

# Einstieg

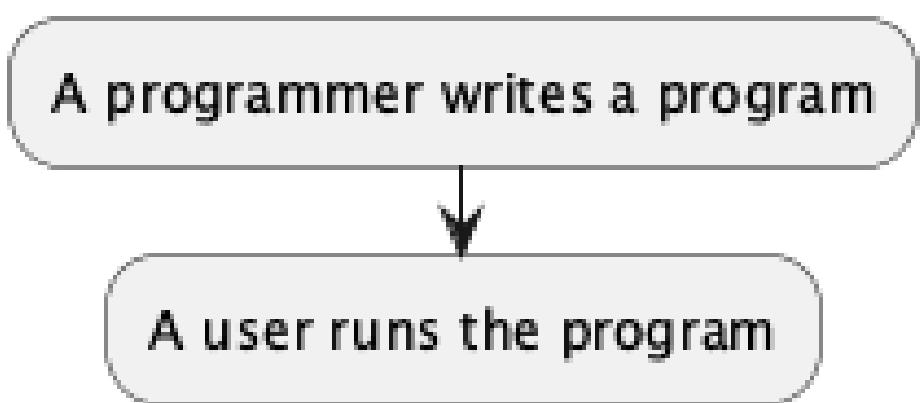
## **Softwareentwickler:innen bauen Maschinen**

- 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)

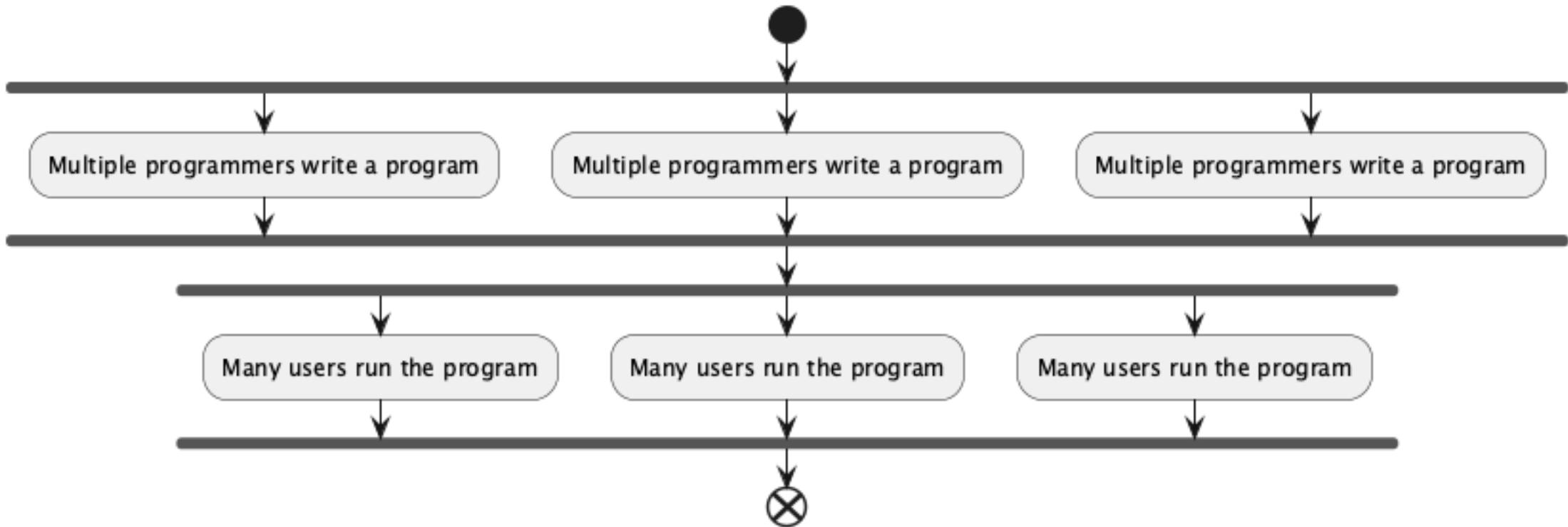
# Computer

- 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.

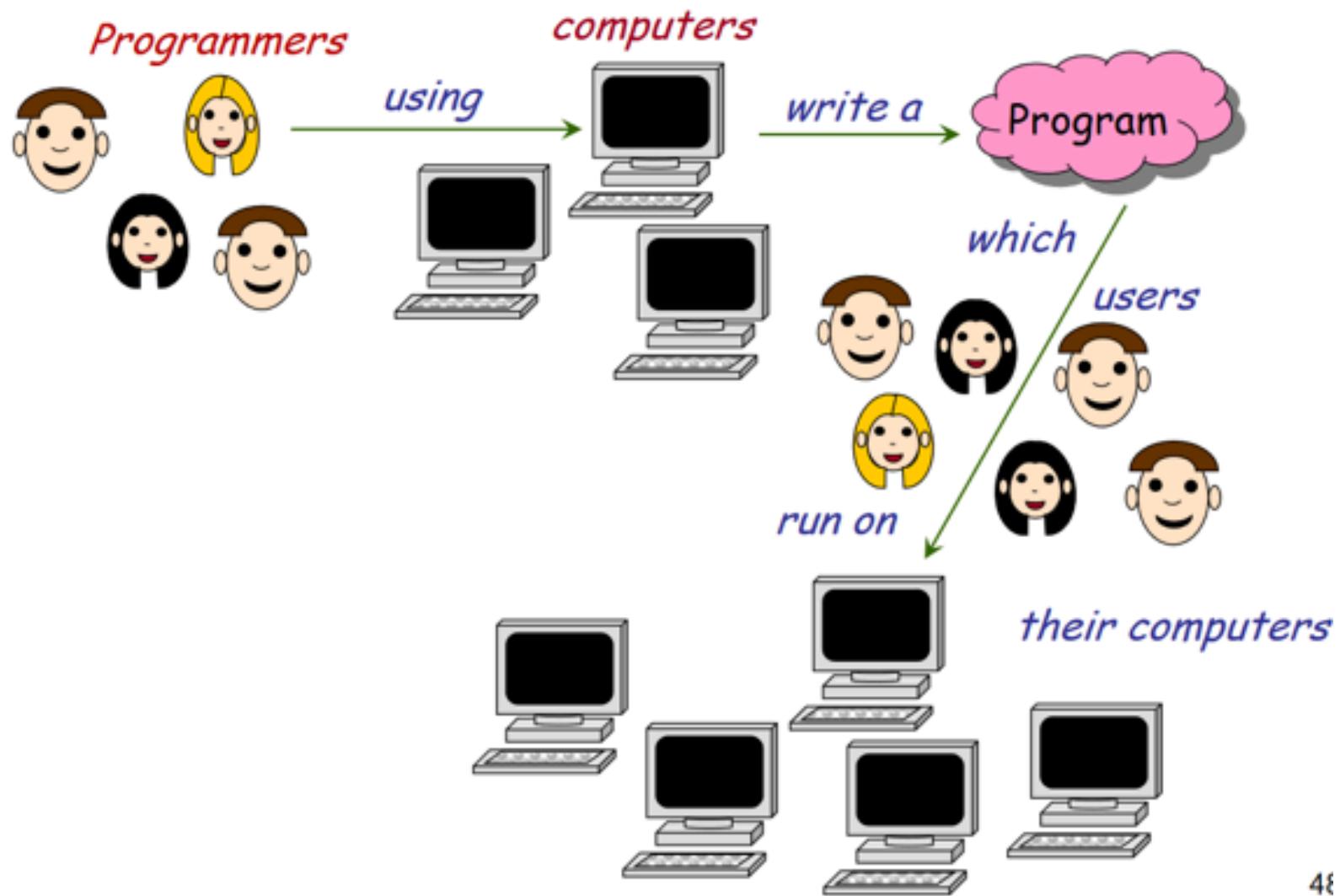
# Programme erstellen und laufen lassen



# 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»

# 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**

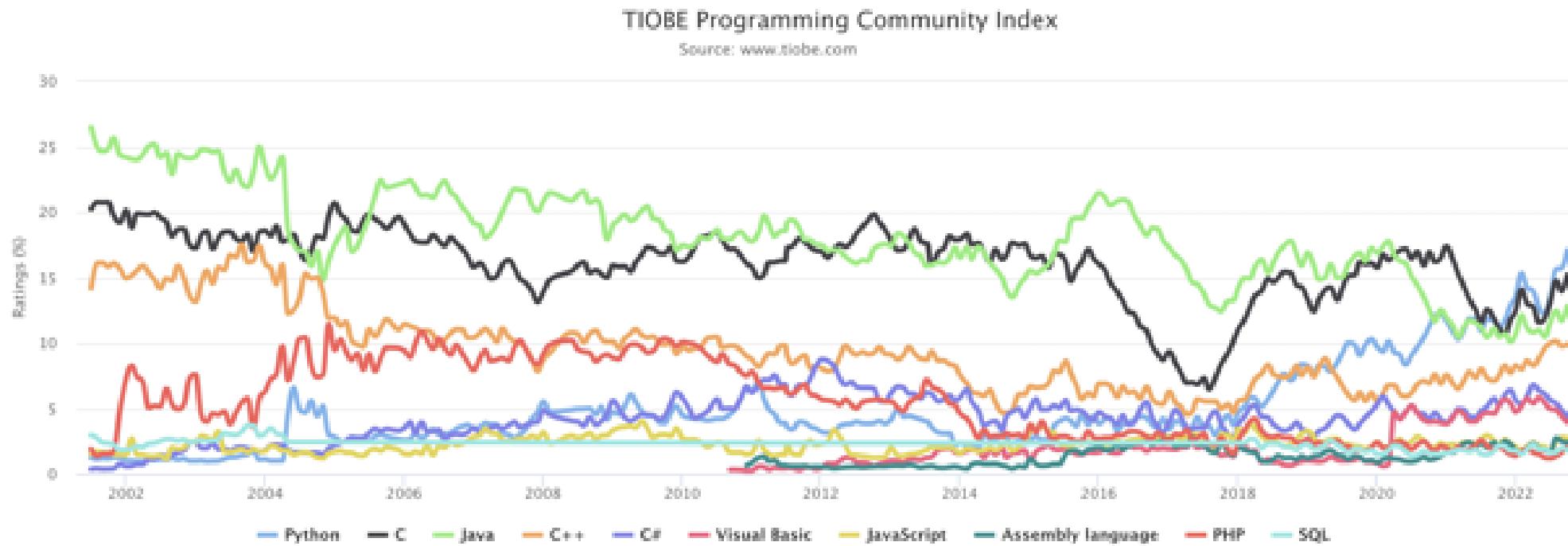
- Es ist schwierig, das Programm korrekt zu schreiben
- Trial-and-error ist ineffizient

## **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

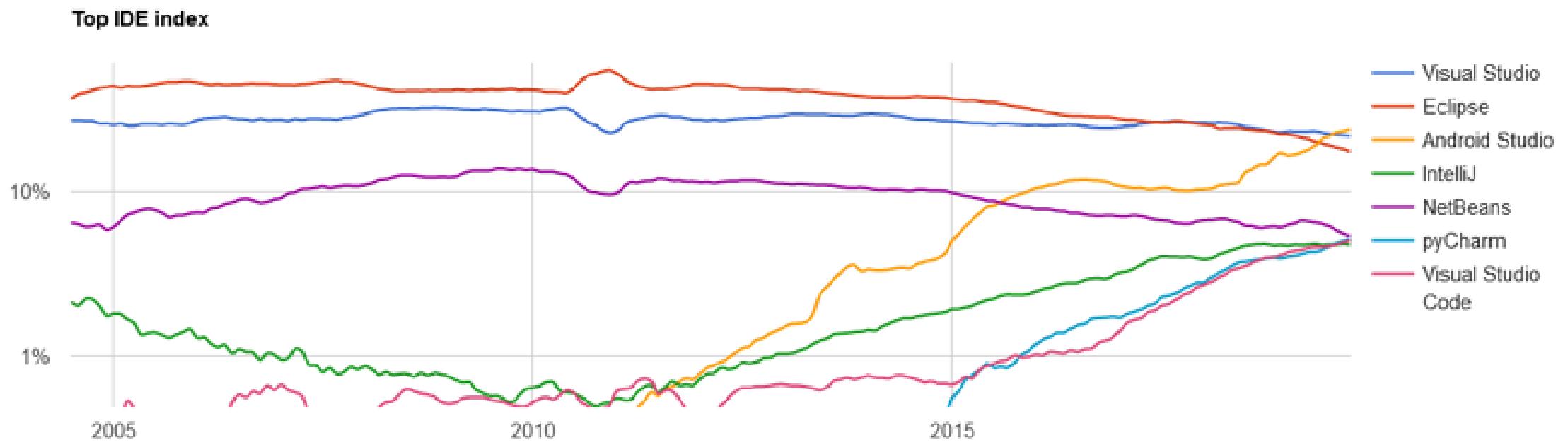
God-Tier Developer Roadmap

						Anwendungen
C	1972	Bell Labs	Kompiliert	statisch typisiert	imperativ, strukturiert	Betriebssysteme und Embedded
C++	1985	Bell Labs	Kompiliert	statisch typisiert	imperativ, objektorientiert	Betriebssysteme, Desktop Applikationen, Games, Datenbanken, Interpreter
.	1995	.	Kompiliert	statisch	imperativ,	.

# 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



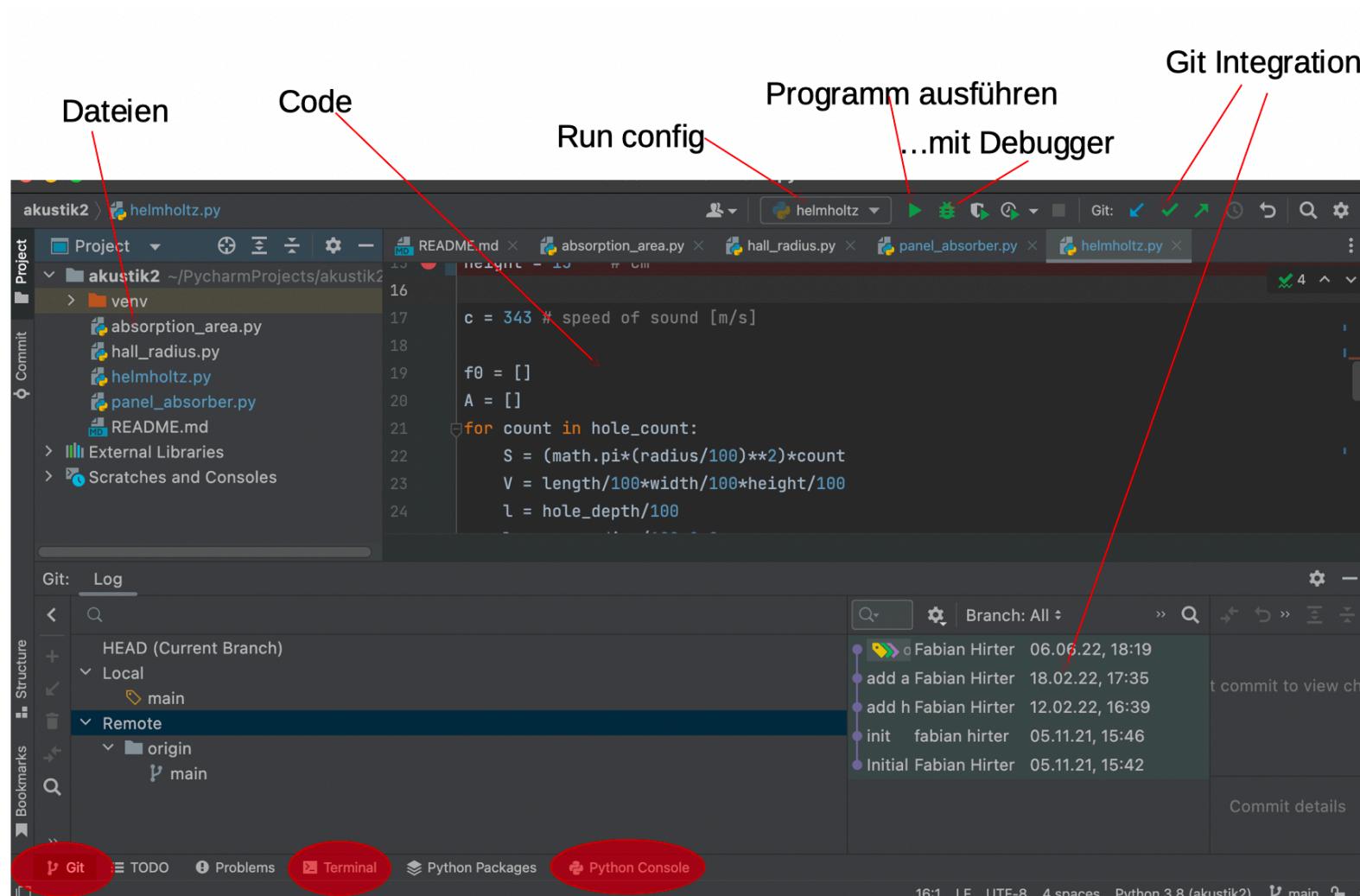
## **Microsoft Visual Studio Code**

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

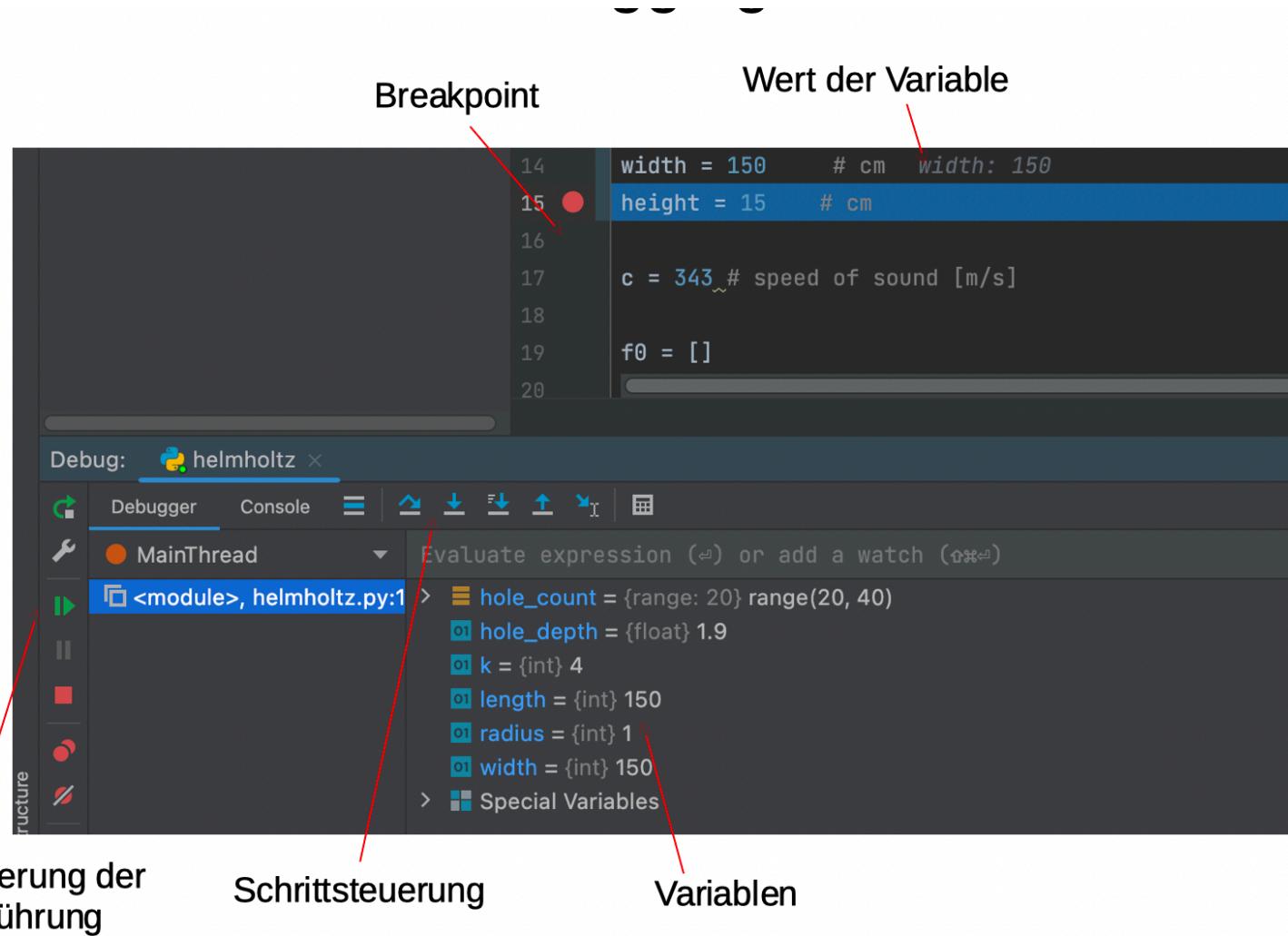
## JetBrains

- 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

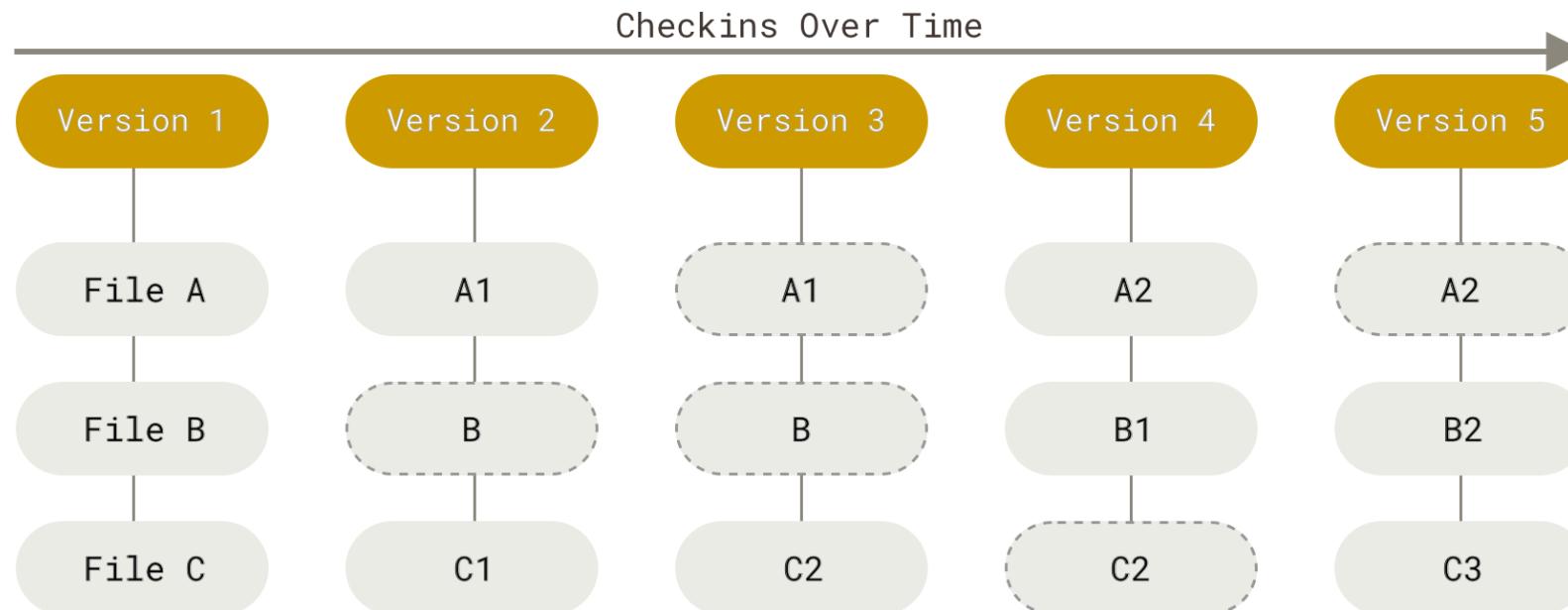
- Protokollierung von Änderungen
- Wiederherstellung von alten Ständen
- Archivierung
- Koordinierung des gemeinsamen Zugriffs
- Entwicklungszweige (Branches) -> **Don't Branch!**

# Moderne Versionsverwaltung

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

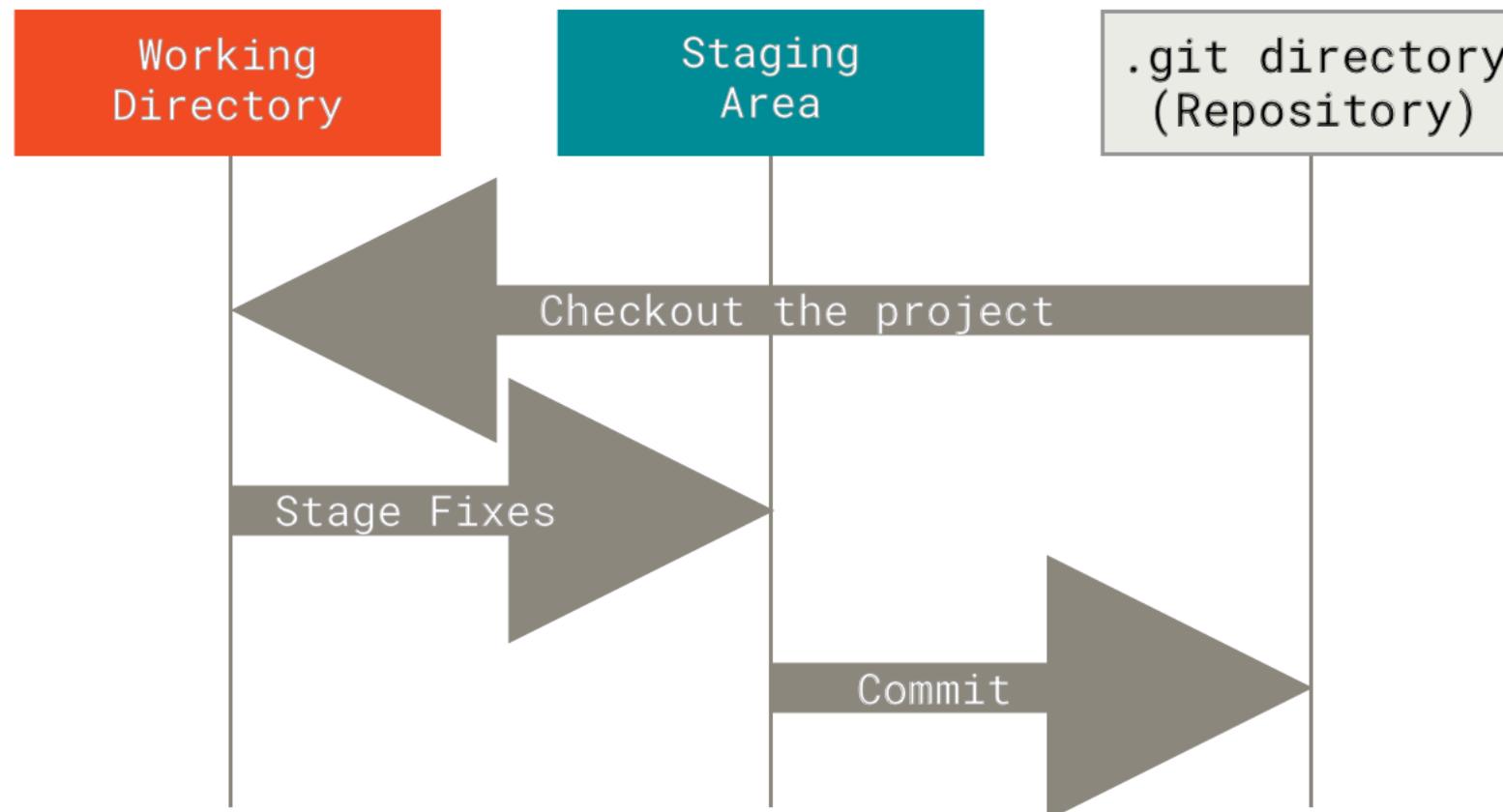
# Git

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



# 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

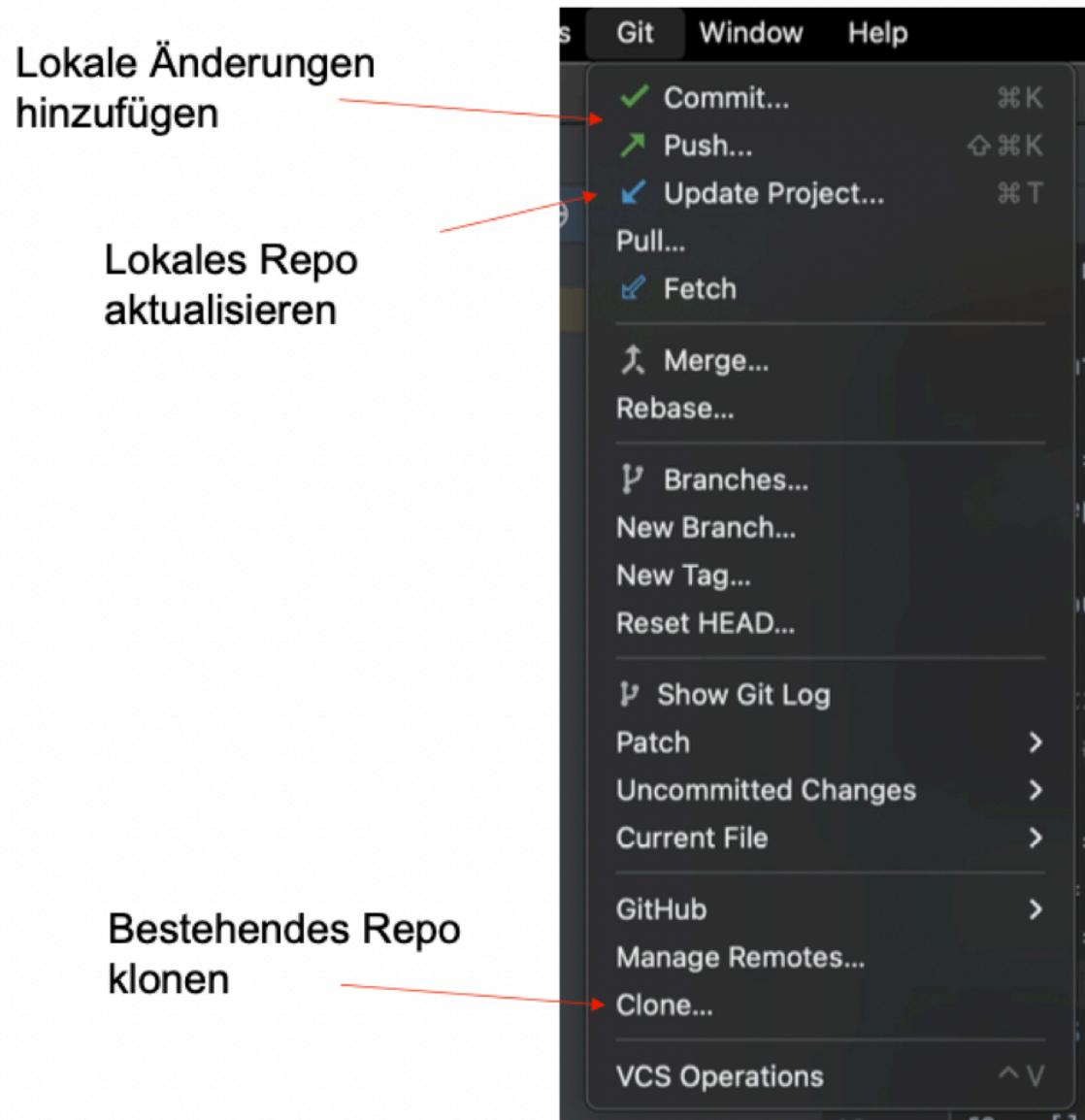
```
feat(logging): added logs for failed signups
```

```
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

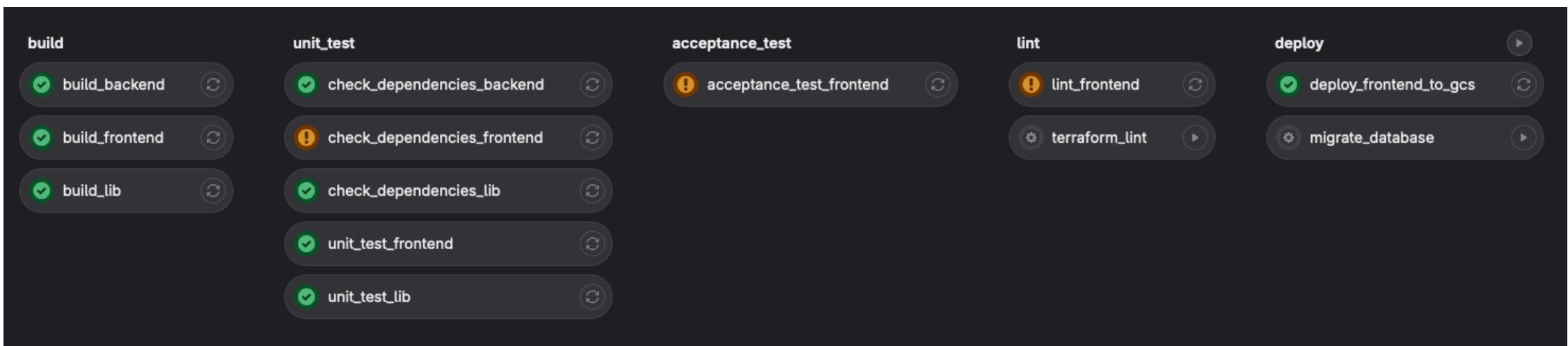
- [Cheatsheet](#)
- [Atlassian Tutorials](#)
- [Git Tutorials](#)
- [Simulationstool](#)

# 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 four columns: Open, Next, In Progress, and Closed. Each column contains a list of tasks with their respective labels and IDs.

- Open:** 27 items.
  - Parcel visualization (Security, #95)
  - Tags as out of ordinary visualization (#94)
  - Payment integration (#45)
  - Set Security Headers (Security, #91)
  - lane editing (Feature, low hanging fruit, #6)
- Next:** 6 items.
  - 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 items.
  - Multitenancy (Feature, #46)
  - compact mode (#8)
- Closed:** 30 items.
  - 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)



# 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
- Datenobjekte

## Beispiele

```
// C
struct date {
    int year;
    int month;
    int day;
}
```

```
# python
tup1 = ('physics', 'chemistry', 1997, 2000)
```

# **Set**

- Anordnung von Elementen
- keine Duplikate
- keine definierte Ordnung
- testen, ob Teil des Sets

## Beispiel

```
# python  
thisset = {"apple", "banana", "cherry"}
```

# List

- Definierte Ordnung
- Elemente hinzufügen und entfernen
- Element mit einem Index abrufen
- Duplikate möglich
-

## Beispiel

```
# python
list2 = [1, 2, 3, 4, 5, 6, 7]
first_item = list2[0] # select first item
list2[1:5] # select items 2 to 6
```

# Map

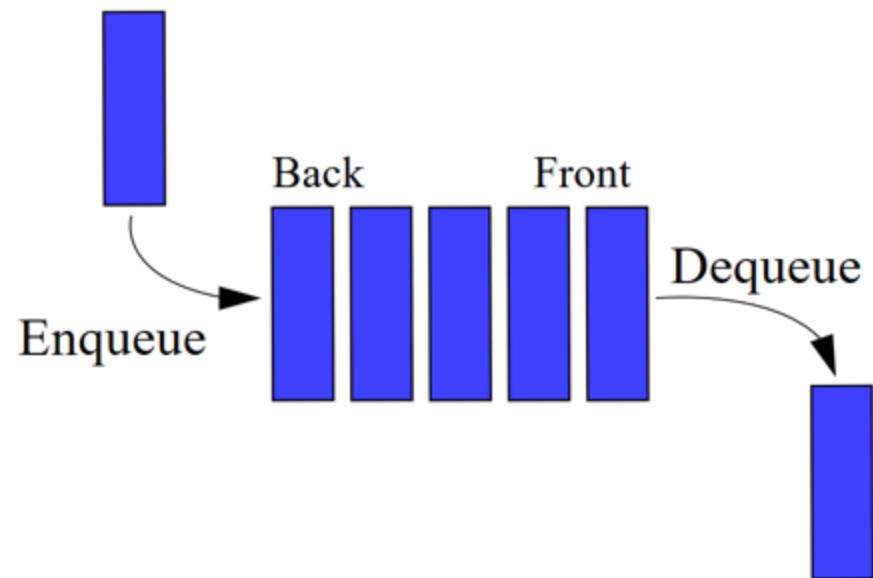
- Schlüssel / Wert Paare
- Hinzufügen, Entfernen, Ändern, Abrufen
- Assoziatives Array, Lookup Table, Dictionary

## Beispiel

```
# python
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Name'] # Zara
```

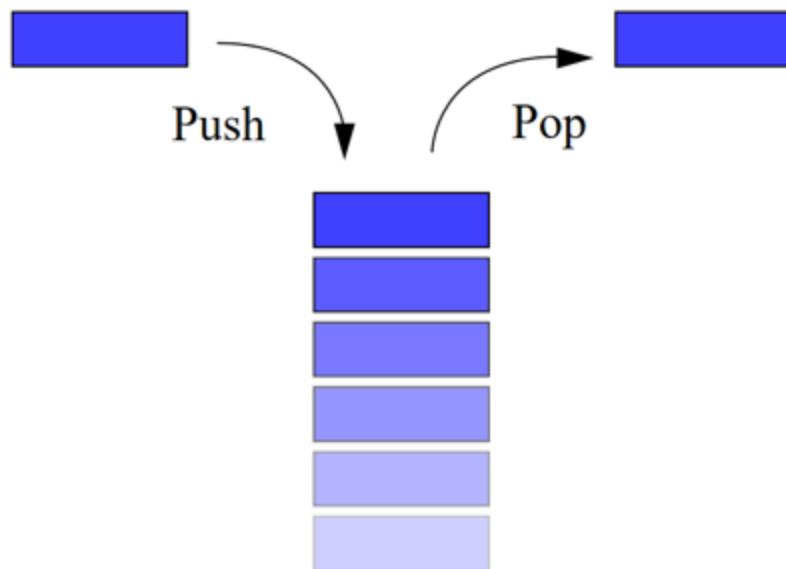
# Queue

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

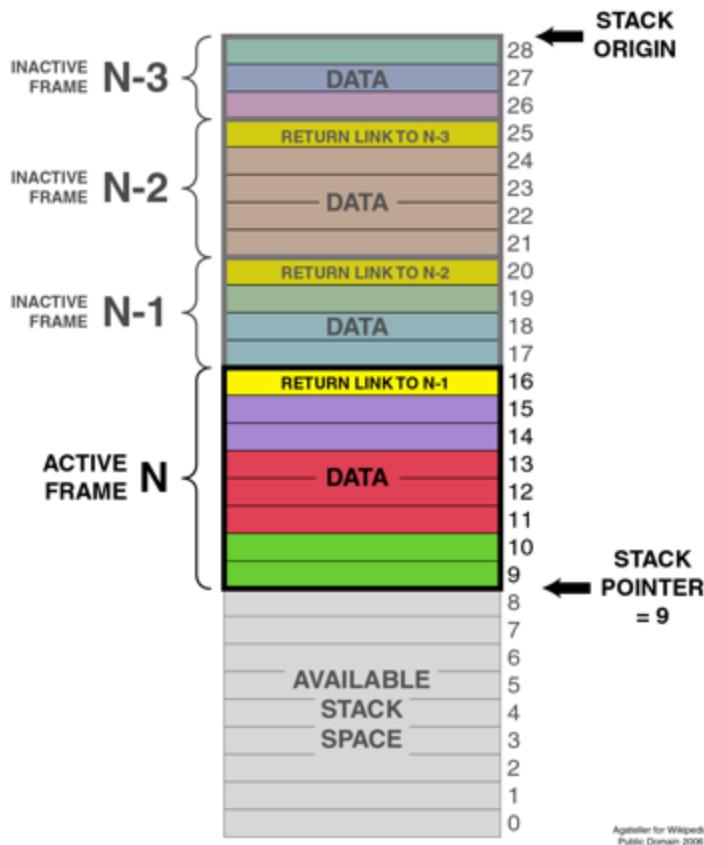


# Stack

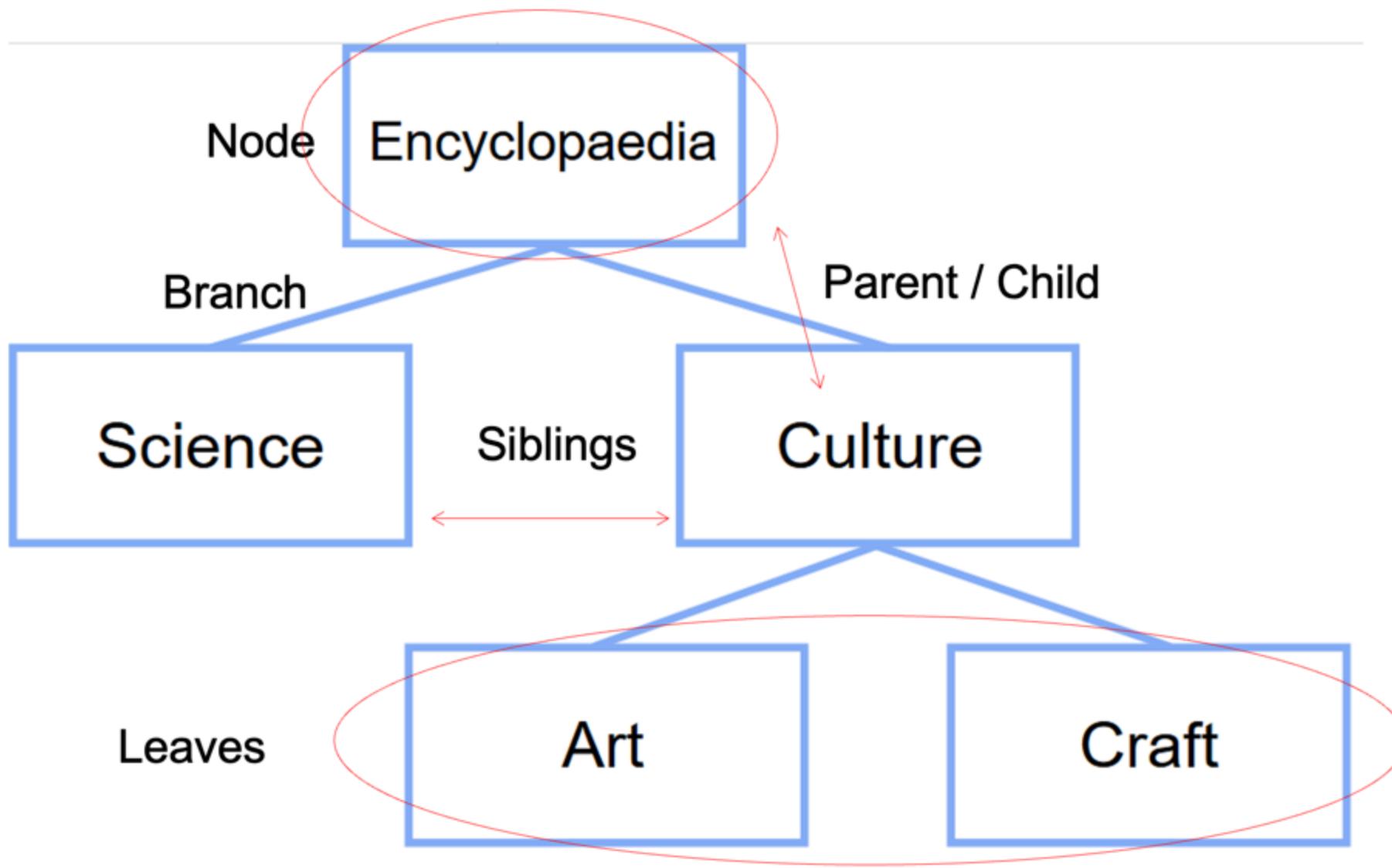
- LIFO: Last In, First Out
- push: Neues Element speichern
- pop: Letztes Element abrufen und entfernen
- Stapelspeicher, Kellerspeicher

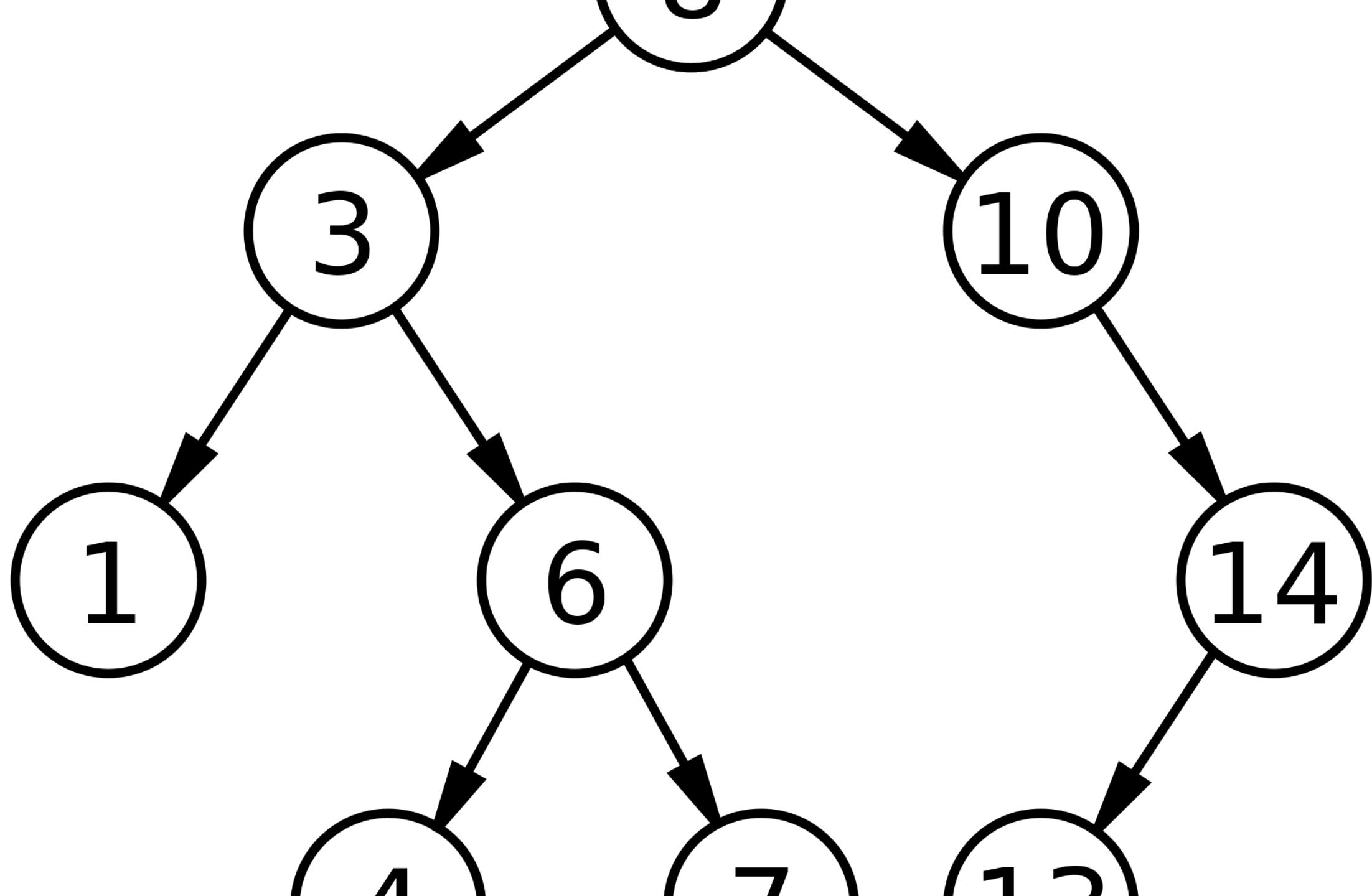


# Stack

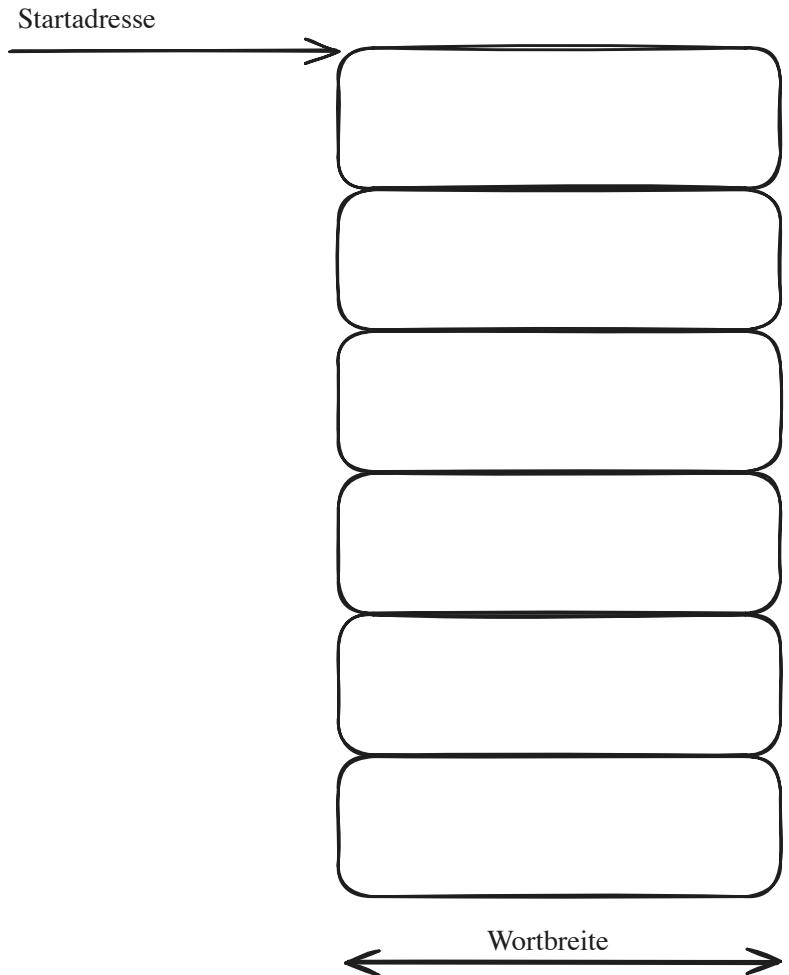


# Tree

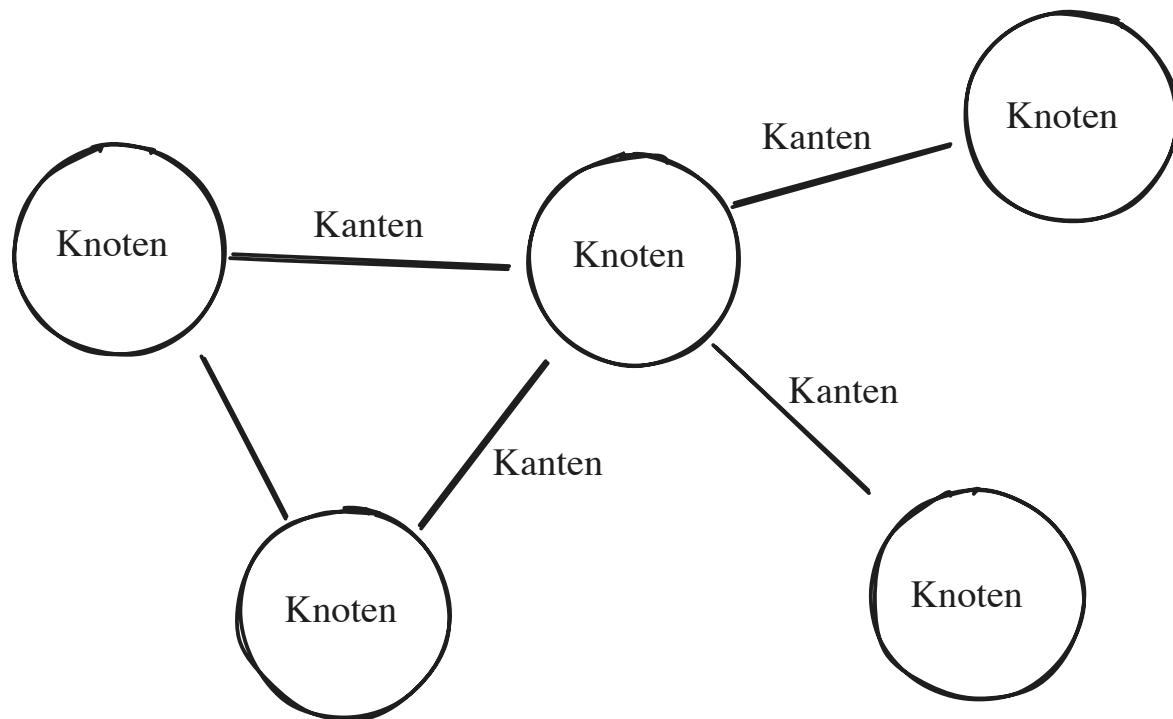




# Konkrete Datenstrukturen: Array



# Konkrete Datenstrukturen: Graph

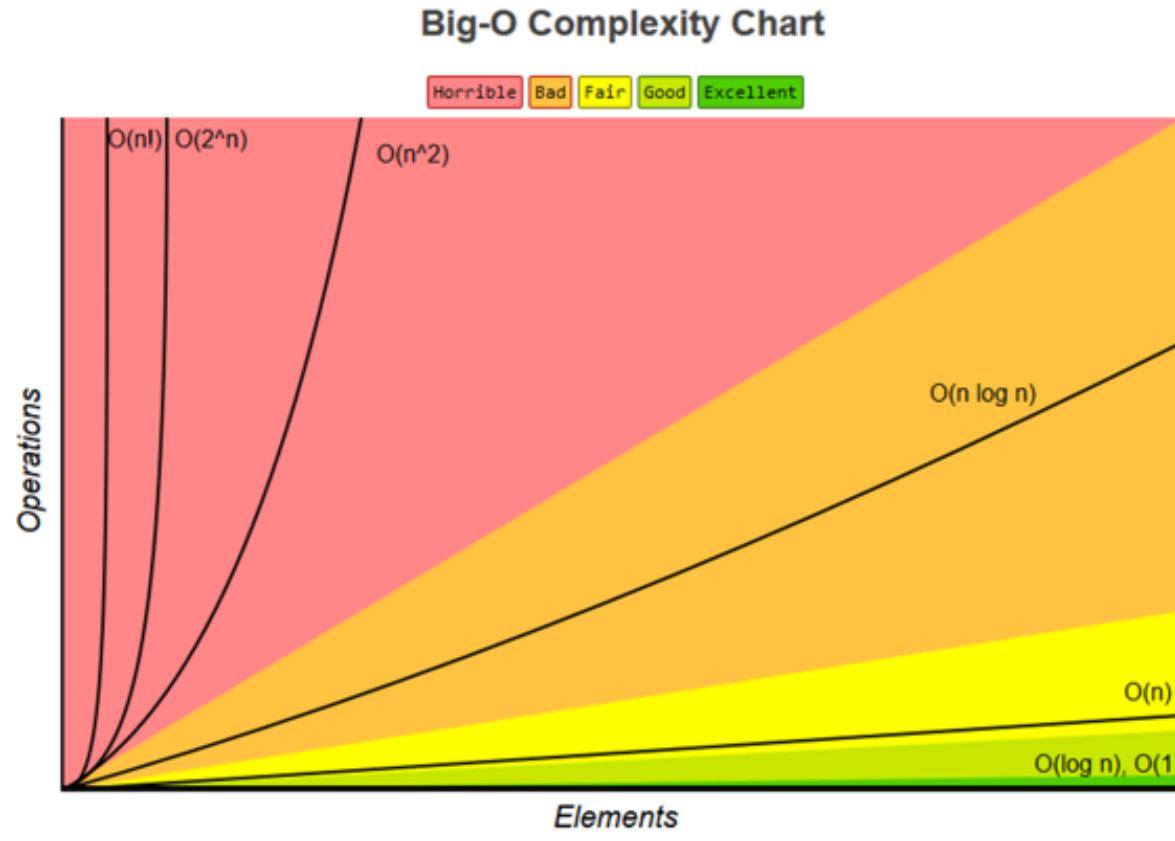


# Komplexität von Algorithmen

# Speicher und Rechenaufwand von Datenstrukturen

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

# Big O Notation



## Big O Cheatsheet

- $O(1)$ : 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)

## Alternative Big O Notation

O(1)	O(yeah)
O(logn)	O(nice)
O(n)	O(k)
O(n^2)	O(my)
O(2^n)	O(no)
O(n!)	O(mg)
O(n^n)	O(sh*t!)

<https://quanticdev.com/algorithms/primitives/alternative-big-o-notation/>

# 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

# 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");
}
```

# Fehlerbehandlung

- 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
}
```