

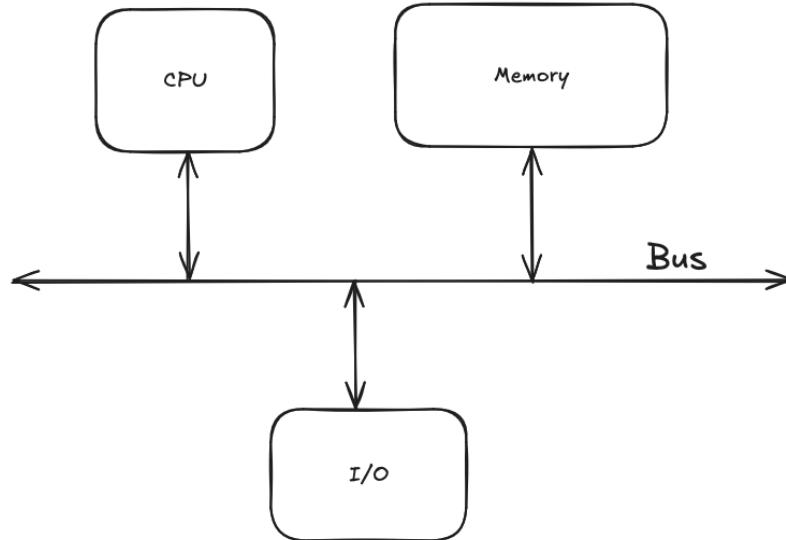
Einstieg

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

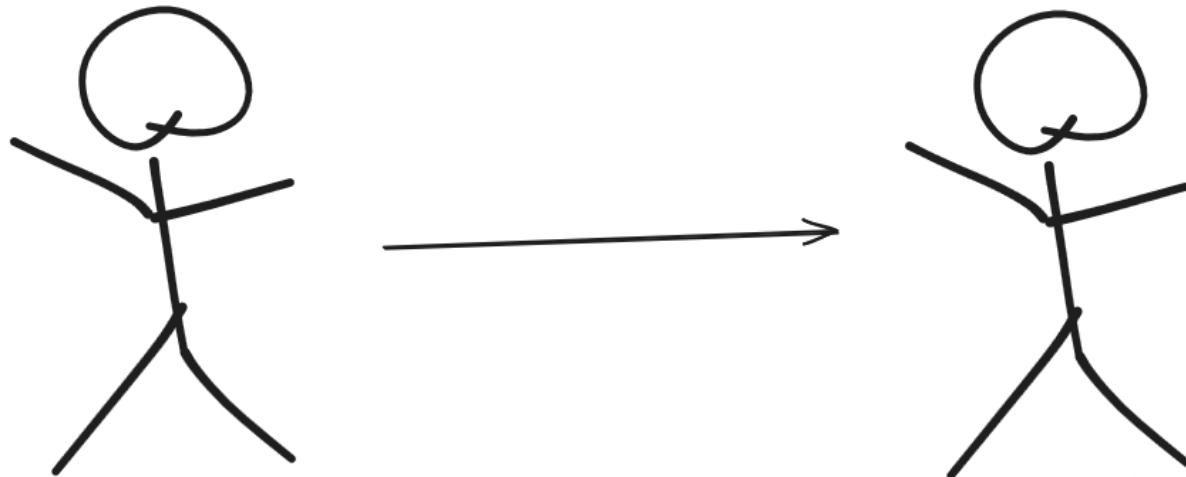
- Computer sind universelle Maschinen. Sie führen die Programme aus, die wir ihnen füttern.
- Der Computer findet das Programm im Speicher und führt es aus.
- Die einzigen Grenzen sind unsere Vorstellungskraft



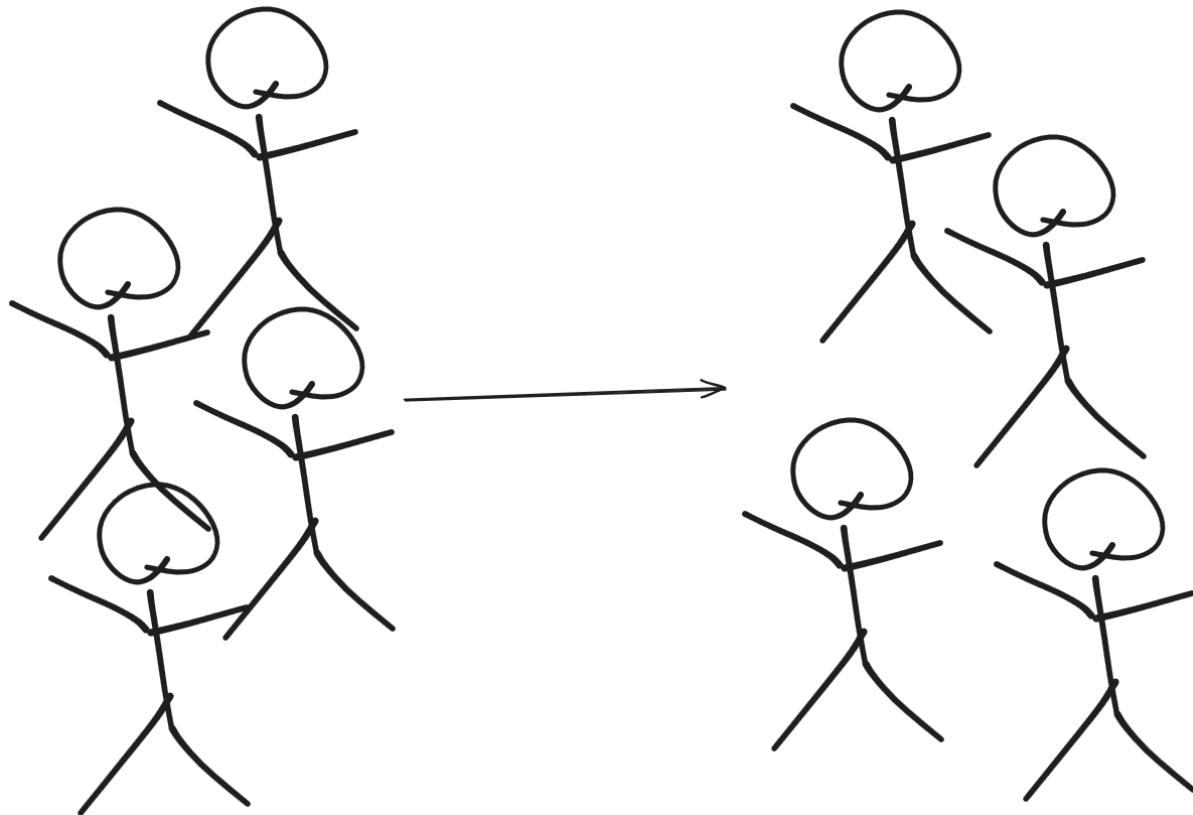
Computer

- 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.

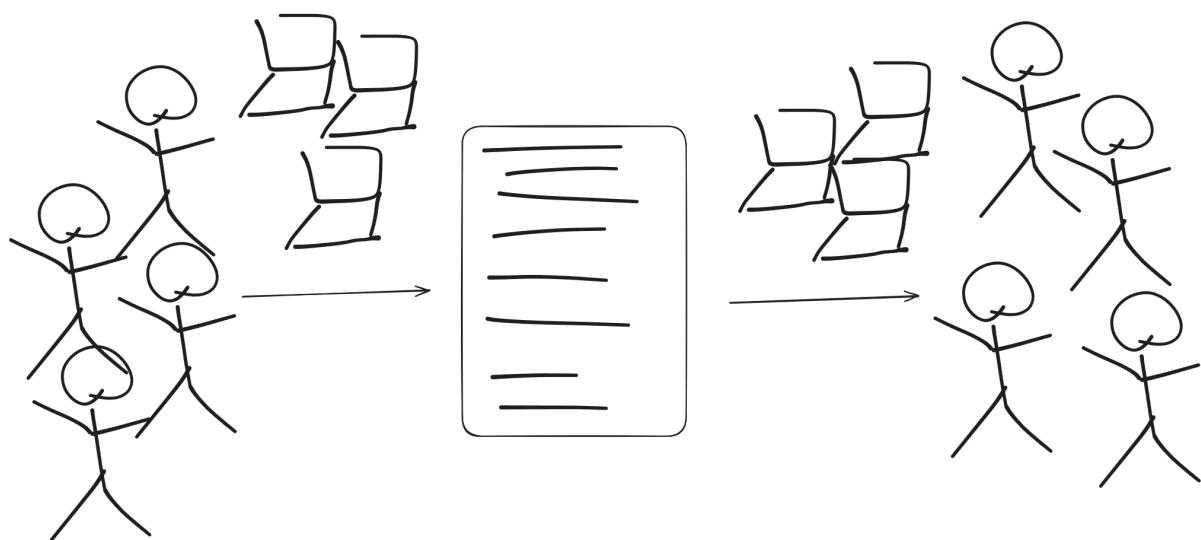
Programme erstellen und laufen lassen



Programme erstellen und laufen lassen



Programme erstellen und laufen lassen



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

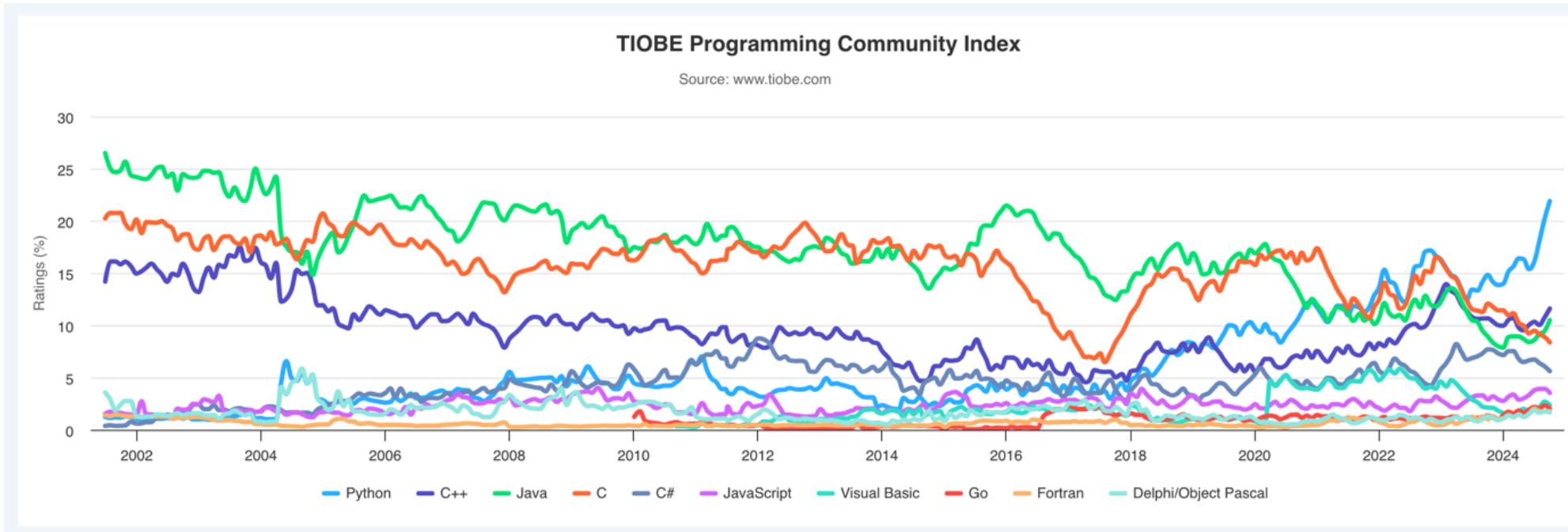
- Programme können «abstürzen»
- Programme, die nicht «abstürzen» funktionieren nicht zwangsläufig richtig.
 - [Ariane5 Rakete, 1996](#)
- Fehlerhafte Programme können Menschen töten (medizinische Geräte, Luftfahrt).
 - [Boeing 737 MCAS](#)
- Programmierer sind verantwortlich für das korrekte Funktionieren der Programme.
- Das Ziel dieses Fachs ist, nicht nur programmieren zu lernen, sondern gut programmieren zu lernen.

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

Übersicht

Name	Veröffentlichung		Typisierung	Paradigmen (imperativ sind alle)
C	1972	Kompiliert	statisch	strukturiert
C++	1985	Kompiliert	statisch	objektorientiert
Python	1991	Interpretiert	dynamisch	strukturiert, objektorientiert
JavaScript	1995	Interpretiert	dynamisch	objektorientiert
PHP	1995	Interpretiert	dynamisch	strukturiert, objektorientiert
Java	1995	Kompiliert (VM)	statisch	objektorientiert

Name	Veröffentlichung		Typisierung	Paradigmen
Go	2012	Kompiliert	statisch	strukturiert, objektorientiert
Rust	2015	Kompiliert	statisch	strukturiert, funktional, objektorientiert

Anwendungsgebiete

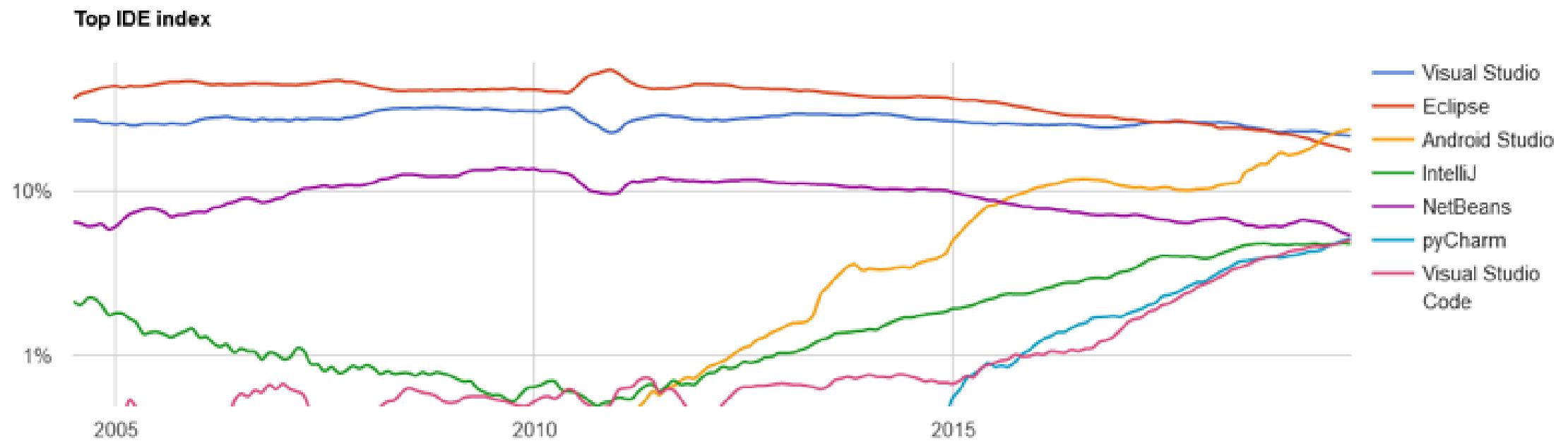
	Anwendungsgebiet
C	Betriebssystemen und Embedded
C++	Betriebssystemen, Desktop Applikationen, Games, Datenbanken, Interpreter
Java	Enterprise Umfeld
C#	Game Entwicklung (Unity), Microsoft Ökosystem, Enterprise Umfeld
Python	Wissenschaften, Machine Learning, Big Data, Einsteiger, Automation
JavaScript	Web Frontend und Backend
PHP	Web

	Anwendungsbereich
Go	Web Backend, Tooling, DevOps
Rust	Betriebssystemen, Desktop Applikationen, Games, Datenbanken, Interpreter, Web Backend

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



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

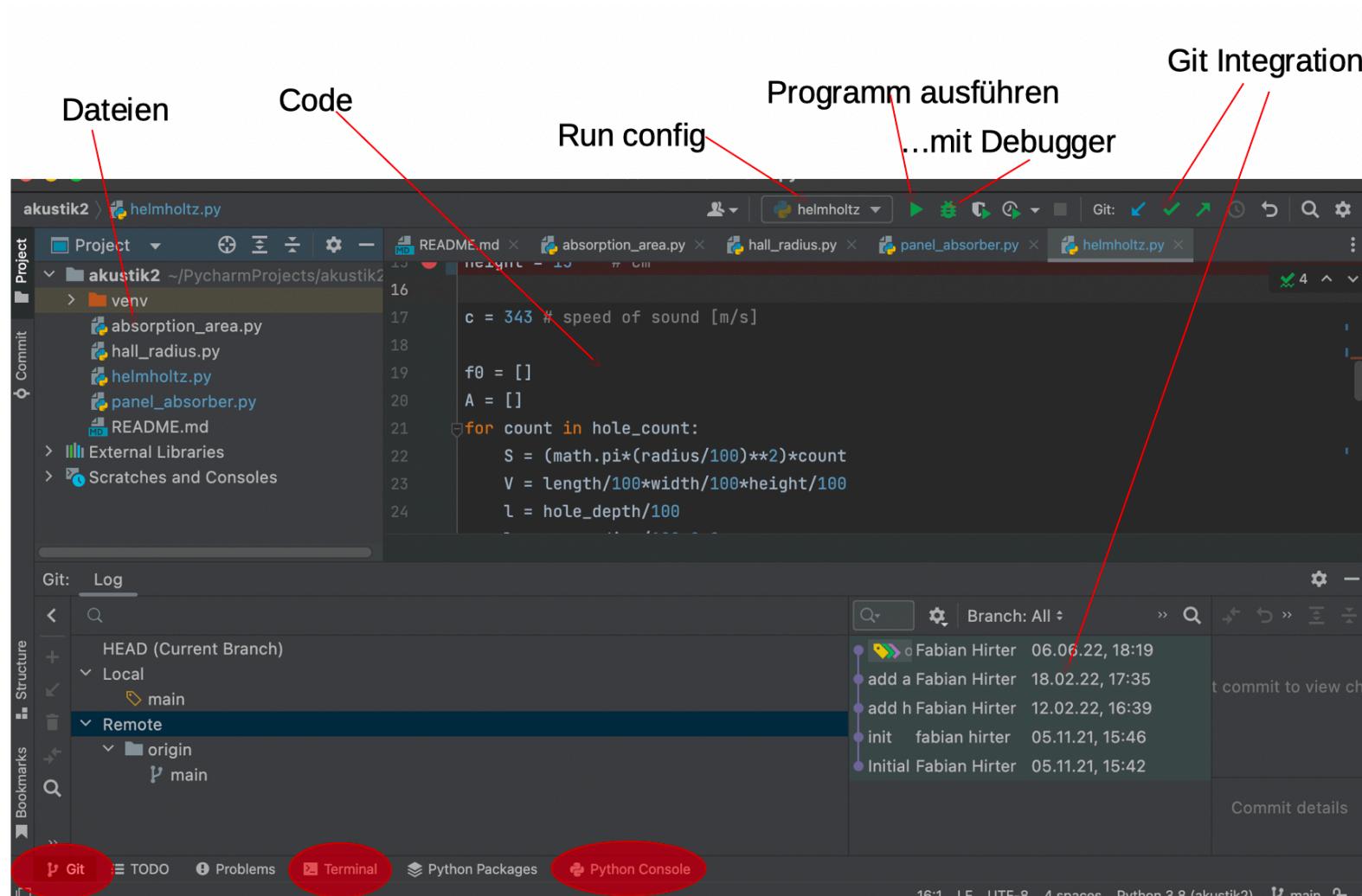
Microsoft Visual Studio Code

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

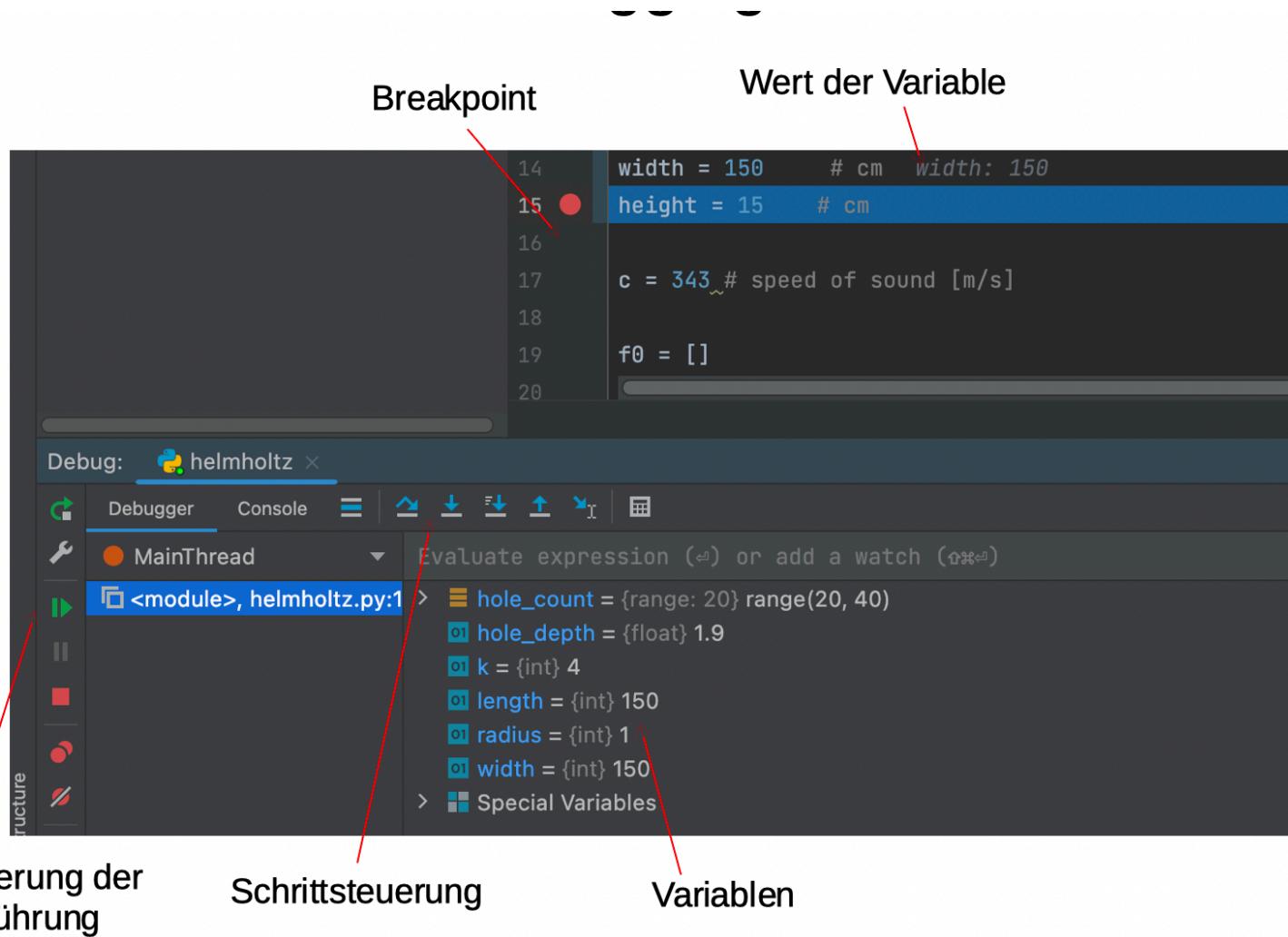
JetBrains

- 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

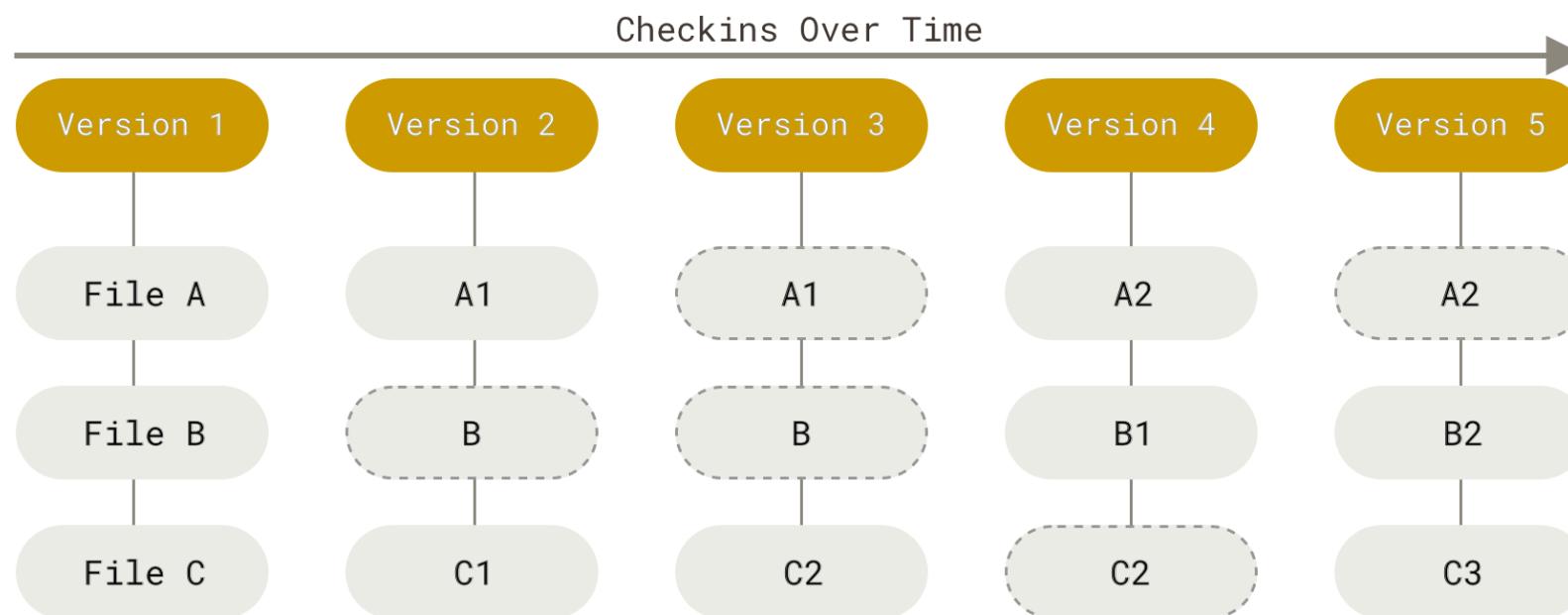
- Protokollierung von Änderungen
- Wiederherstellung von alten Ständen
- Archivierung
- Koordinierung des gemeinsamen Zugriffs
- Entwicklungszweige (Branches) -> **Don't Branch!**

Moderne Versionsverwaltung

- CI/CD
- GitOps
- Infrastructure as Code
- Documentation as Code
- Everything as Code

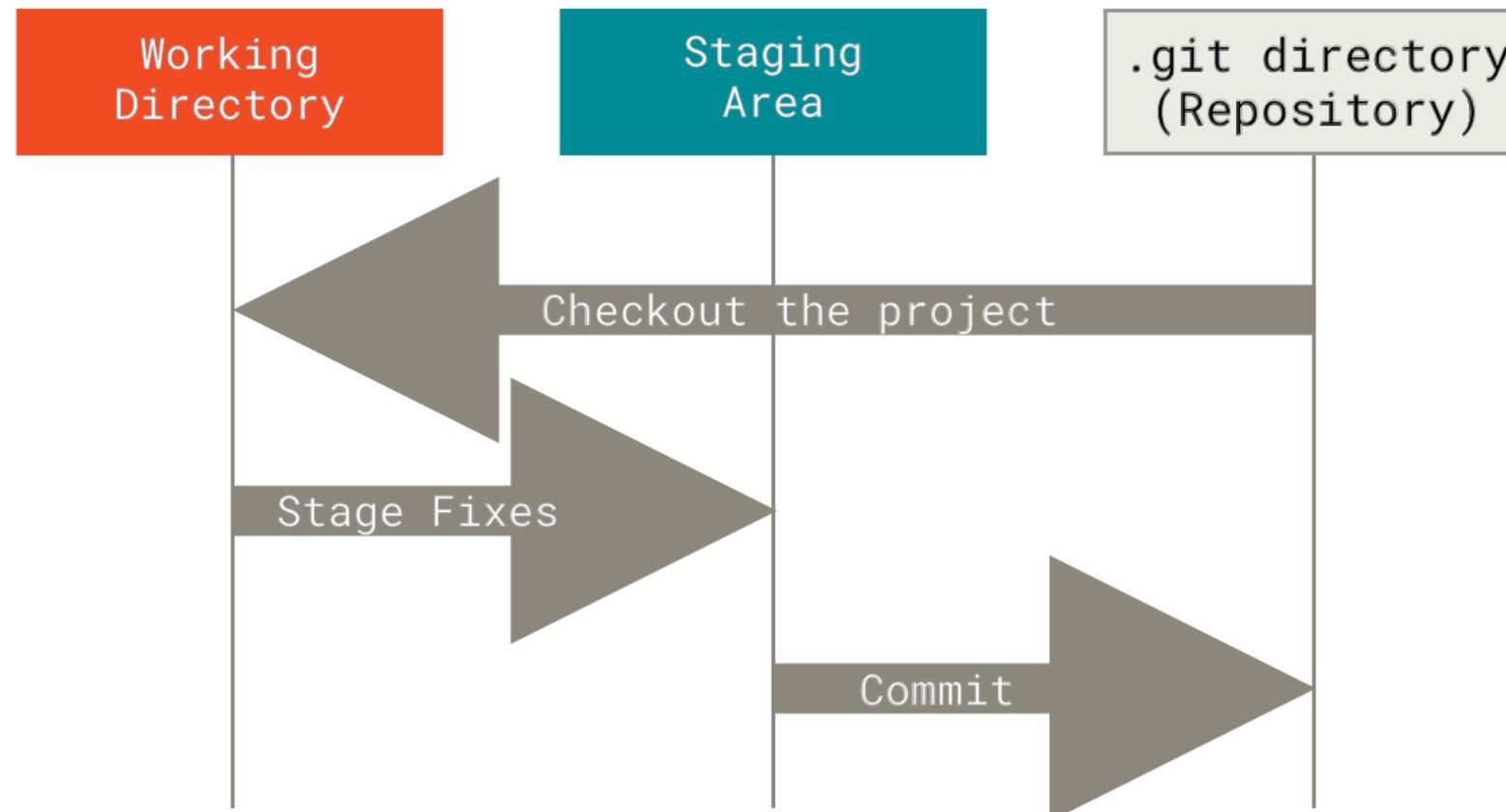
Git

- 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

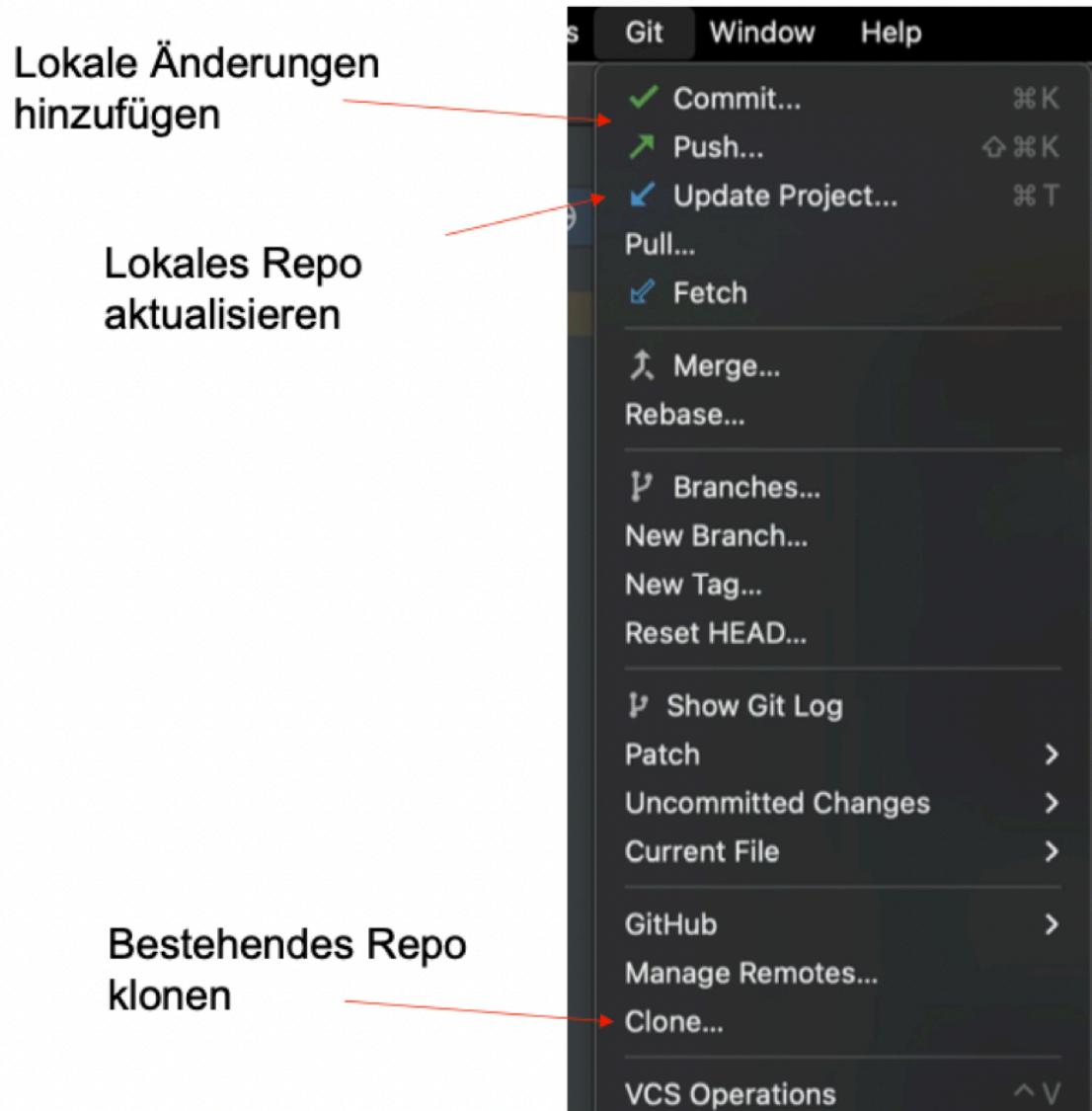
Initialisieren

- 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")

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

Conventional Commits

- Die Commit History (`git log`) sollte sich idealerweise wie ein Buch lesen
- [Conventional Commits](#) sind spezifizieren die Struktur von Commits
- Dies macht die Art der Änderung klarer
- Damit können Versionen und Changelogs automatisch erstellt werden
- Mit Tags werden Releases markiert. [semantic versioning](#).

```
feat: allow provided config object to extend other configs
```

```
BREAKING CHANGE: `extends` key in config file is now used for extending other config files
```

Git Ressourcen

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

Plattformen: Github / Gitlab

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

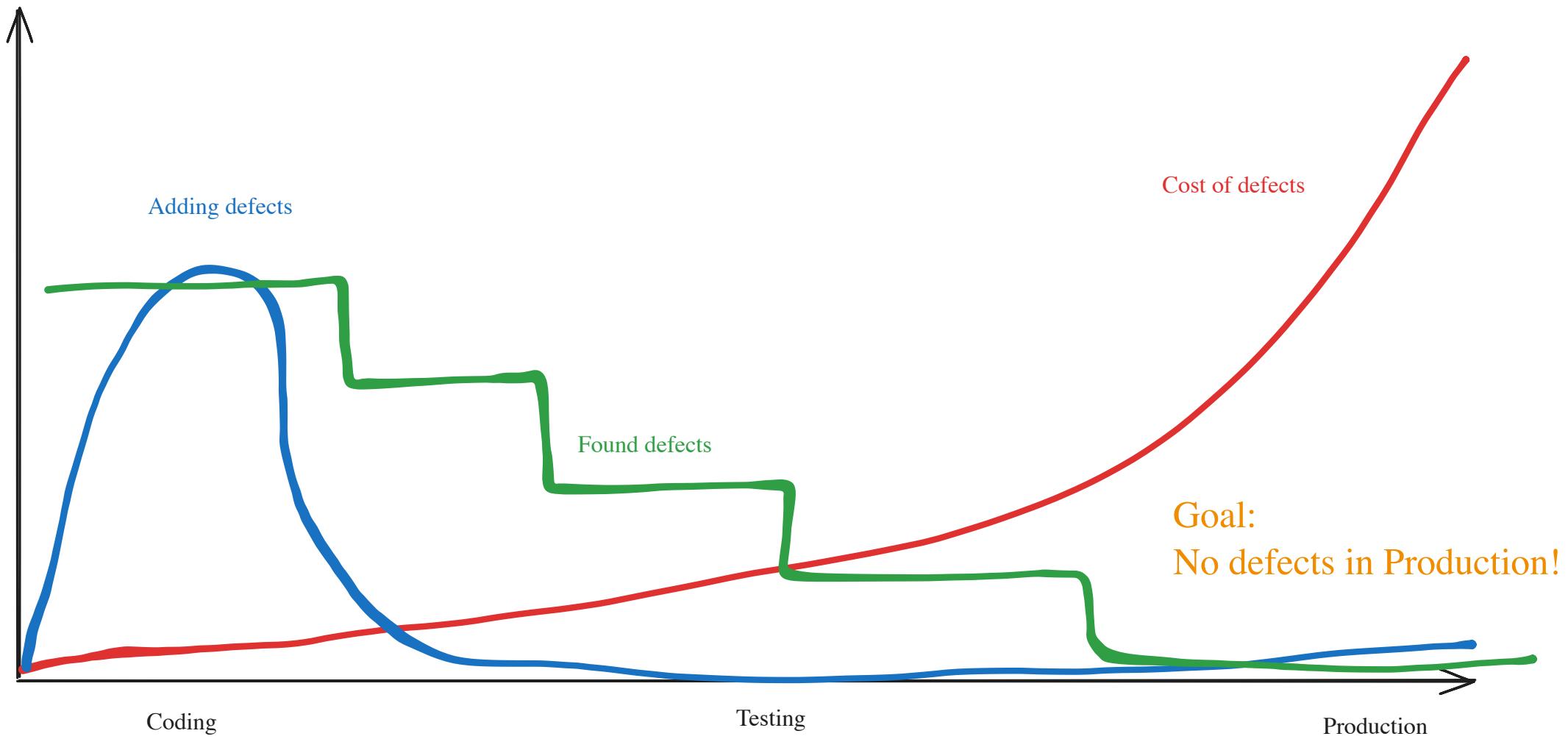
Issue Tracking

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.

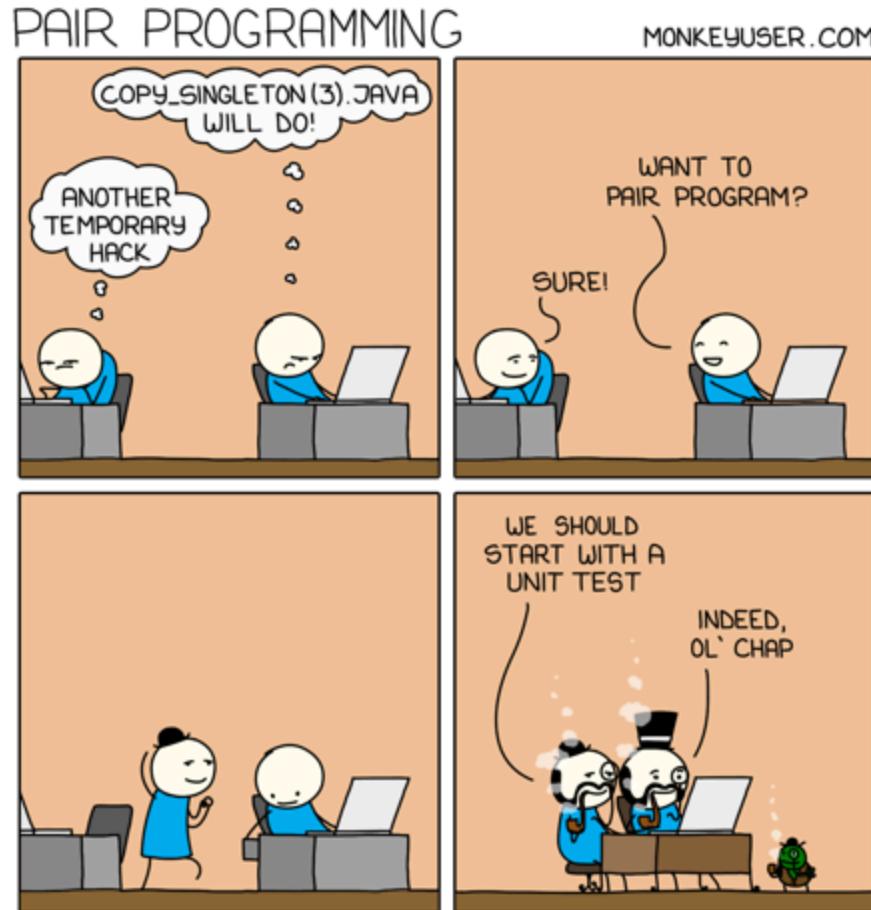
- Open:** 27 items.
 - Parcel visualization (#95)
 - Tags as out of ordinary visualization (#94)
 - Payment integration (#45)
 - Set Security Headers (Security) (#91)
 - lane editing (Feature, low hanging fruit) (#6)
- Next:** 6 items.
 - Authentication (Security) (#39)
 - Create REST API (Feature) (#51)
 - 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)

Testing

Kosten von Defekten



Pair Programming



Test Driven Development (TDD)

- Test First: Fokus auf die Problemstellung und Schnittstelle
- Nur eigenen Code testen. Datenbanken, APIs oder Libraries werden nur im Rahmen von Integrationstests aufgerufen.
- Tests geben eine Rückmeldung zum Code: Wenn Code schwierig zu testen ist, sollte er vermutlich anders strukturiert werden.
- **Humble Object**: Code, der schwierig zu testen ist in einem minimalen Objekt isolieren

Write a
failing
test

Make the
test pass

Refactor



Write a
failing
test

Make the
test pass

Refactor

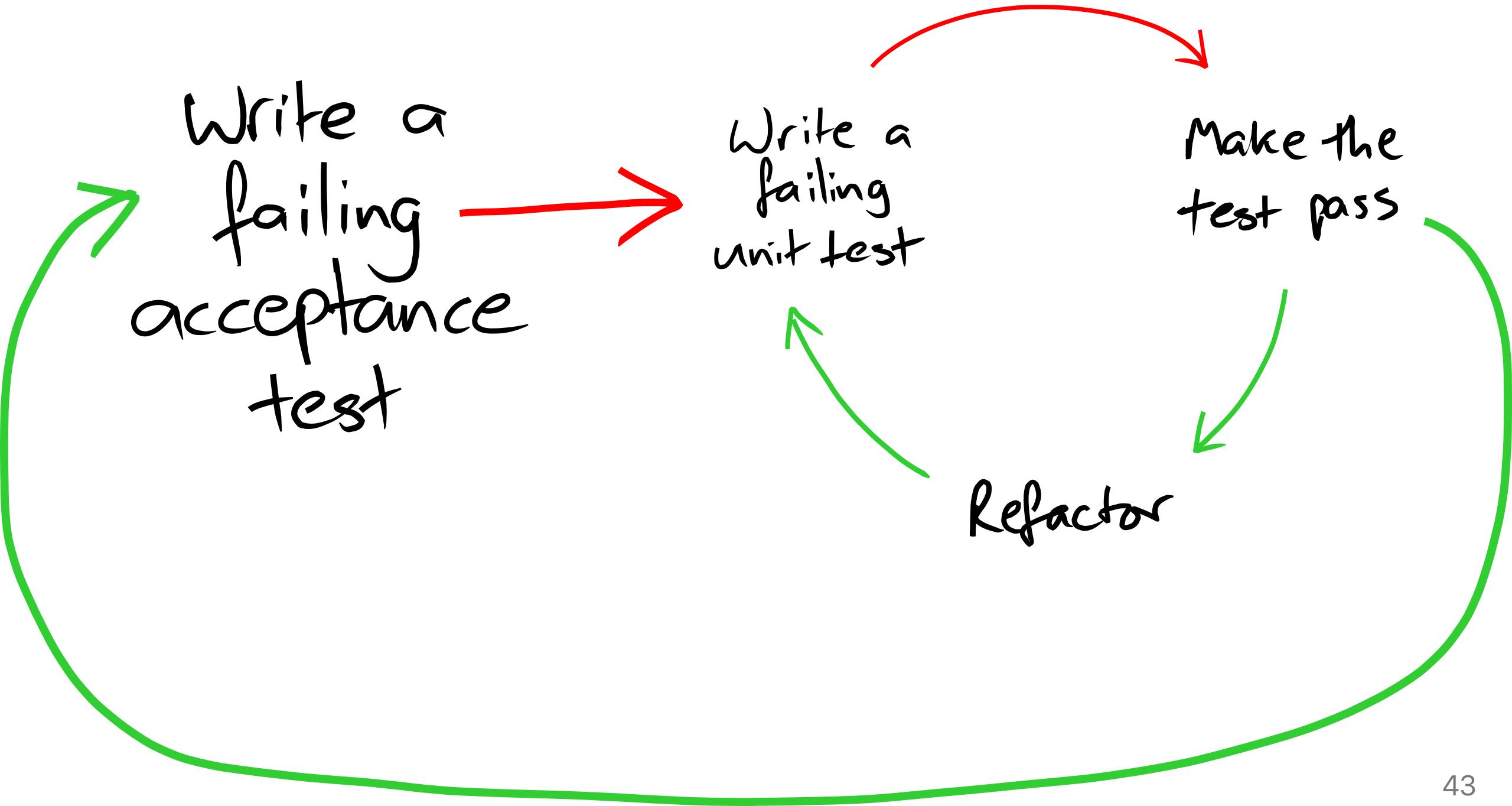
Hard to write a test?

Write a failing acceptance test

Write a failing unit test

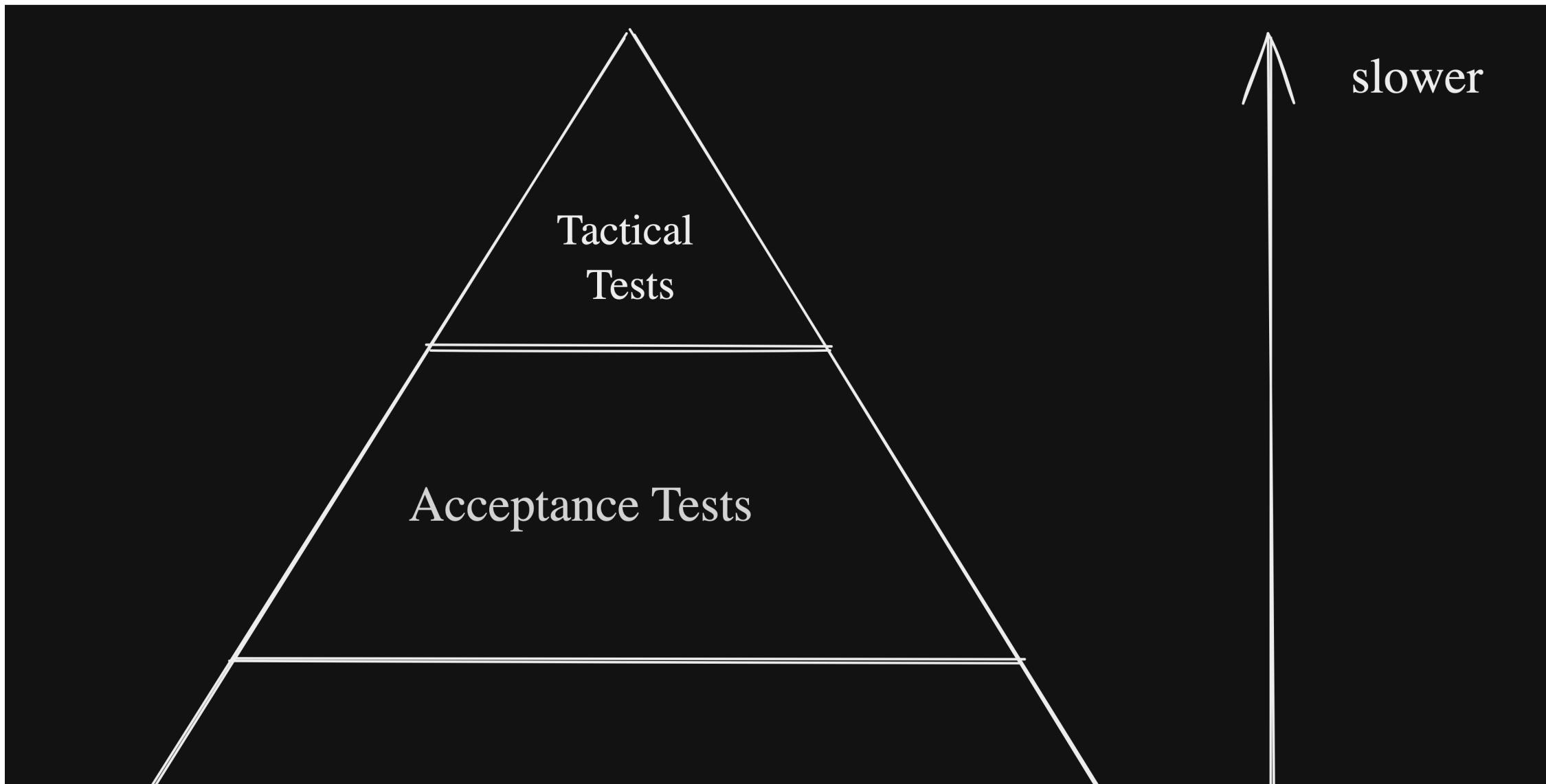
Make the test pass

Refactor



Drei Abbildungen aus: Growing Object-Oriented Software by Nat Pryce and Steve Freeman

Testpyramide



Testing: AAA

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

```
# Unit Under Test
from ProgrammingBasicsAndAlgorithms.Exercises.Basics.Solutions.sort import sort
# Testing Library
import unittest

# Test suite
class TestSort(unittest.TestCase):
    # Test
    def test_sort(self):
        a = [3, 0, 12, 8]  # Arrange
        b = sort(a)  # Act
        self.assertEqual(b, [0, 3, 8, 12])  # Assert
```

FIRST

- Fast: Tests should run quickly to encourage frequent execution.
- Isolated: Each test should run independently and not depend on other tests or external systems.
- Repeatable: Tests should produce the same results every time, regardless of the environment.
- Self-validating: Tests should have clear pass/fail outcomes without manual inspection.
- Timely: Write tests when the code is fresh, ideally before or during implementation.

IDE Integration

```
4 ▶ class TestSort(unittest.TestCase):  ↗ Fabian Hirter
5 ▶     def test_sort(self):  ↗ Fabian Hirter
6         a = [3, 0, 12, 8]
7         b = sort(a)
8         self.assertEqual(b, second: [0, 3, 8, 12])
```

Testing: Further Reading

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

Dokumentation (as Code)

- Geeignet für Versionverwaltung
- Kein Kontextwechsel für Dokumentation

Markdown

Markdown

- Triviale Syntax
- Sehr zukunftssicher
- Syntax-highlight für Code
- Automatisierbar
- Sehr hohe Verbreitung

Markdown Tools

- [Documents](#)
- [Github](#)
- [Gitlab](#)
- [Websites](#)
- [Notes](#)
- [Books](#)
- [Documentation](#)
- [etc etc etc](#)

Markdown Basic Syntax

Basic Syntax

These are the elements outlined in John Gruber's original design document. All Markdown applications support these elements.

Element	Markdown Syntax
Heading	# H1 ## H2 ### H3
Bold	bold text
Italic	<i>italicized text</i>
Blockquote	> blockquote
Ordered List	1. First item 2. Second item 3. Third item
Unordered List	- First item - Second item - Third item

Markdown Renderer

- Dokumentationswebsites mit MKDocs
- Slides mit Marp
- Websites mit Hugo

Diagramme in Markdown

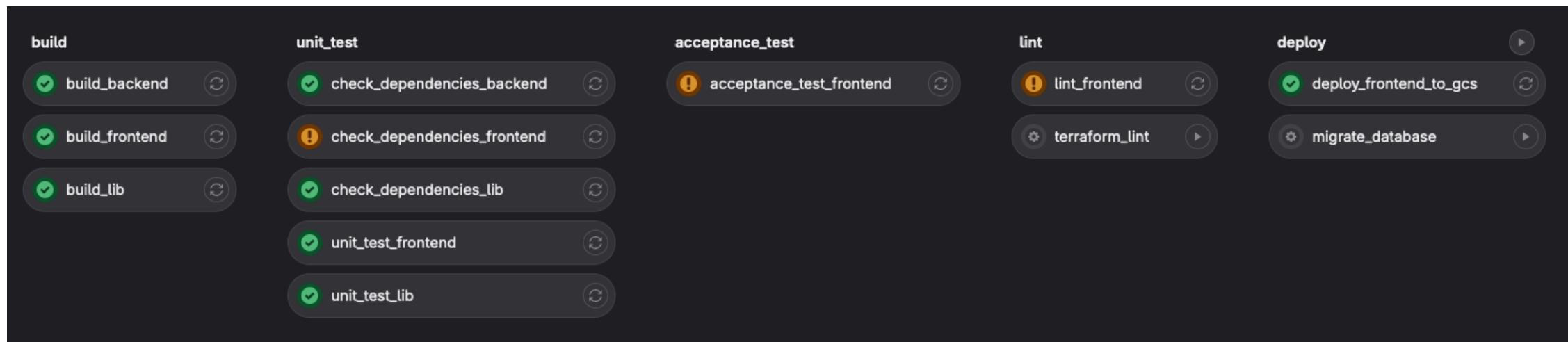
-Mermaid

CI / CD

CI/CD mit Git

- Tests und Linter werden bei Commit automatisch ausgeführt und Commit ggf. abgelehnt.
- Das neuste Release wird automatisch deployed.

Gitlab CI/CD Plattform



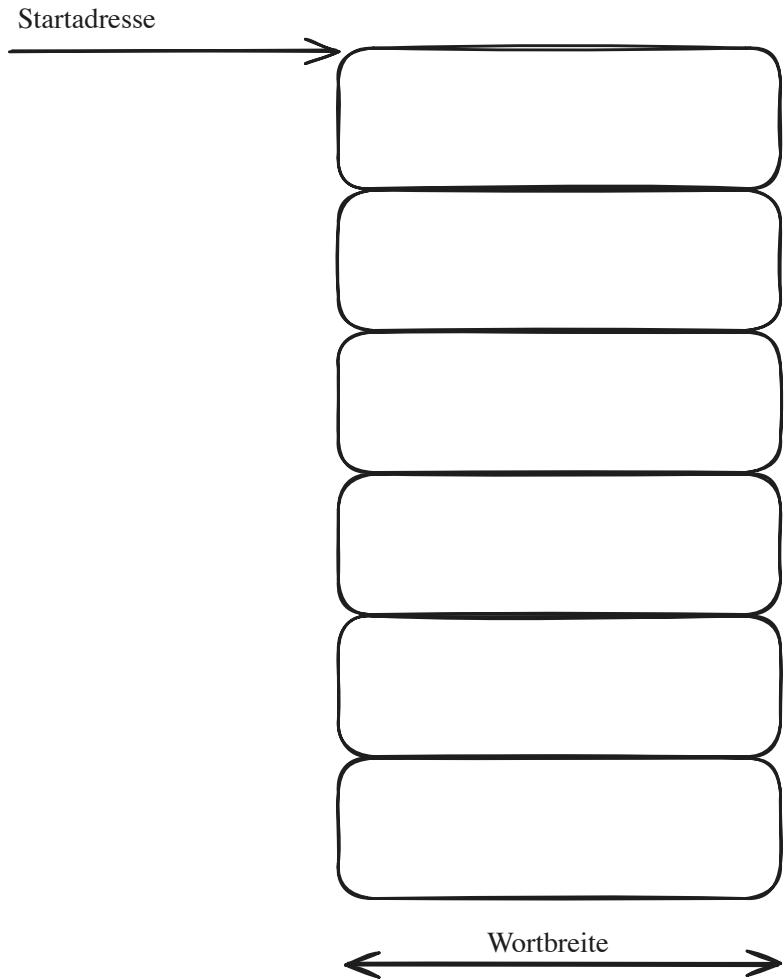
Algorithmen und Datenstrukturen

Containerdatenstrukturen

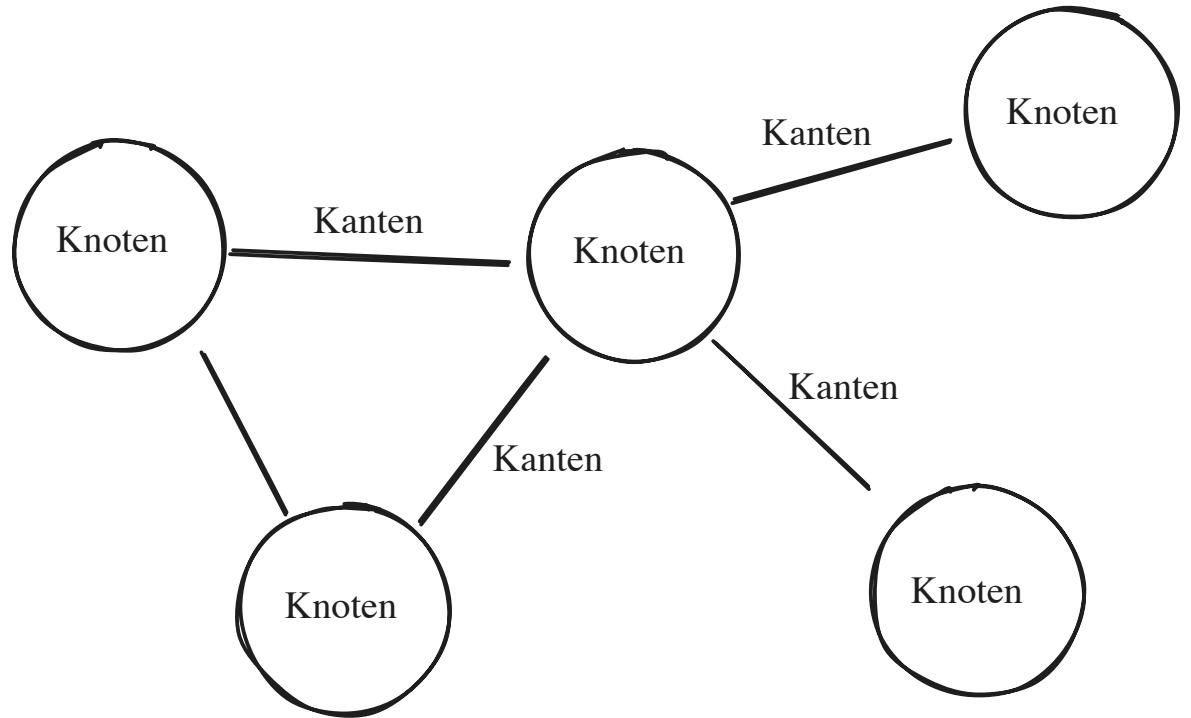
- Enthalten andere Objekte («items»)
- Grundsätzliche Operationen:
 - Elemente hinzufügen
 - Elemente entfernen
 - Ein Element suchen
 - Über alle Elemente iterieren
- Verschiedene Implementationen unterscheiden sich
 - Welche Operationen möglich sind
 - Wie schnell diese sind
 - Wie der Speicher ausgenutzt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

Konkrete Datenstrukturen

Array



Graph



Abstrakte Datenstrukturen

Record

- Einfachste Anordnung von Daten
- Zeile in Datenbank / Tabelle
- Datenobjekte

```
// 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

```
# 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]
first_item = list2[0] # select first item
list2[1:5] # select items 2 to 6
```

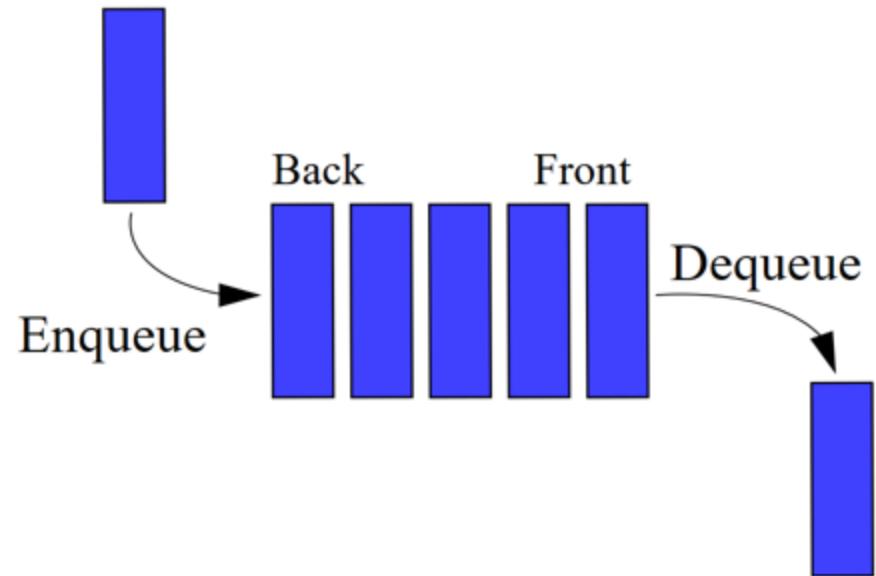
Map

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

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

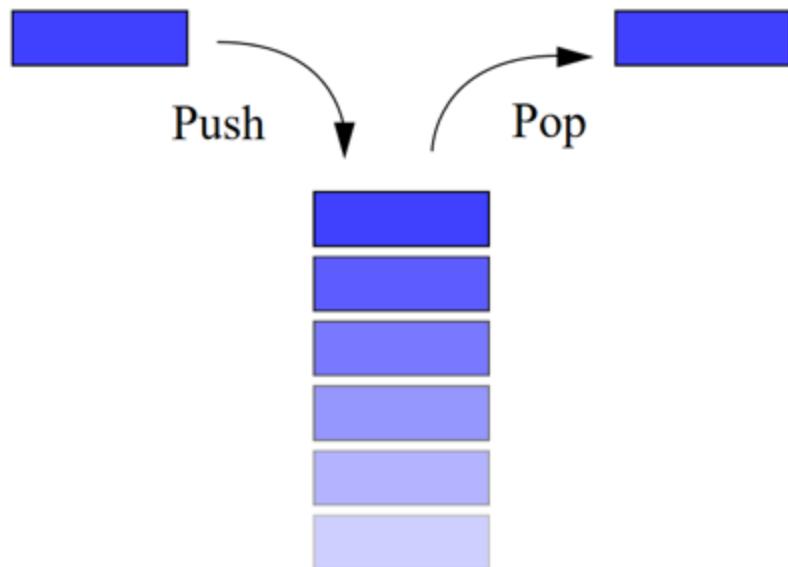
Queue

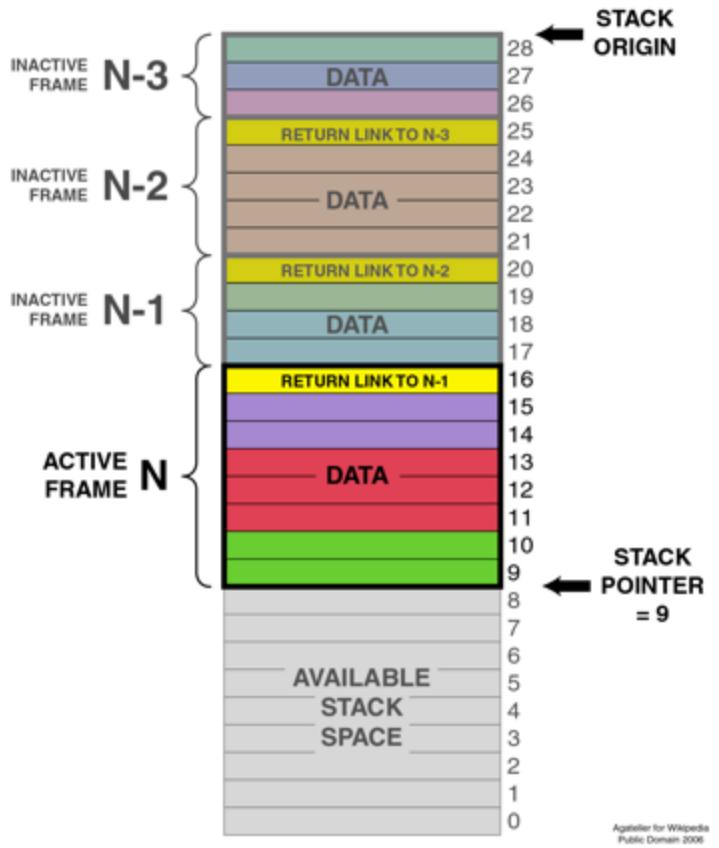
- FIFO: First In, First Out
- Warteschlange, Pipe



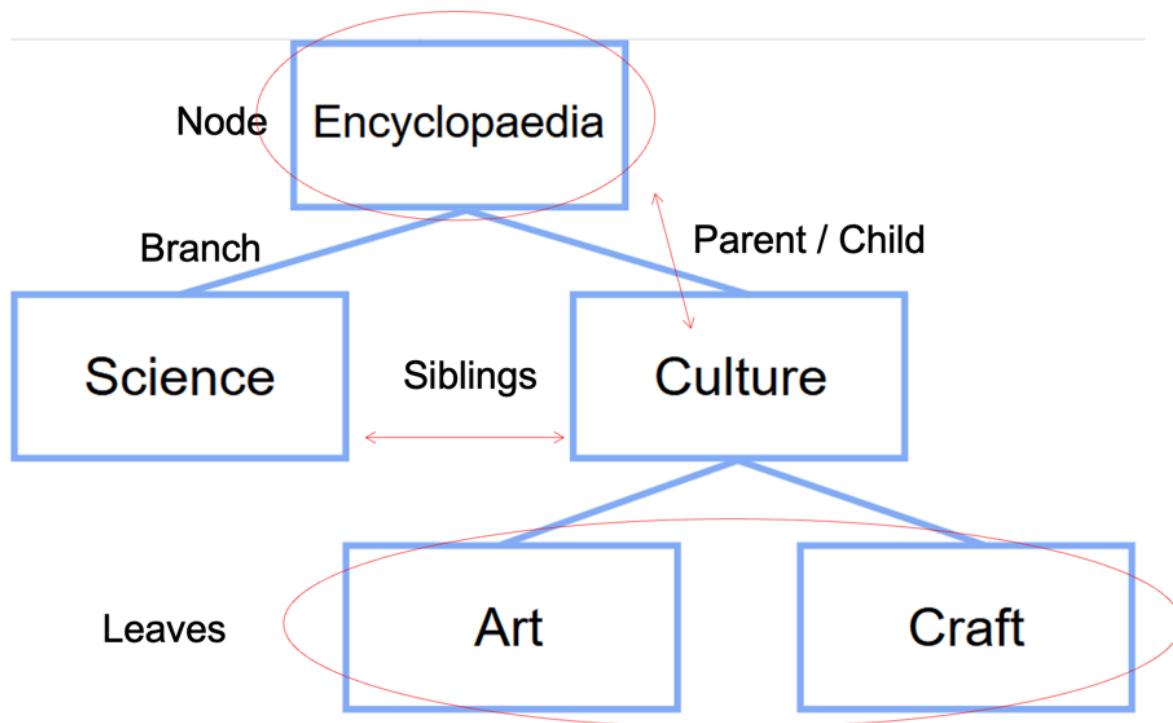
Stack

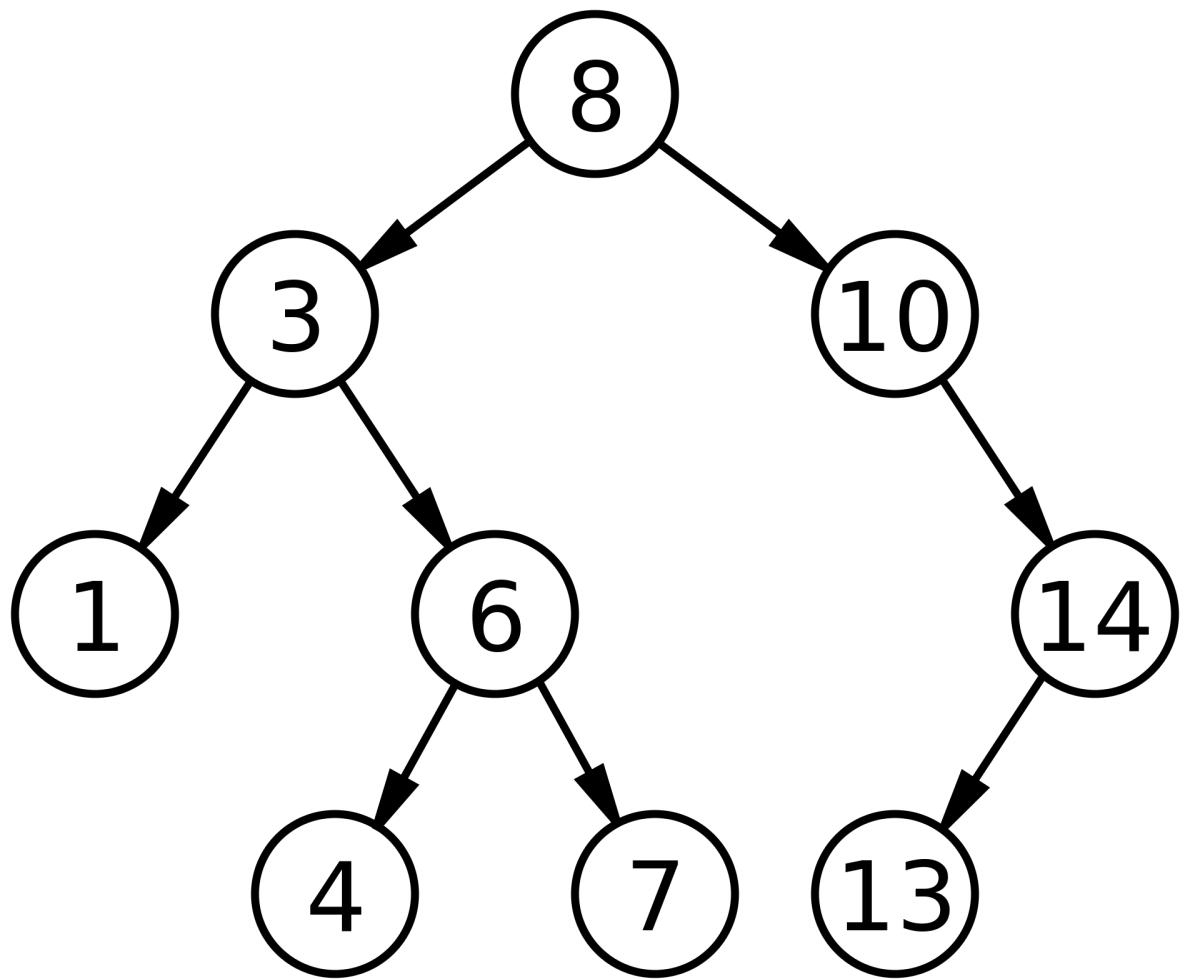
- LIFO: Last In, First Out
- push: Neues Element speichern
- pop: Letztes Element abrufen und entfernen
- Stapspeicher, Kellerspeicher





Tree



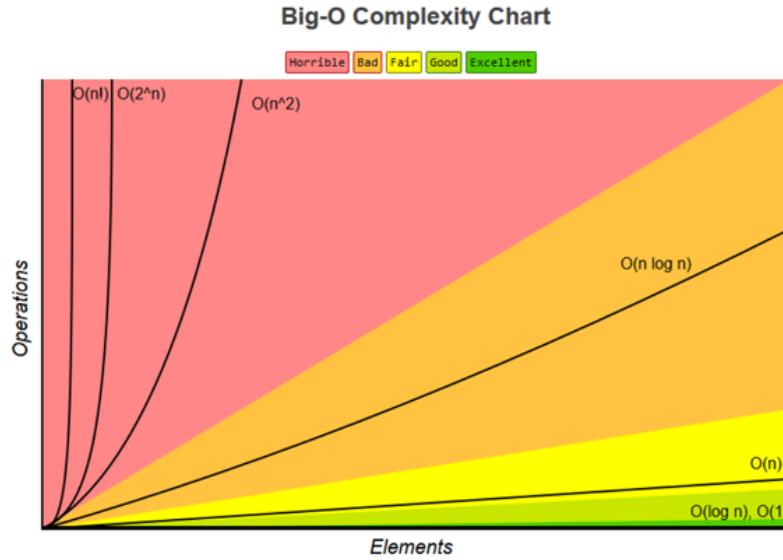


Komplexität von Algorithmen

Speicher und Rechenaufwand von Datenstrukturen

Operation	Array	Linked List	Binary Tree	Hashtable
Einfügen	$O(n)$	$O(1)$		
Löschen	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$
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/>

Dependencies

Herausforderungen

- Verschiedene Projekte benötigen verschiedene Versionen der gleichen Bibliothek
- Sicherheitslücken müssen erkannt und behoben werden
- Neue Funktionalität sollte geladen werden können
- Builds sollten reproduzierbar sein

Lösung: Packetmanager

- Python: PIP
- NodeJS: NPM
- PHP: Composer
- Go: go get
- Rust: cargo
- Java: Maven

Paketmanager

- Laden von Bibliotheken `pip install pip-audit`
- Aktualisierungen `pip install --upgrade <package-name>`
- Dependency Checks: `pip-audit` , `npm audit`
- Dependencies werden pro Projekt festgehalten
 - Python: `requirements.txt`, `virtual env`
 - JS: `package.json`, `node_modules`
 - GO: `go.mod`

Requirements.txt

```
pip freeze > requirements.txt # generate  
pip install -r requirements.txt # load
```

```
click == 8.1.3  
ghp-import == 2.1.0  
importlib-metadata == 4.11.4  
Jinja2 == 3.1.2  
joblib == 1.2.0  
Markdown == 3.3.7
```

Semantic Versioning

Auf Grundlage einer Versionsnummer von MAJOR.MINOR.PATCH werden die einzelnen Elemente folgendermaßen erhöht:

1. MAJOR wird erhöht, wenn API-inkompatible Änderungen veröffentlicht werden,
2. MINOR wird erhöht, wenn neue Funktionalitäten, die kompatibel zur bisherigen API sind, veröffentlicht werden, und
3. PATCH wird erhöht, wenn die Änderungen ausschließlich API-kompatible Bugfixes umfassen.

Außerdem sind Bezeichner für Vorveröffentlichungen und Build-Metadaten als Erweiterungen zum MAJOR.MINOR.PATCH-Format verfügbar.

Tipps

- Abhängigkeiten Automatisiert auf Updates und Sicherheitslücken testen
- Versionen immer "pinnen": 3.2.1, ^3.2.1, 3
- Vorsicht bei indirekten Abhängigkeiten!

Container

Applikation wird mitsamt ihren Abhängigkeiten in einen "Container" gepackt.

```
FROM node:20

ARG ENVIRONMENT="development"

WORKDIR /usr/src/app

COPY backend /usr/src/app
RUN rm /usr/src/app/lib
COPY lib /usr/src/app/lib

RUN npm ci --verbose

EXPOSE 8080

CMD ./setup-ci.sh
```

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

Refactoring

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

Fehlerbehandlung

- 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.

Geworfene Exceptions

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

```
raise Exception('<error message>')
```

Fehler als Rückgabewert

- Exceptions können es schwierig machen, den Programmablauf nachzuvollziehen, weil Exceptions den normalen Programmablauf unterbrechen.
- In Go müssen Fehler als Rückgabewert explizit angegeben werden.
- Das kann mit Fehlertypen auch in den meisten anderen Sprachen erreicht werden

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

```
func Sqrt(f float64) (float64, error) {
    if f < 0 {
        return 0, errors.New("math: square root of negative number")
    }
}
```

Fehler als Rückgabewert in Python

```
def return_value():
    return 'foo', None

def test_return_value(self):
    value, error = return_value()
    self.assertIsNotNone(value)
    self.assertIsNone(error)

def return_error():
    return None, Exception('something went wrong')

def test_return_error(self):
    value, error = return_error()
    self.assertIsNotNone(error)
    self.assertIsNone(value)
```

Programmierelemente

Alle folgenden Folien aus: <https://go.dev/ref/spec>

Lexical Elements

Kommentare

```
// single line comment

/*
multi
line
comment
*/
```

Keywords

break
case
chan
const
continue

default
defer
else
fallthrough
for

func
go
goto
if
import

interface
map
package
range
return

select
struct
switch
type
var

Operators and Punctuation

+	&	+=	&=	&&	==	!=	()
-	-	-=	=		<	<=	[]
*	^	*=	^=	<-	>	>=	{	}
/	<<	/=	<<=	++	=	:=	,	;
%	>>	%=	>>=	--	!	:
& [^]		& [^] =		~				

Identifiers

- Identifiers name program entities such as variables and types.
- An identifier is a sequence of one or more letters and digits.
- The first character in an identifier must be a letter.

Predefined Identifiers

Types:

any bool byte comparable
complex64 complex128 error float32 float64
int int8 int16 int32 int64 rune string
uint uint8 uint16 uint32 uint64 uintptr

Constants:

true false iota

Zero value:

nil

Functions:

append cap clear close complex copy delete imag len
make max min new panic print println real recover

Literals

```
// Integer  
42  
0x_67_7a_2f_cc_40_c6  
170_141183_460469_231731_687303_715884_105727  
0b0101010101  
  
// Floating Point  
0.  
72.40  
2.71828  
6.67428e-11  
1E6  
  
// String  
"Hello, world!"
```

Variables

- A variable is a storage location for holding a value.
- The set of permissible values is determined by the variable's type.

Types

- A type determines a set of values together with operations and methods specific to those values.
- A type may be denoted by a type name [...].

Numeric Types

<code>uint8</code>	the set of all unsigned 8-bit integers (0 to 255)
<code>uint16</code>	the set of all unsigned 16-bit integers (0 to 65535)
<code>uint32</code>	the set of all unsigned 32-bit integers (0 to 4294967295)
<code>uint64</code>	the set of all unsigned 64-bit integers (0 to 18446744073709551615)
<code>int8</code>	the set of all signed 8-bit integers (-128 to 127)
<code>int16</code>	the set of all signed 16-bit integers (-32768 to 32767)
<code>int32</code>	the set of all signed 32-bit integers (-2147483648 to 2147483647)
<code>int64</code>	the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)
<code>float32</code>	the set of all IEEE 754 32-bit floating-point numbers
<code>float64</code>	the set of all IEEE 754 64-bit floating-point numbers
<code>complex64</code>	the set of all complex numbers with <code>float32</code> real and imaginary parts
<code>complex128</code>	the set of all complex numbers with <code>float64</code> real and imaginary parts
<code>byte</code>	alias for <code>uint8</code>
<code>rune</code>	alias for <code>int32</code>
<code>uint</code>	either 32 or 64 bits
<code>int</code>	same size as <code>uint</code>

Array Types

```
[32]byte  
[3][5]int  
[2][2][2]float64 // same as [2]([2]([2]float64))
```

Struct Types

```
struct {
    x, y int
    u float32
    _ float32 // padding
    A *[]int
    F func()
}
```

Blocks

- A block is a possibly empty sequence of declarations and statements within matching brace brackets.
- Blocks nest and influence scoping.

Declarations

- A declaration binds a non-blank identifier to a constant, [...] variable, function, [...].
- Every identifier in a program must be declared.
- No identifier may be declared twice in the same block, [...].

```
// constant
const Pi float64 = 3.14159265358979323846

// variable
var x int
var i = 42

// function
func IndexRune(s string, r rune) int {
    // implementation
    return
}
```

Scope

The scope of a declared identifier is the extent of source text in which the identifier denotes the specified constant, type, variable, function, label, or package.

Go is lexically scoped using blocks:

1. The scope of a predeclared identifier is the universe block.
2. The scope of an identifier denoting a constant, type, variable, or function [...] declared at top level (outside any function) is the package block.
3. The scope of an identifier denoting a [...] function parameter, or result variable is the function body.
4. The scope of a constant or variable identifier declared inside a function begins at the end of the ConstSpec or VarSpec [...] and ends at the end of the innermost containing block.

Expressions

An expression specifies the computation of a value by applying operators and functions to operands.

```
2 + 3  
square(5)
```

Statements

Statements control execution.

Assignment

An assignment replaces the current value stored in a variable with a new value specified by an expression.

```
x = 1  
*p = f()  
a[i] = 23
```

If Statements

- "If" statements specify the conditional execution of two branches according to the value of a boolean expression.
- If the expression evaluates to true, the "if" branch is executed, otherwise, if present, the "else" branch is executed.

```
if x > max {  
    x = max  
} else {  
    x = 3  
}
```

Switch Statements

- "Switch" statements provide multi-way execution.
- An expression or type is compared to the "cases" inside the "switch" to determine which branch to execute.

```
switch {  
    case x < y:  
        f1()  
    case x < z:  
        f2()  
    case x == 4:  
        f3()  
}
```

For Statements

A "for" statement specifies repeated execution of a block.

```
for a < b {  
    a *= 2  
}  
  
for i := 0; i < 10; i++ {  
    f(i)  
}
```

Return statements

A "return" statement in a function F terminates the execution of F, and optionally provides one or more result values.

```
func noResult() {  
    return  
}
```