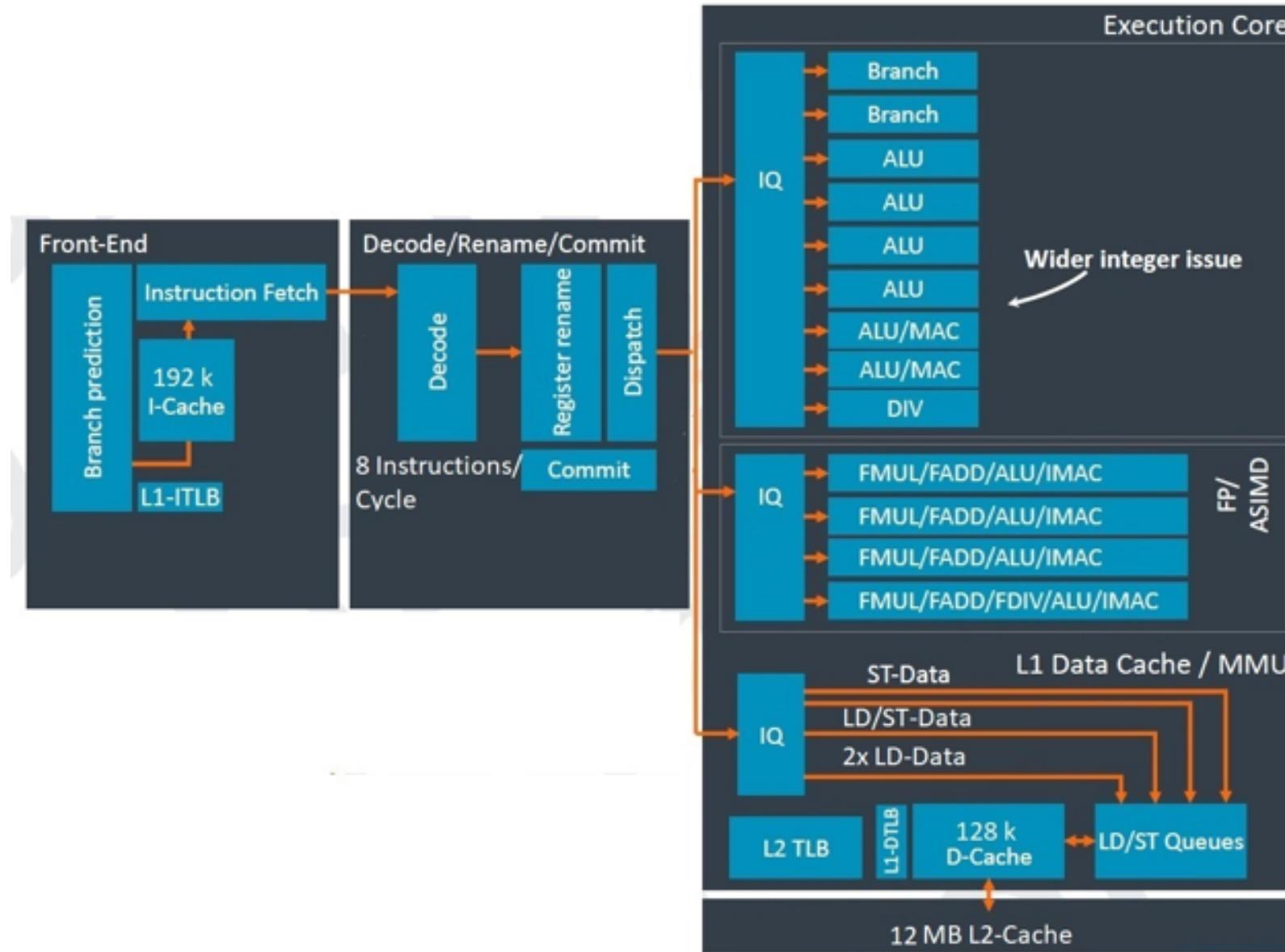
### TLBs

- L1 64 entries I & D, all page sizes
- L2 512 I, 2K D, everything but 1G

### TWO 256-BIT FP MULTIPLY ACCUMULATE / CYCLE

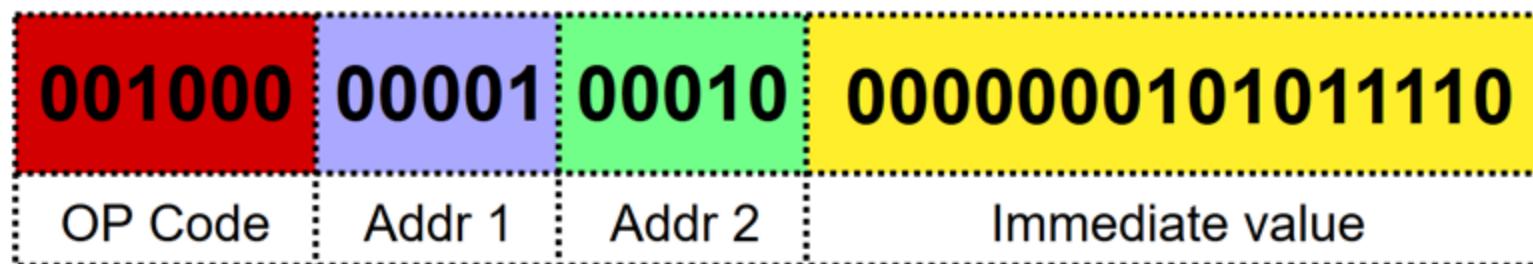


# Apple M1



# Maschinensprache (1. Generation)

MIPS32 Add Immediate Instruction



Equivalent mnemonic:

**addi \$r1 , \$r2 , 350**

# RISC vs CISC

## RISC

- Ausführung der Instruktionen dauert meistens nur 1 Takt
- Nur spezifische Load und Store Befehle greifen auf den Speicher zu
- Grosse Anzahl Register
- Befehle mit fester Länge

## CISC

Alles was nicht RISC ist.

# RISC

- Besser geeignet für "moderne" Compiler
- Intel / AMD haben lange den CPU Markt mit CISC CPUs dominiert
- Im mobile und embedded Bereich ist ARM (RISC) extrem verbreitet
- Seit 2020 gibt es auch im Desktop wieder RISC Systeme (Apple M1) mit grossen Vorteilen in der Effizienz
- Verschiedene Hersteller bieten auch für RISC Server-CPUs an die v.a. bei Cloud Anbietern (AWS, Google, etc) Verbreitung finden

# Assembler (2. Generation)

```
;  
; This program runs in 32-bit protected mode.  
; build: nasm -f elf -F stabs name.asm  
; link: ld -o name name.o  
  
; In 64-bit long mode you can use 64-bit registers (e.g. rax instead of eax, rbx instead of ebx, etc.)  
; Also change "-f elf" for "-f elf64" in build command.  
  
section .data ; section for initialized data  
str: db 'Hello world!', 0Ah ; message string with new-line char at the end (10 decimal)  
str_len: equ $ - str ; calcs length of string (bytes) by subtracting the str's start address  
; from this address ($ symbol)  
  
section .text ; this is the code section  
global _start ; _start is the entry point and needs global scope to be 'seen' by the  
; linker --equivalent to main() in C/C++  
_start: ; definition of _start procedure begins here  
    mov eax, 4 ; specify the sys_write function code (from OS vector table)  
    mov ebx, 1 ; specify file descriptor stdout --in gnu/linux, everything's treated as a file,  
; even hardware devices  
    mov ecx, str ; move start _address_ of string message to ecx register  
    mov edx, str_len ; move length of message (in bytes)  
    int 80h ; interrupt kernel to perform the system call we just set up -  
; in gnu/linux services are requested through the kernel  
    mov eax, 1 ; specify sys_exit function code (from OS vector table)  
    mov ebx, 0 ; specify return code for OS (zero tells OS everything went fine)  
    int 80h ; interrupt kernel to perform system call (to exit)
```

# Programmiersprachen der 3. Generation

- ALGOL
- Cobol
- Fortran
- C, C++
- C#
- Java
- Python
- etc.

## 4. Generation

- SQL
- Unix Shell
- LabVIEW
- Stata
- R
- MATLAB
- MaxMSP

# Zahlendarstellung

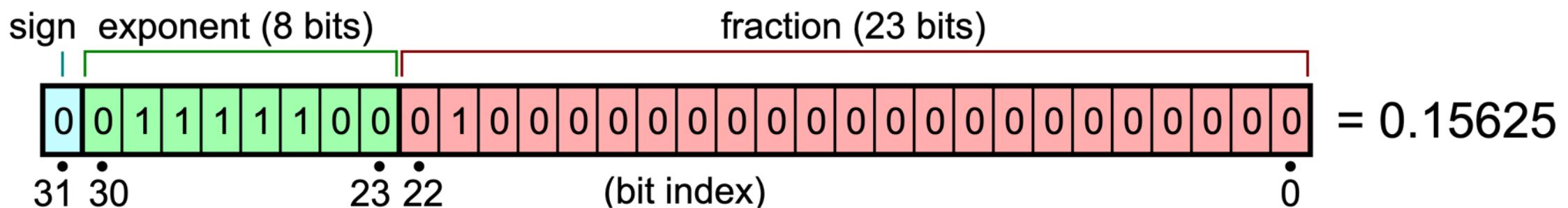
- Binäre Zahlen: Für Maschinen einfach darstellbar (2 mögliche Zustände, idR. Spannungen)

# Integer

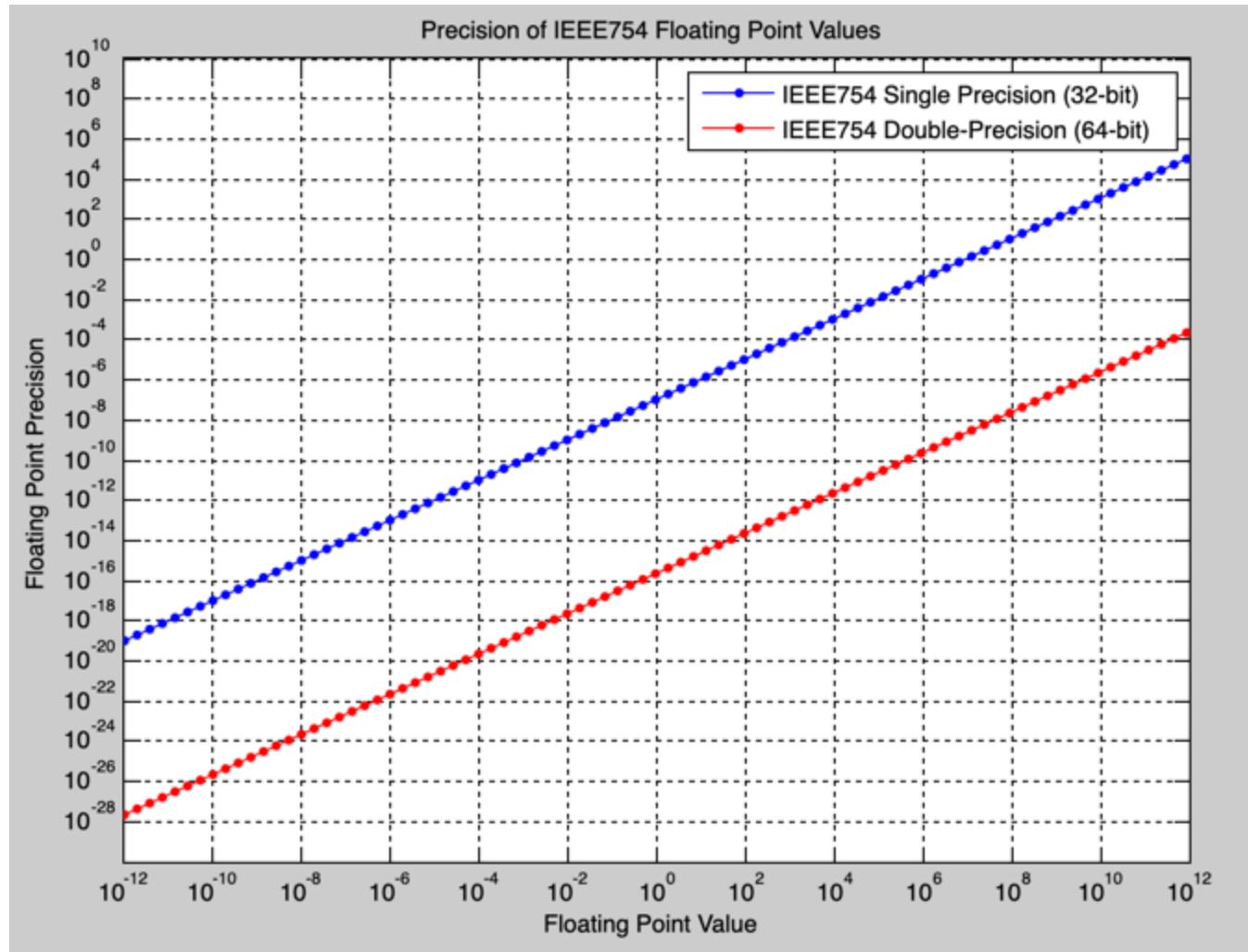
- Ganze Zahlen
- Natürliche Zahlen (Negativ): Das MSB (most significant bit) wird für das Vorzeichen verwendet

# Fliesskommazahlen

- Normiert in IEEE 754
- $x = s \cdot m \cdot b^e$ 
  - Vorzeichen s
  - Mantisse m
  - Basis b ( $b=2$ )
  - Exponent e



# Floating Point: Präzision



# Strings

- Array von Buchstaben (Char)

## ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[END OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	l					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	-					

## Datentypen in Go (Auswahl)

`bool` boolean, 1-bit, true or false

`int8` 8-bit signed integer (-128 bis 127)

`int16` 16-bit signed integer (-32'768 bis 32'767)

`int32` 32-bit signed integer (-2'147'483'648 bis 2'147'483'647)

`uint8` 8-bit unsigned integer (0 bis 255)

`float32` 32-bit IEEE 754 floating-point number (1.2E-38 bis 3.4E38)

`string` "Sequence of Unicode code points"

# Statische Typisierung

- Zur Laufzeit hat jedes Objekt einen (Daten)typ
- Im Programmtext hat jeder Ausdruck einen Typ → Der Typ ist zum Zeitpunkt der Kompilierung bekannt
- Vorteile
  - Fehler können früher erkannt werden
  - Effizientere Programme, da keine Typprüfung während der Laufzeit
  - Mehr Optimierungsmöglichkeiten durch Compiler
- statisch typisierte Sprachen: Java, Kotlin, C#, C, Go, Rust

```
final Crossroad crossroad = new Crossroad();
final CrossroadController crossroadController =
final Scene scene = new Scene(crossroadControll
```

# Datentypen in Python (Auswahl)

- str
- int (Kein Limit)
- float (64Bit IEEE 754))
- complex
- bool

# Dynamische Typisierung

- Zur Laufzeit hat jedes Objekt einen Typ
- Der Typ wird zur Laufzeit geprüft
- Duck Typing: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."
- Vorteile
  - Einfachere Programmierung
- Durch Typehints kann die IDE uns bei der Entwicklung dennoch unterstützen
  - `def greeting(name: str) -> str:`
- dynamisch typisierte Sprachen: PHP, Python, Ruby, JavaScript