

# 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

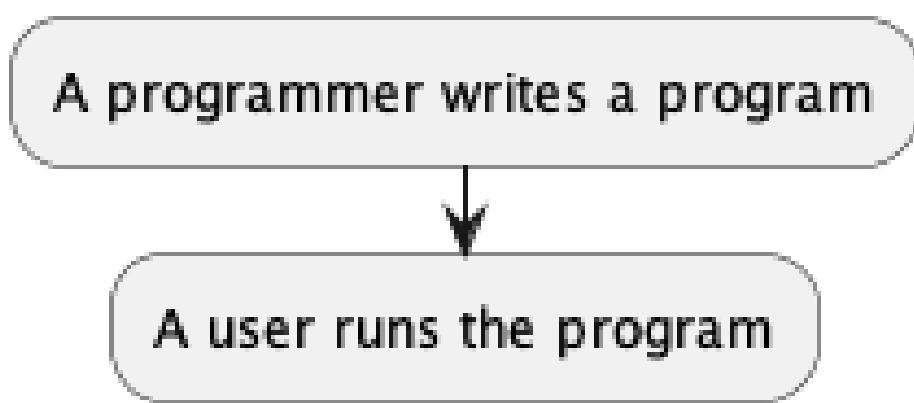
## **Softwareentwickler:innen bauen Maschinen**

- 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)

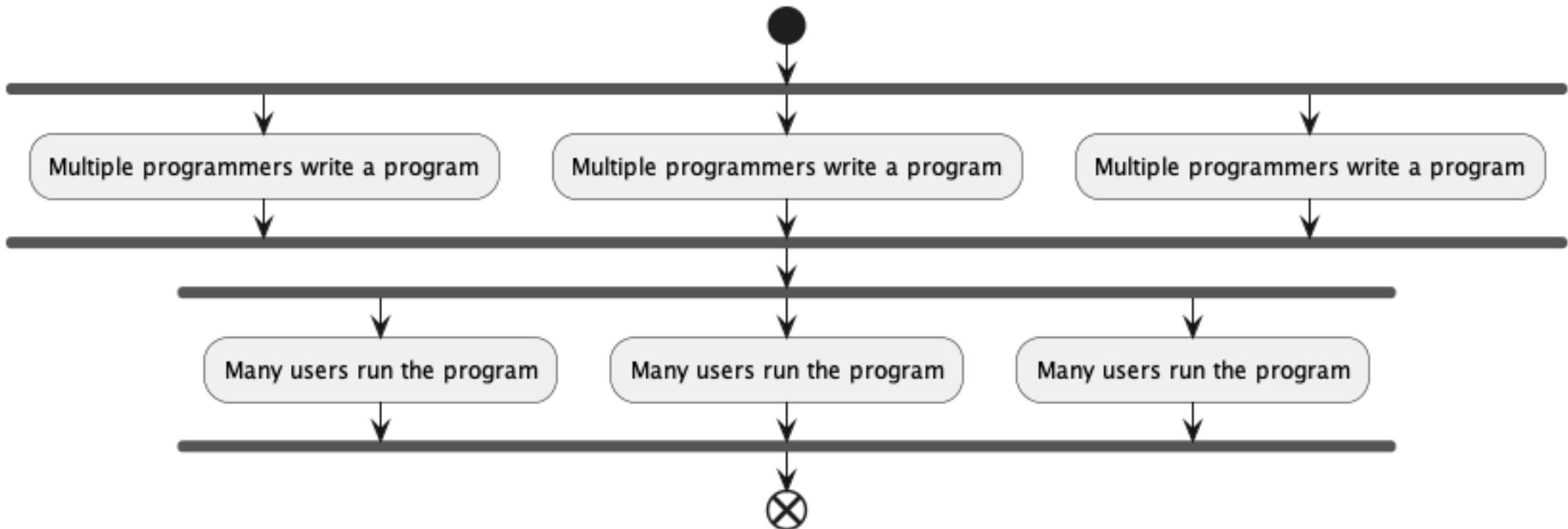
# Computer

- 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.

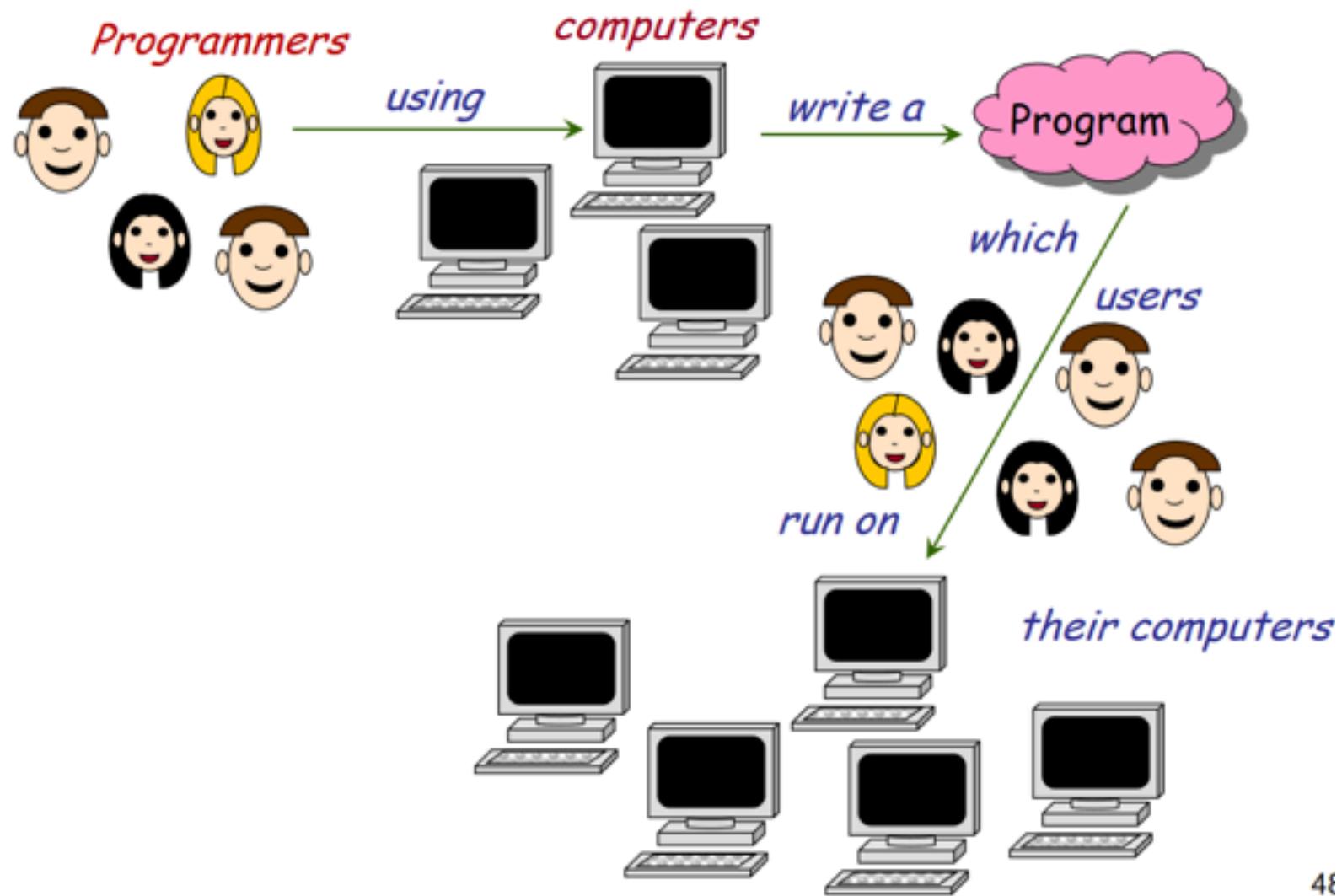
# Programme erstellen und laufen lassen



# 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**

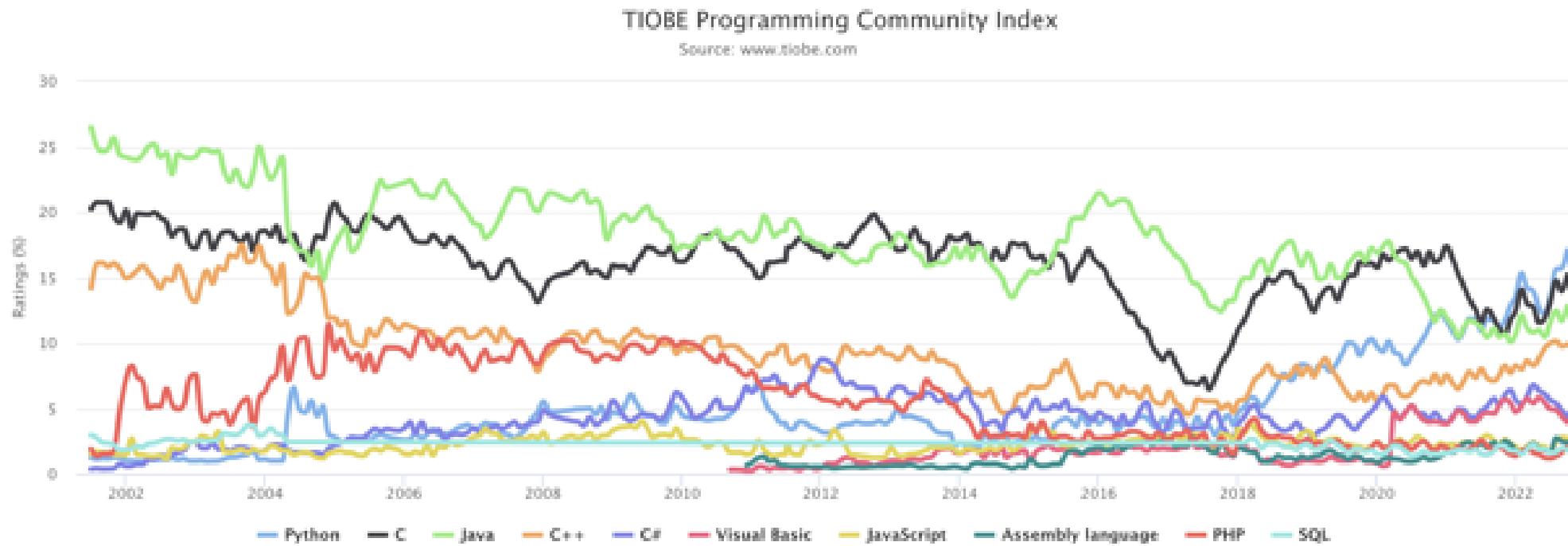
- Es ist schwierig, das Programm korrekt zu schreiben
- Trial-and-error ist ineffizient

## **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

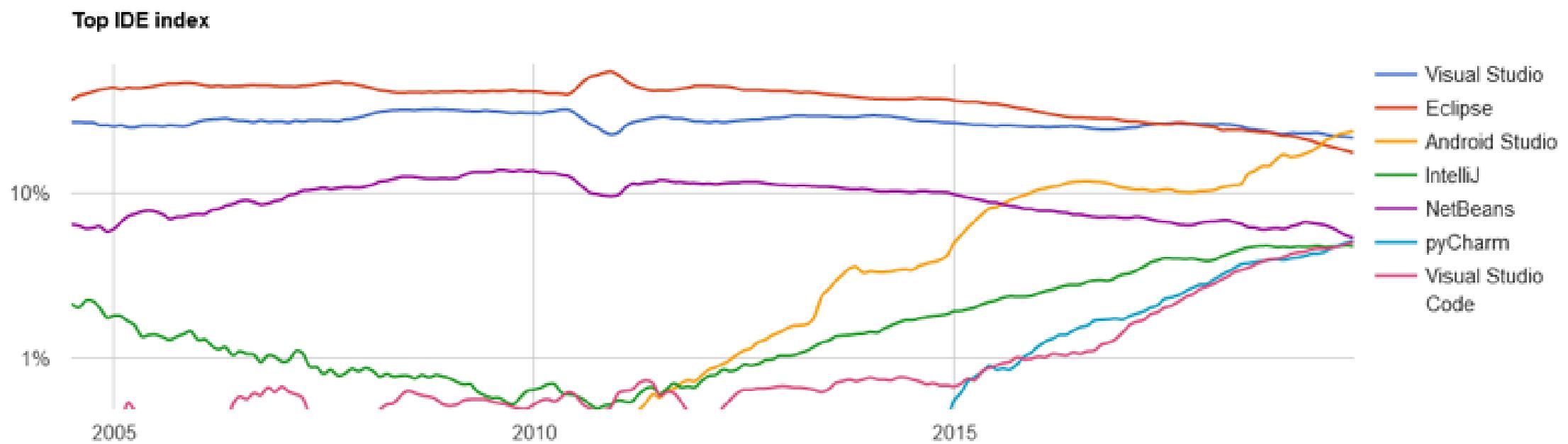
# Go

- 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



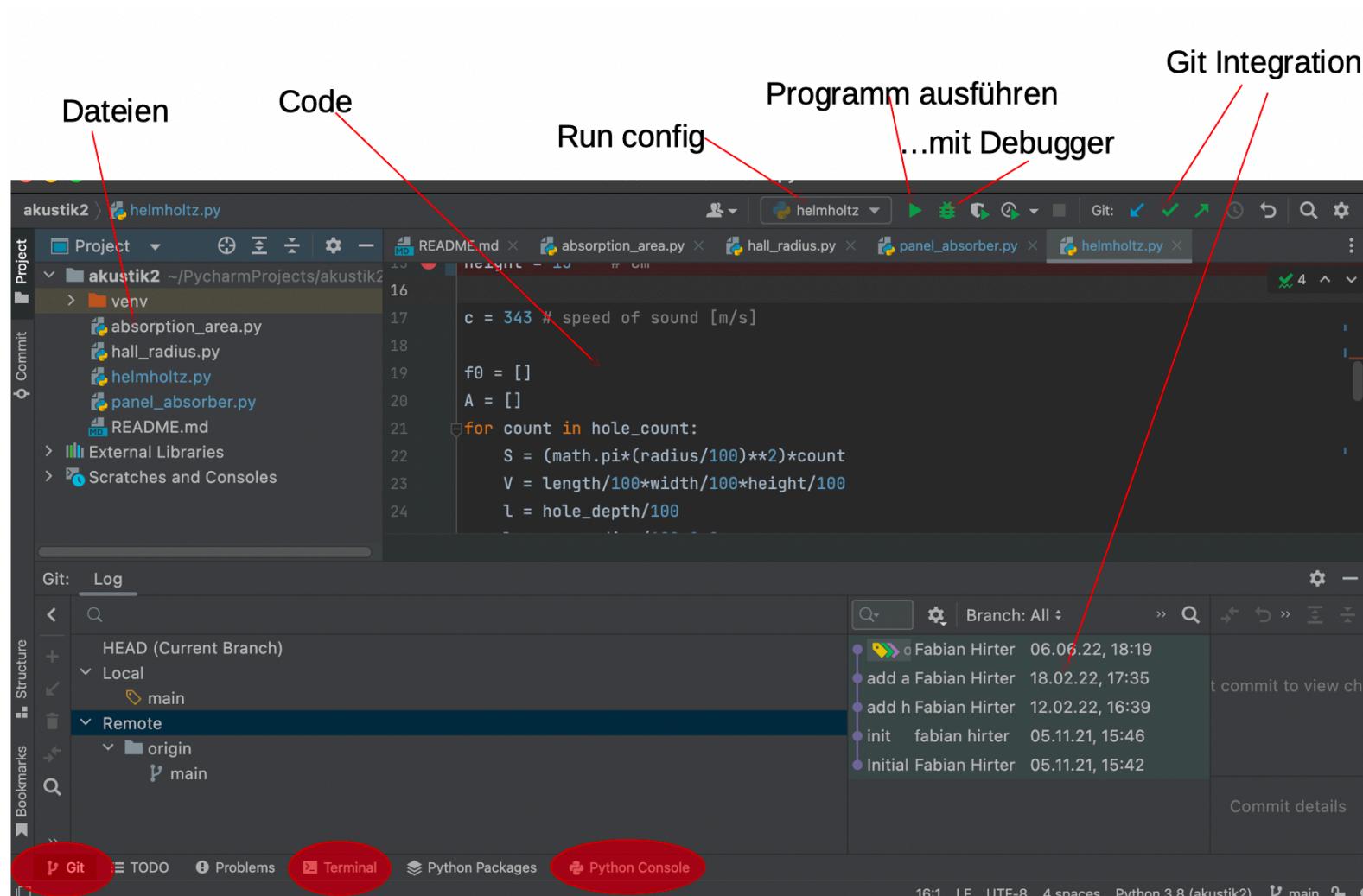
## **Microsoft Visual Studio Code**

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

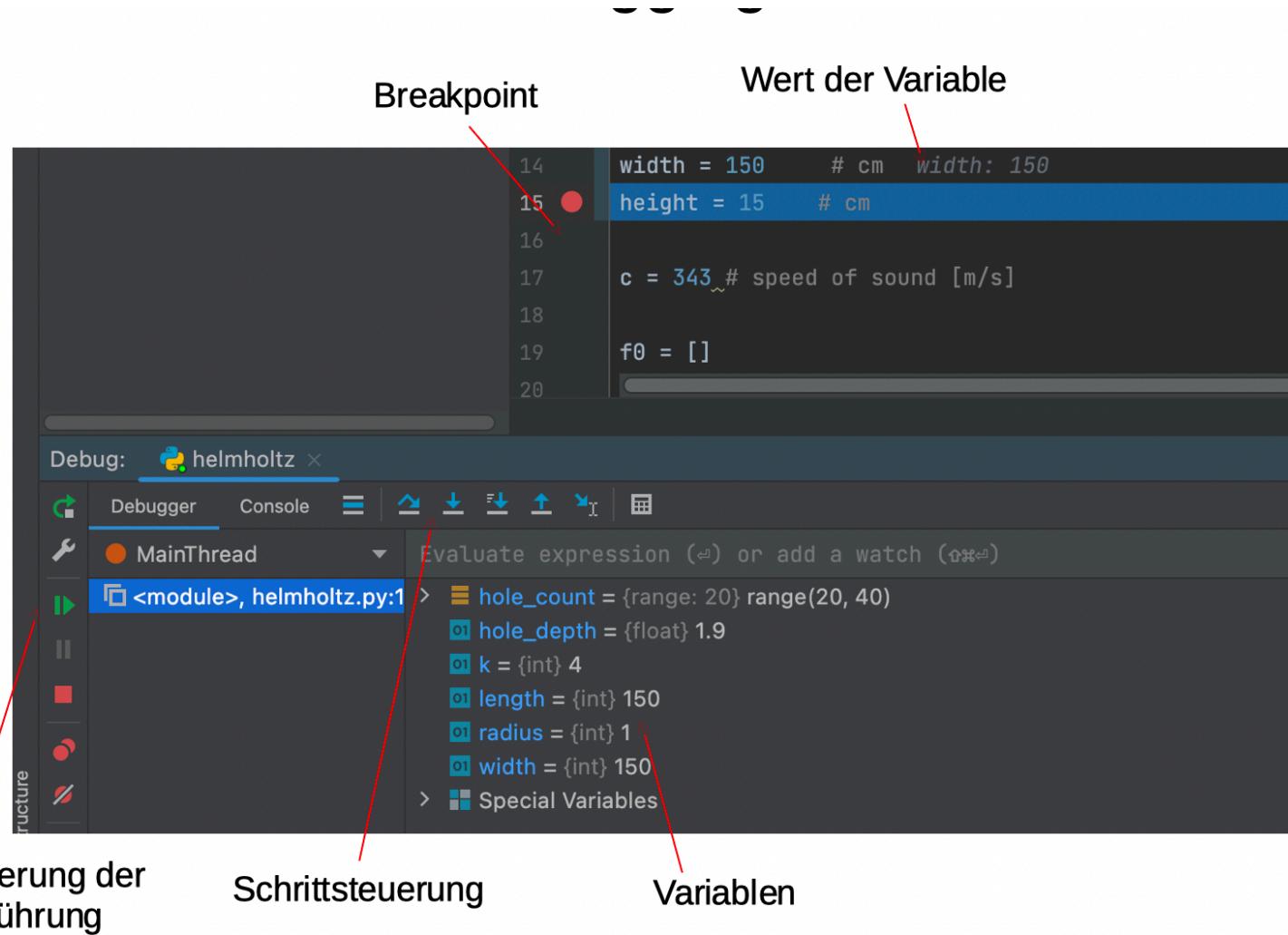
## **JetBrains**

- 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

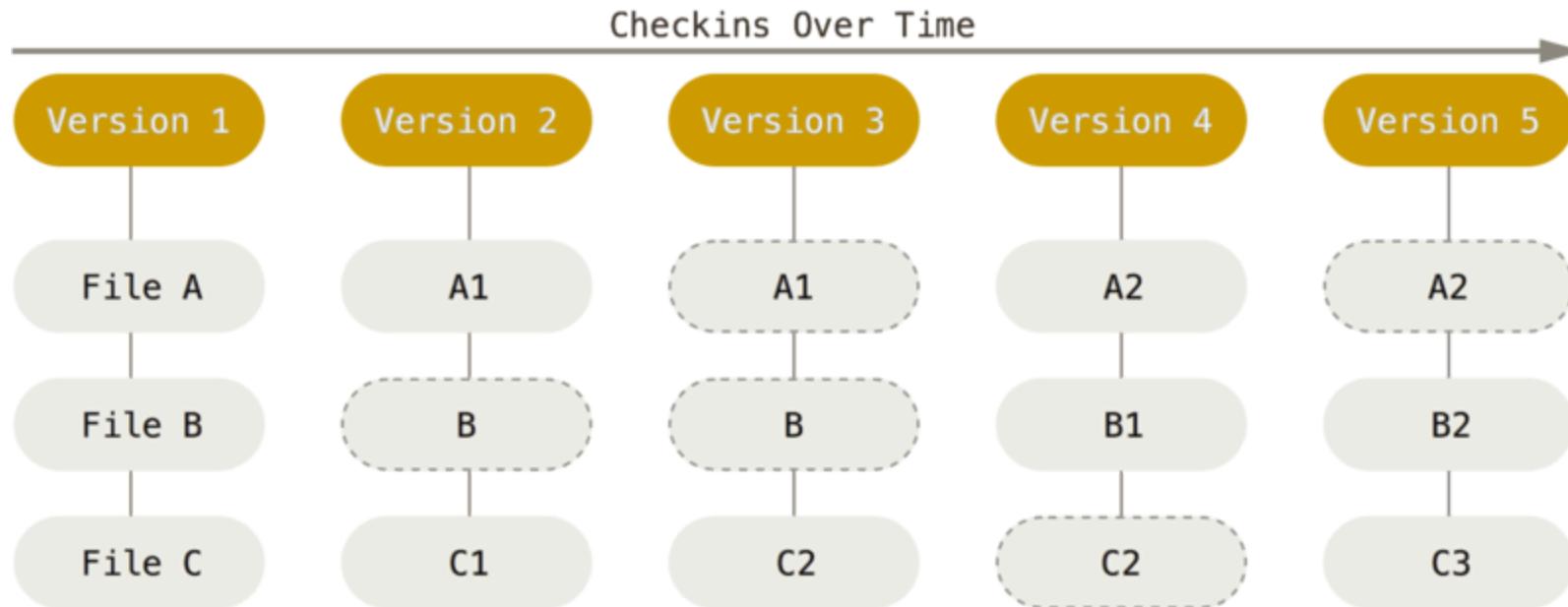
- Protokollierung von Änderungen
- Wiederherstellung von alten Ständen
- Archivierung
- Koordinierung des gemeinsamen Zugriffs
- Entwicklungszweige (Branches) -> **Don't Branch!**

# Moderne Versionsverwaltung

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

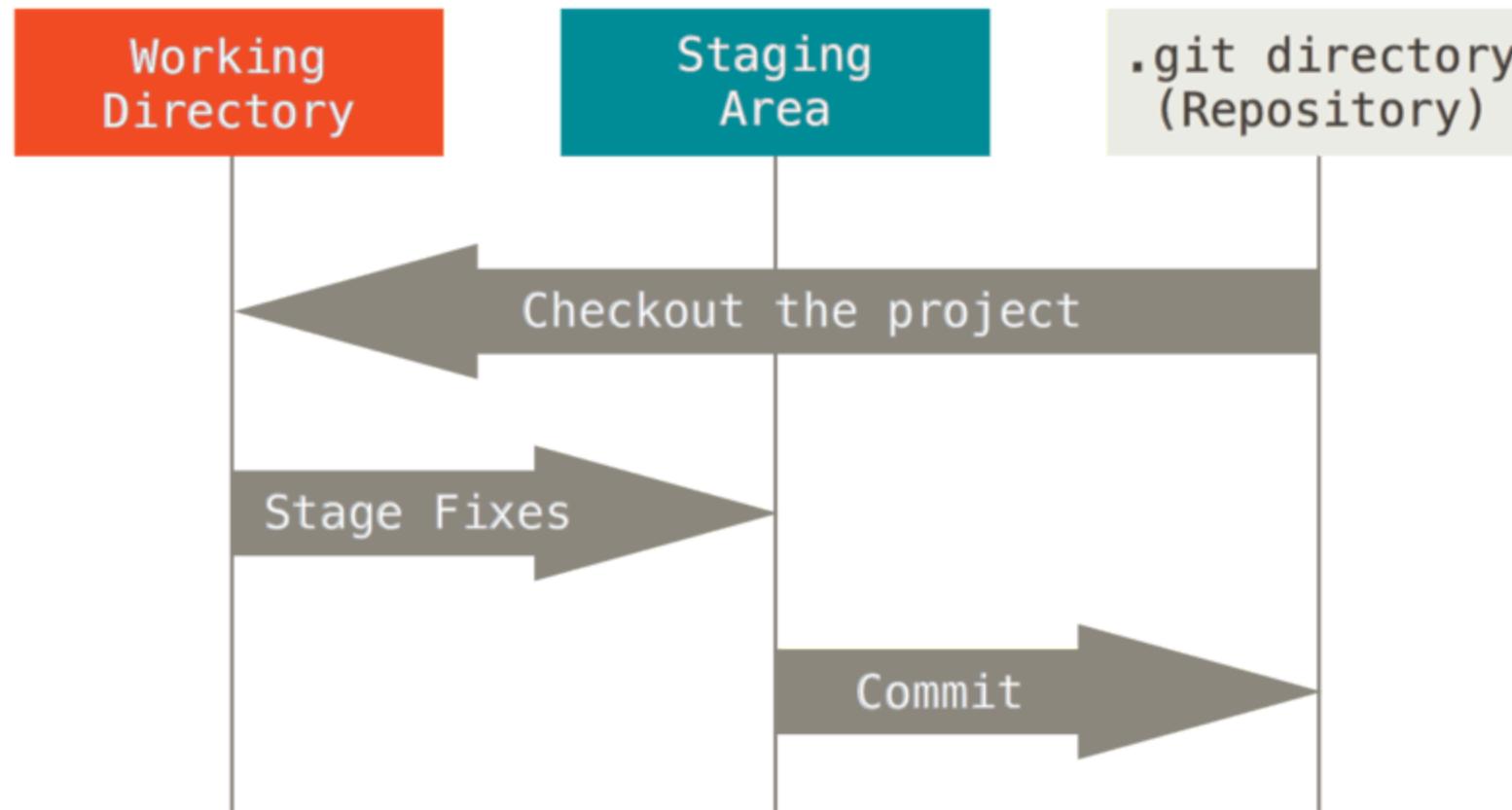
# Git

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

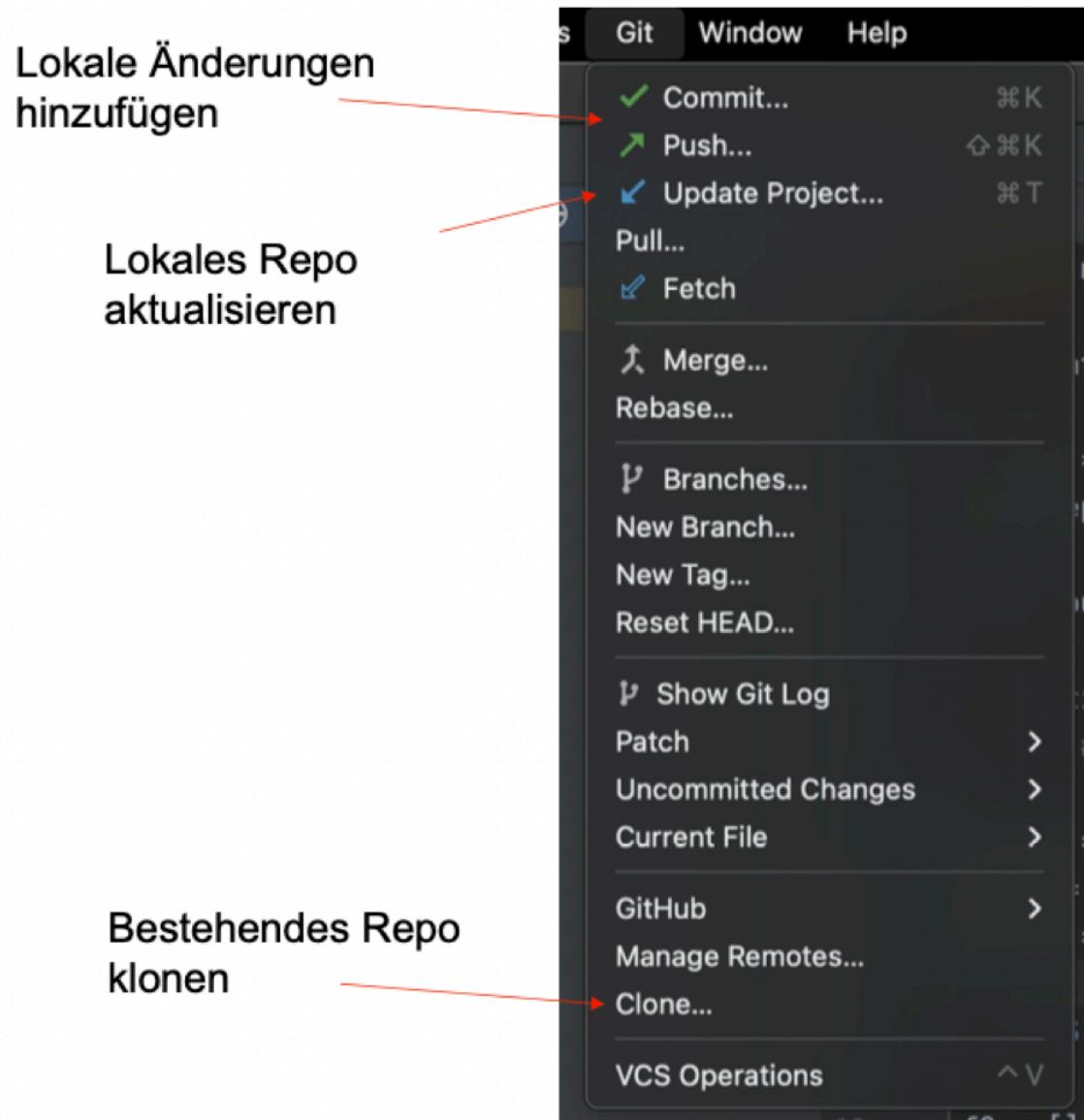
```
feat(logging): added logs for failed signups
```

```
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

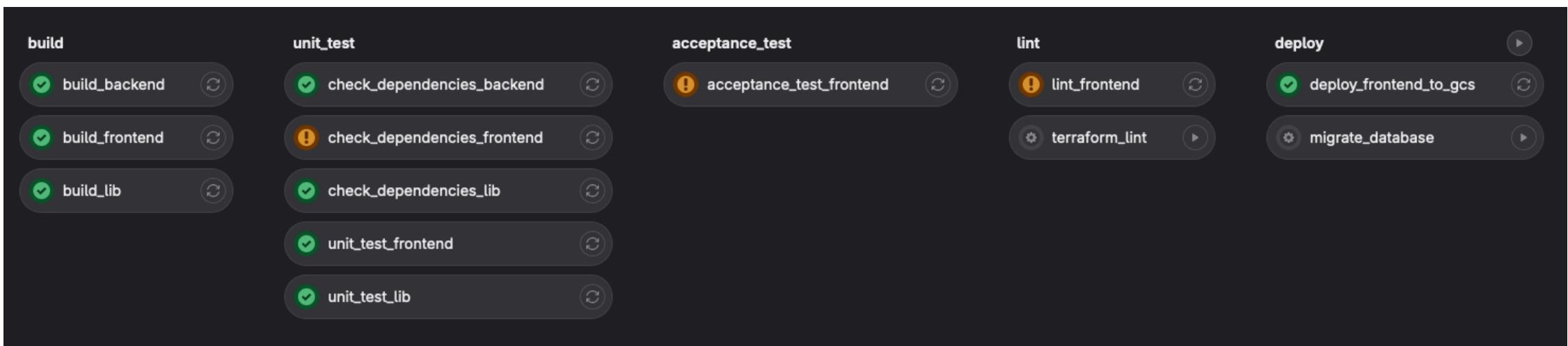
- [Cheatsheet](#)
- [Atlassian Tutorials](#)
- [Git Tutorials](#)
- [Simulationstool](#)

# 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)



# Objektorientierte Programmierung

1. Everything is an object,
2. Objects communicate by sending and receiving messages (in terms of objects),
3. Objects have their own memory (in terms of objects),
4. Every object is an instance of a class (which must be an object),
5. The class holds the shared behavior for its instances (in the form of objects in a program list),
6. To eval a program list, control is passed to the first object and the remainder is treated as its message

1. Alles ist ein Objekt,
  2. Objekte kommunizieren durch das Senden und Empfangen von Nachrichten (welche aus Objekten bestehen),
  3. Objekte haben ihren eigenen Speicher (strukturiert als Objekte),
  4. Jedes Objekt ist die Instanz einer Klasse (welche ein Objekt sein muss),
  5. Die Klasse beinhaltet das Verhalten aller ihrer Instanzen (in der Form von Objekten in einer Programmliste),
  6. Um eine Programmliste auszuführen, wird die Ausführungskontrolle dem ersten Objekt gegeben und das Verbleibende als dessen Nachricht behandelt
- Alan Kay: The Early History of Smalltalk (1993)

**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.')
```

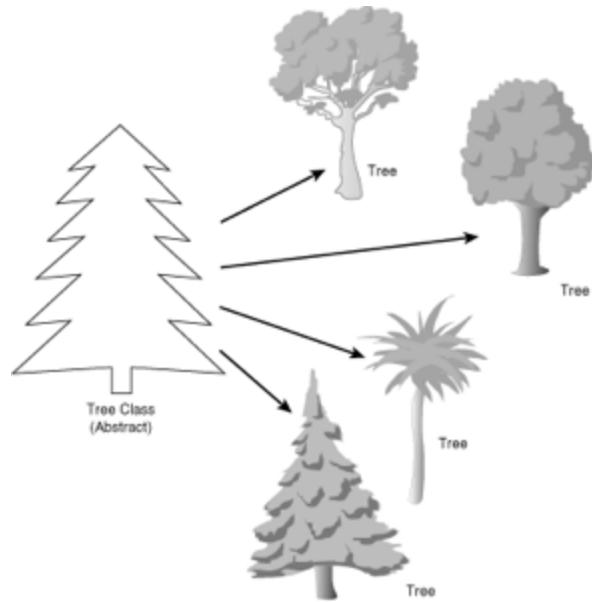
# Klasse

- 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.

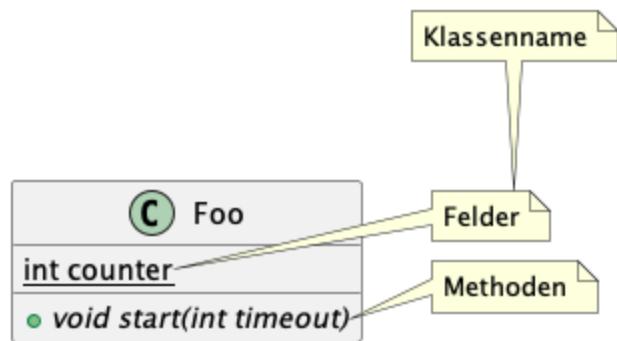
# Schnittstellen definieren

- 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

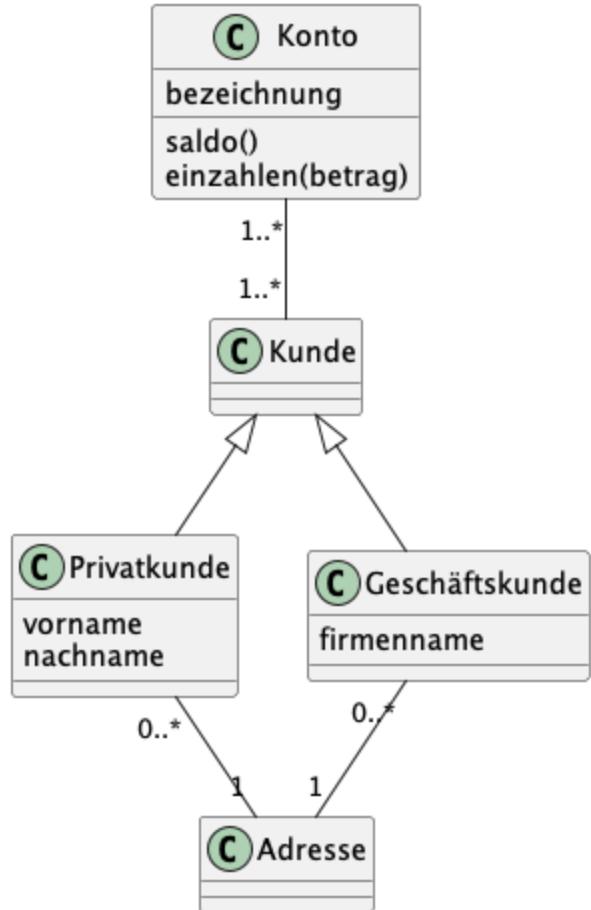
```
/**  
 * Returns an Image object that can then be painted on the screen.  
 * The url argument must specify an absolute {@link URL}. The name  
 * argument is a specifier that is relative to the url argument.  
 * <p>  
 * This method always returns immediately, whether or not the  
 * image exists. 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
}

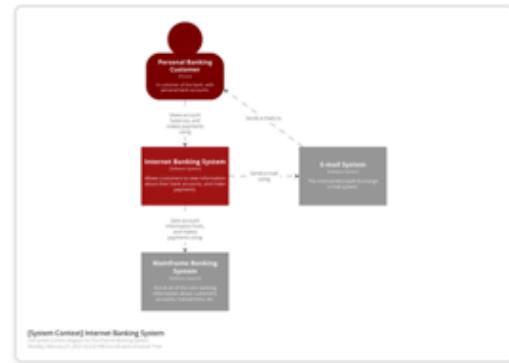
class Adresse {}

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

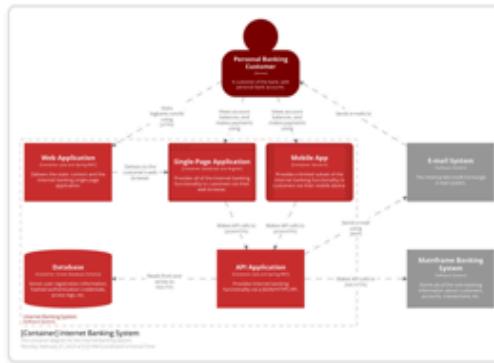
Privatkunde "0..*" -- "1" Adresse
Geschäftskunde "0..*" -- "1" Adresse

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

# C4 Model



Level 1: A **System Context** diagram provides a starting point, showing how the software system in scope fits into the world around it.



Level 2: A **Container** diagram zooms into the software system in scope, showing the high-level technical building blocks.



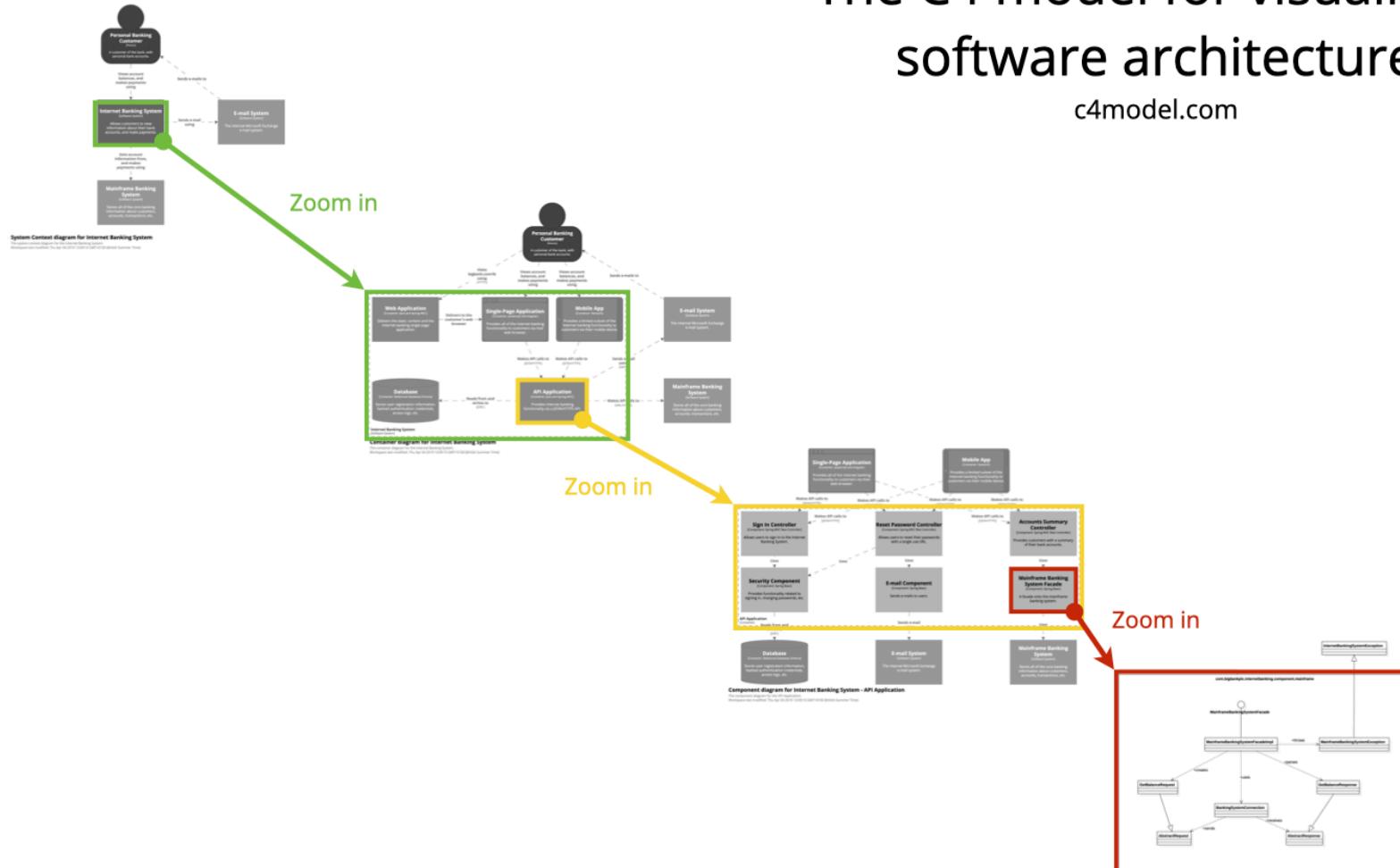
Level 3: A **Component** diagram zooms into an individual container, showing the components inside it.



Level 4: A **code** (e.g. UML class) diagram can be used to zoom into an individual component, showing how that component is implemented.

# The C4 model for visualising software architecture

c4model.com

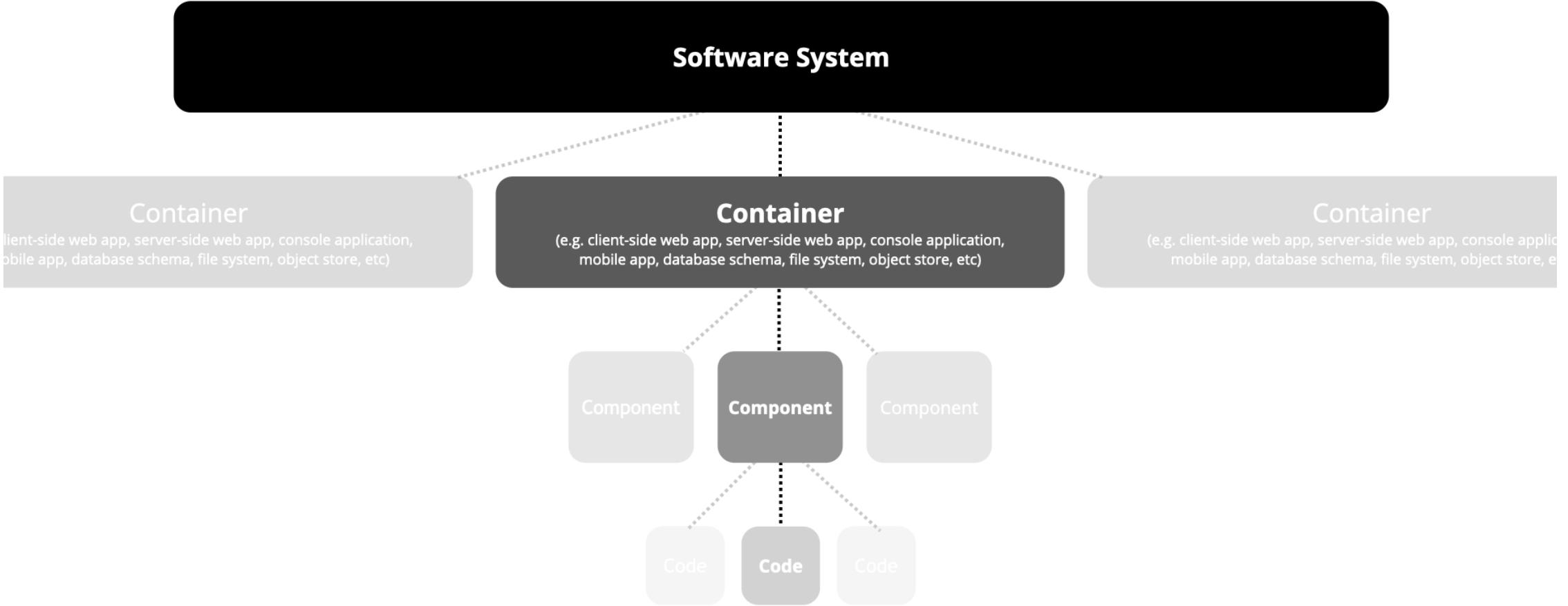


Level 1  
Context

Level 2  
Containers

Level 3  
Components

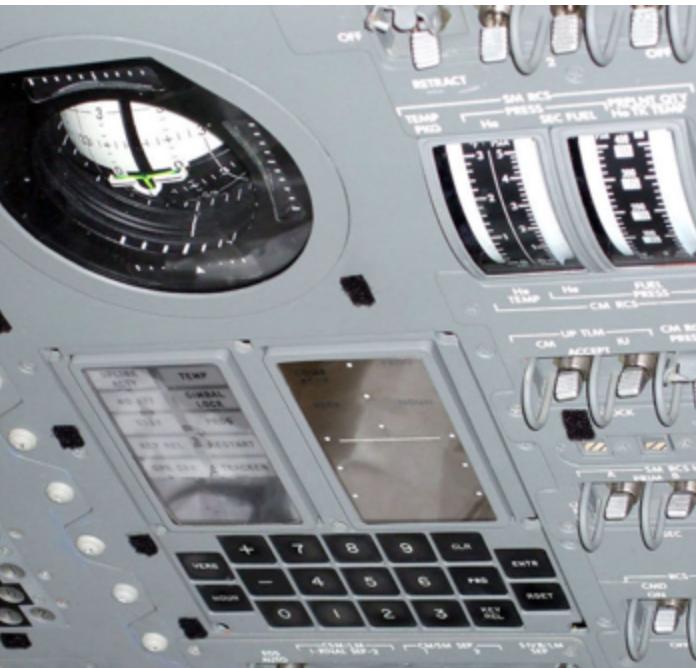
Level 4  
Code



A **software system** is made up of one or more **containers** (applications and data stores), each of which contains one or more **components**, which in turn are implemented by one or more **code** elements (classes, interfaces, objects, functions, etc).

# 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

## "Soft"-Ware

- Softwareentwicklung ist meistens Kreativarbeit
- Die Herausforderung der "Produktion" existiert kaum

# Lösungen

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

## Alles hängt zusammen

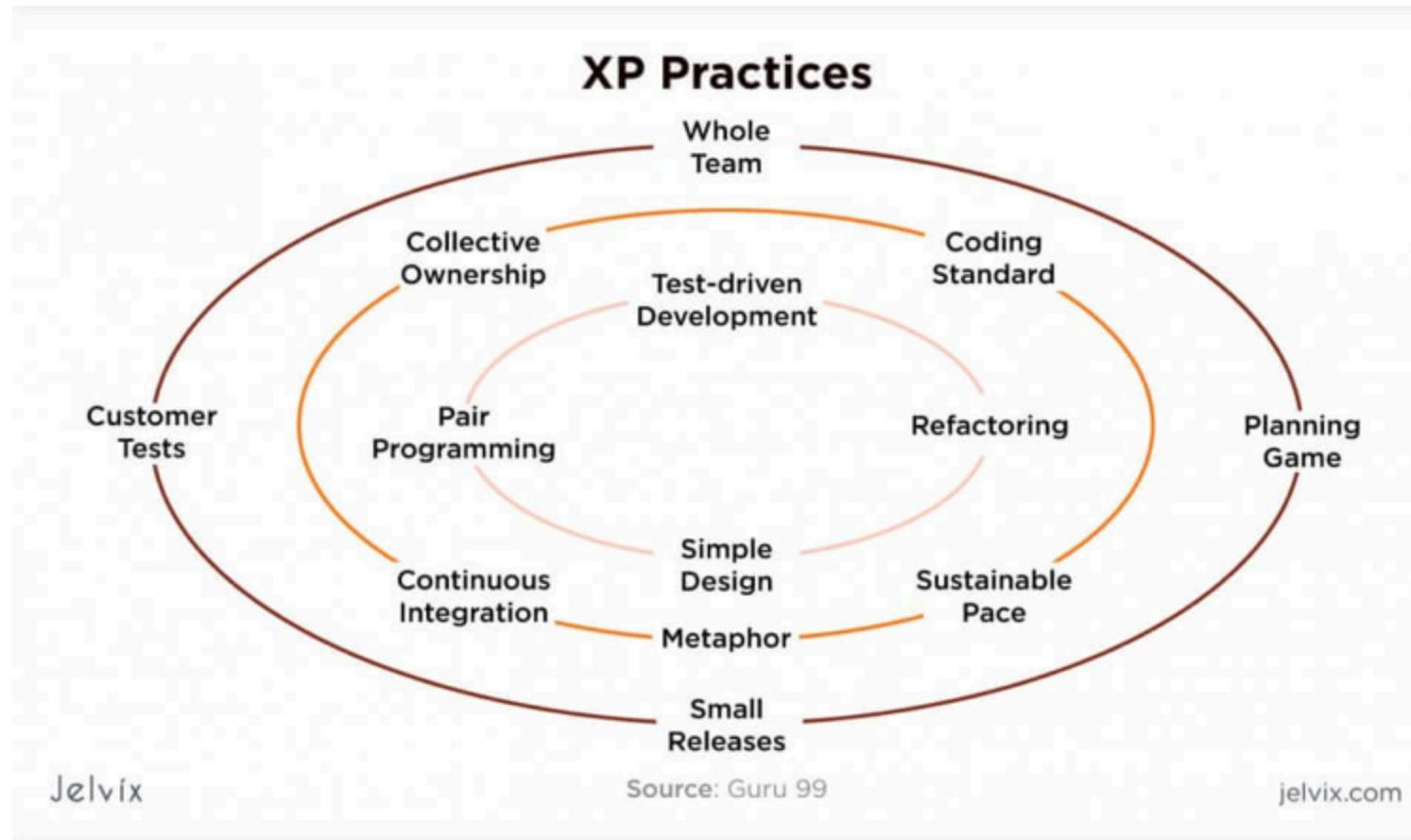
- Hohe Qualität reduziert Komplexität
- Hohe Qualität kürzt den Feedbackloop durch schnelle Entwicklung
- Kleine Arbeitspakete kürzen den Feedbackloop
- Kleine Arbeitspakete reduzieren Komplexität
- ...

# 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/> (2001)

# Extreme Programming

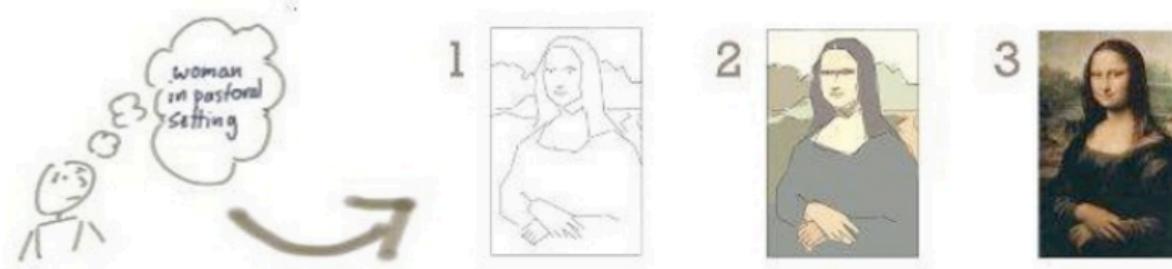


# Embrace Change



# Iteratives und inkrementelles Arbeiten

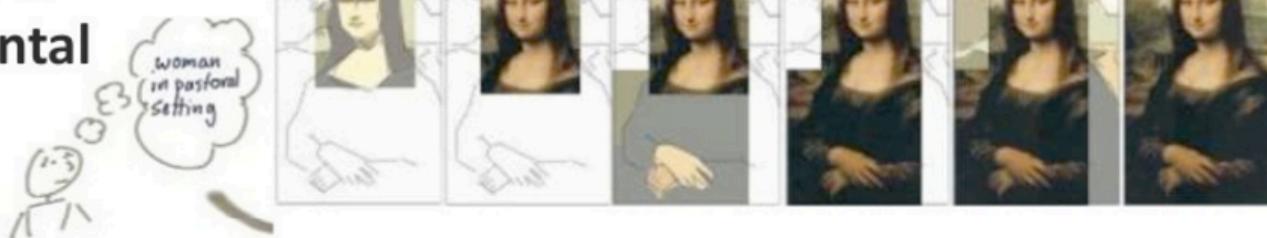
Iterative



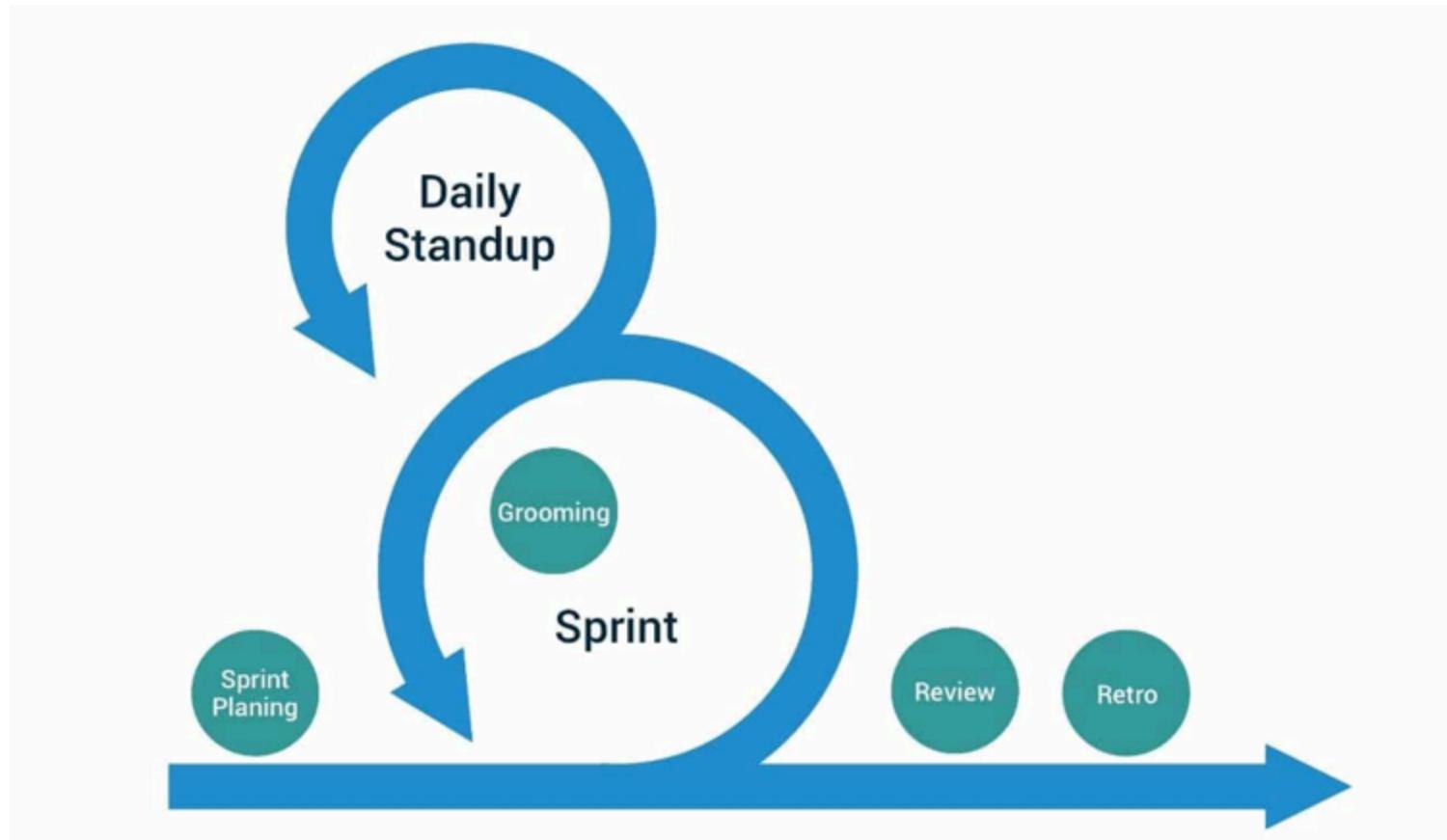
Incremental



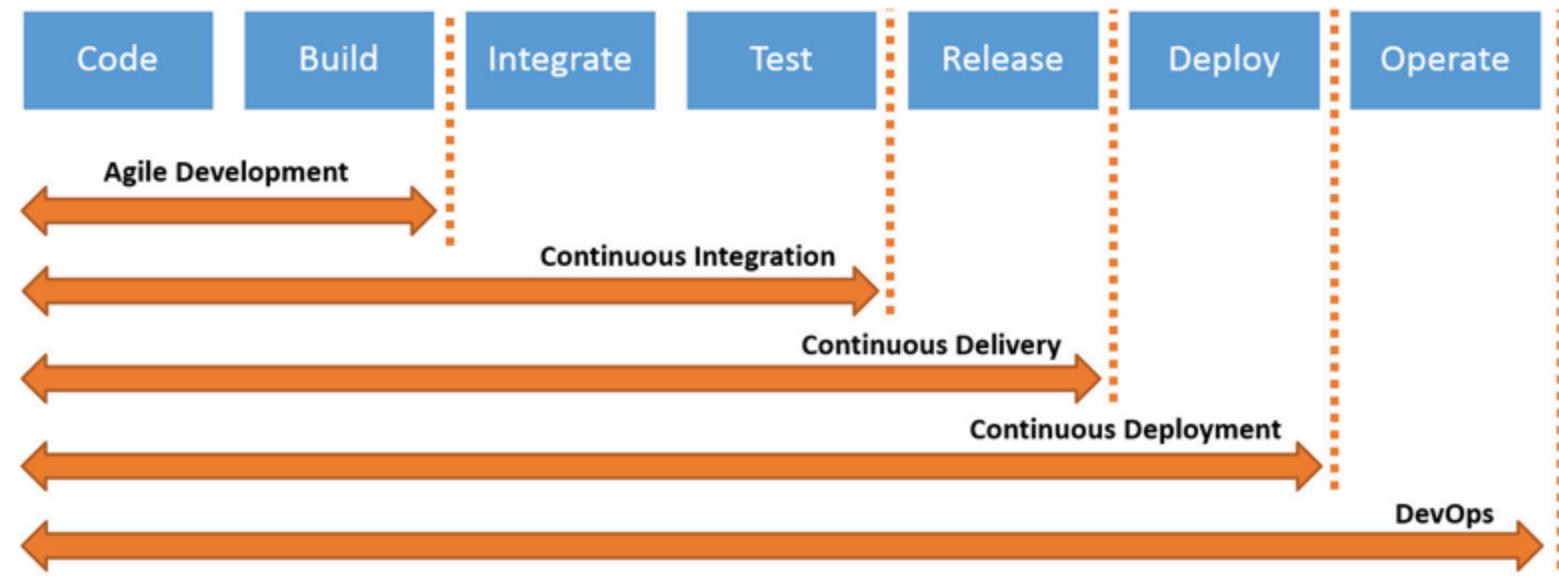
Iterative &  
Incremental



# Iterationen



# Kurzer Feedbackloop: CI/CD



# Continuous Integration

- Kein Branching, alle Änderungen werden von allen Teammitgliedern mehrmals täglich in den Master Branch eingeccheckt.
- Dieser Branch ist jederzeit lauffähig
- Dadurch werden die Releases vereinfachen
- Eine sehr hohe, automatische Testabdeckung ist zwingend

# Continuous Delivery

- 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
- Higher quality
- Lower costs
- Better products
- Happier teams

# Principles

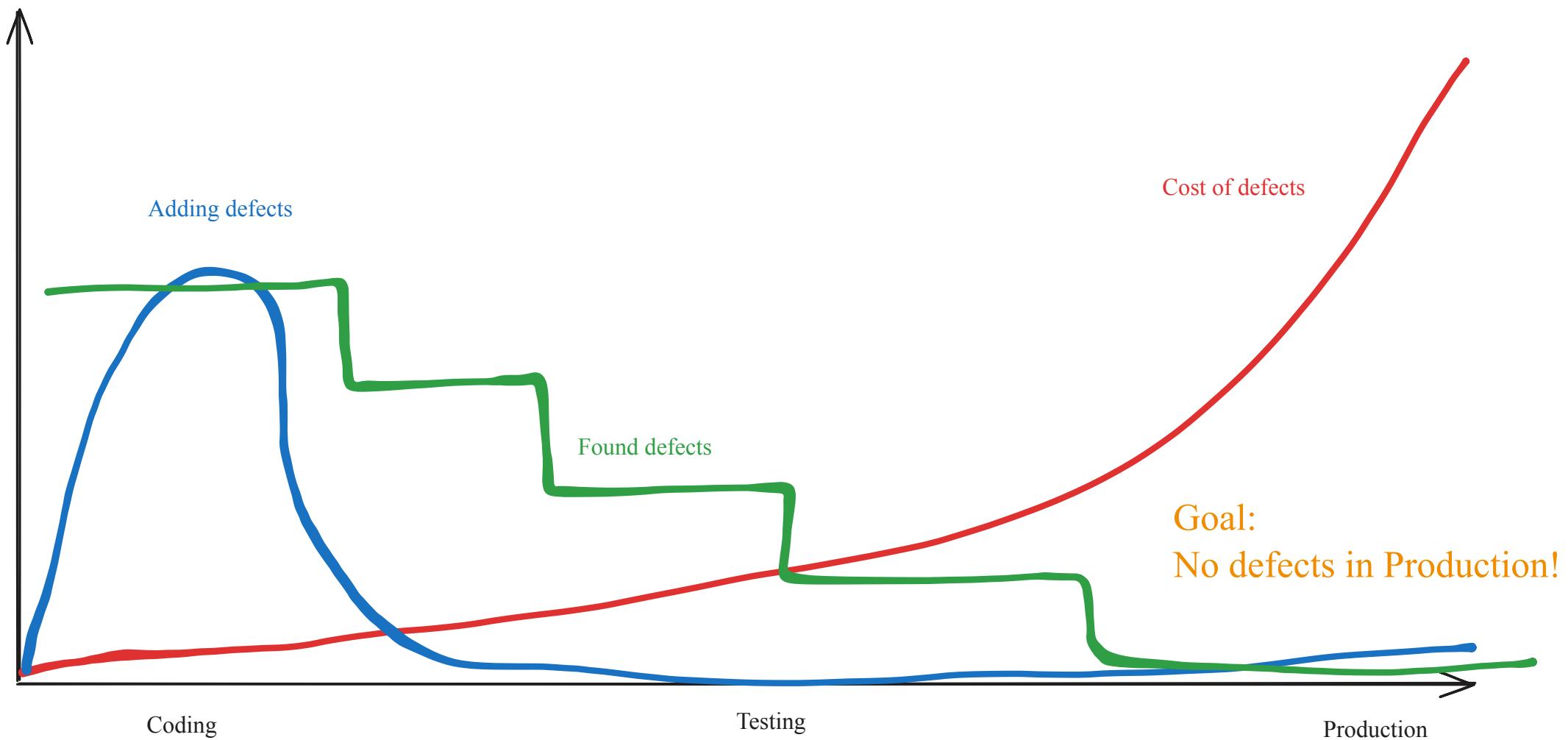
- Build quality in
- Work in small batches
- Computers perform repetitive tasks, people solve problems
- Relentlessly pursue continuous improvement
- Everyone is responsible

<https://www.continuousdelivery.com/>

Modern Software Engineering

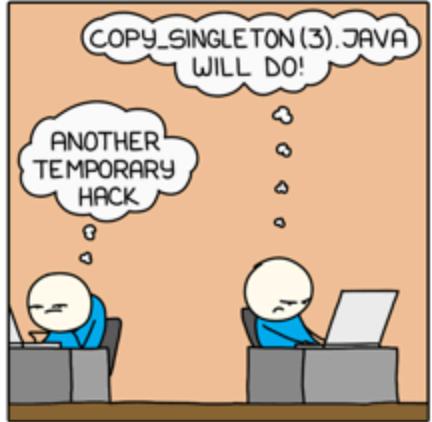
# Testing

# Kosten von Defekten

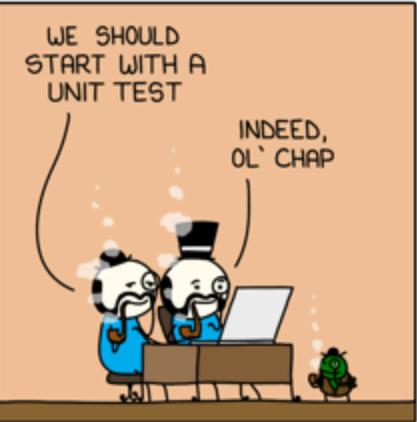
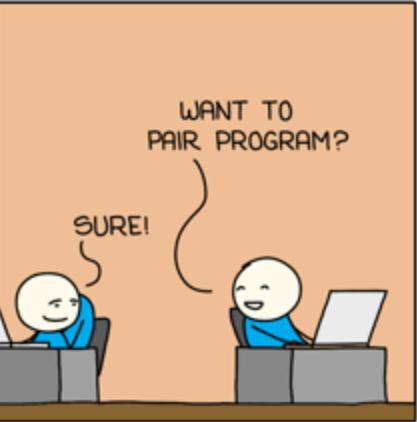


# Pair Programming

PAIR PROGRAMMING



MONKEYUSER.COM



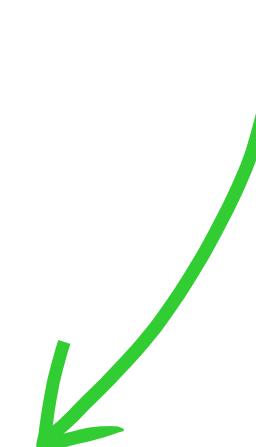
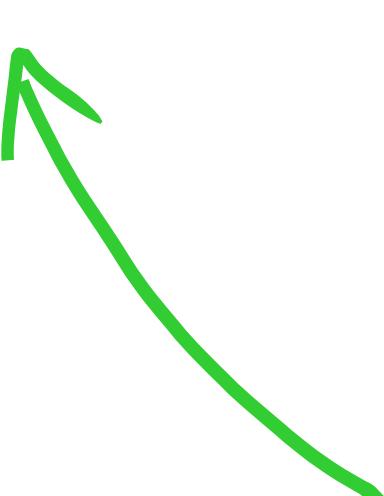
# Test Driven Development (TDD)

- 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

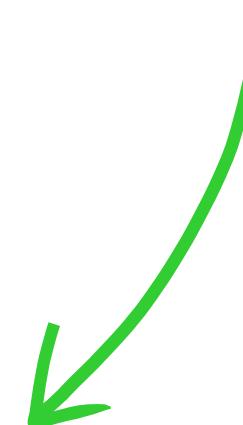
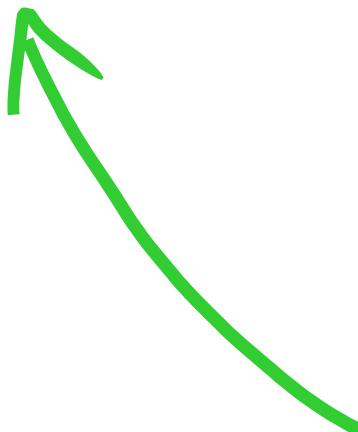
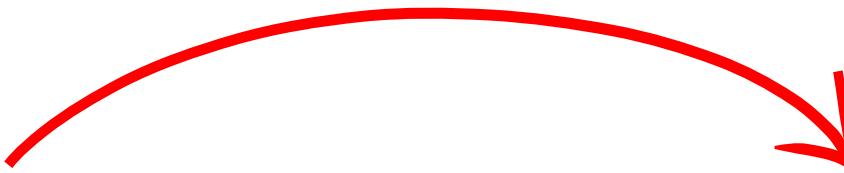


Write a  
failing  
test

Make the  
test pass

Refactor

Hard to write a test?

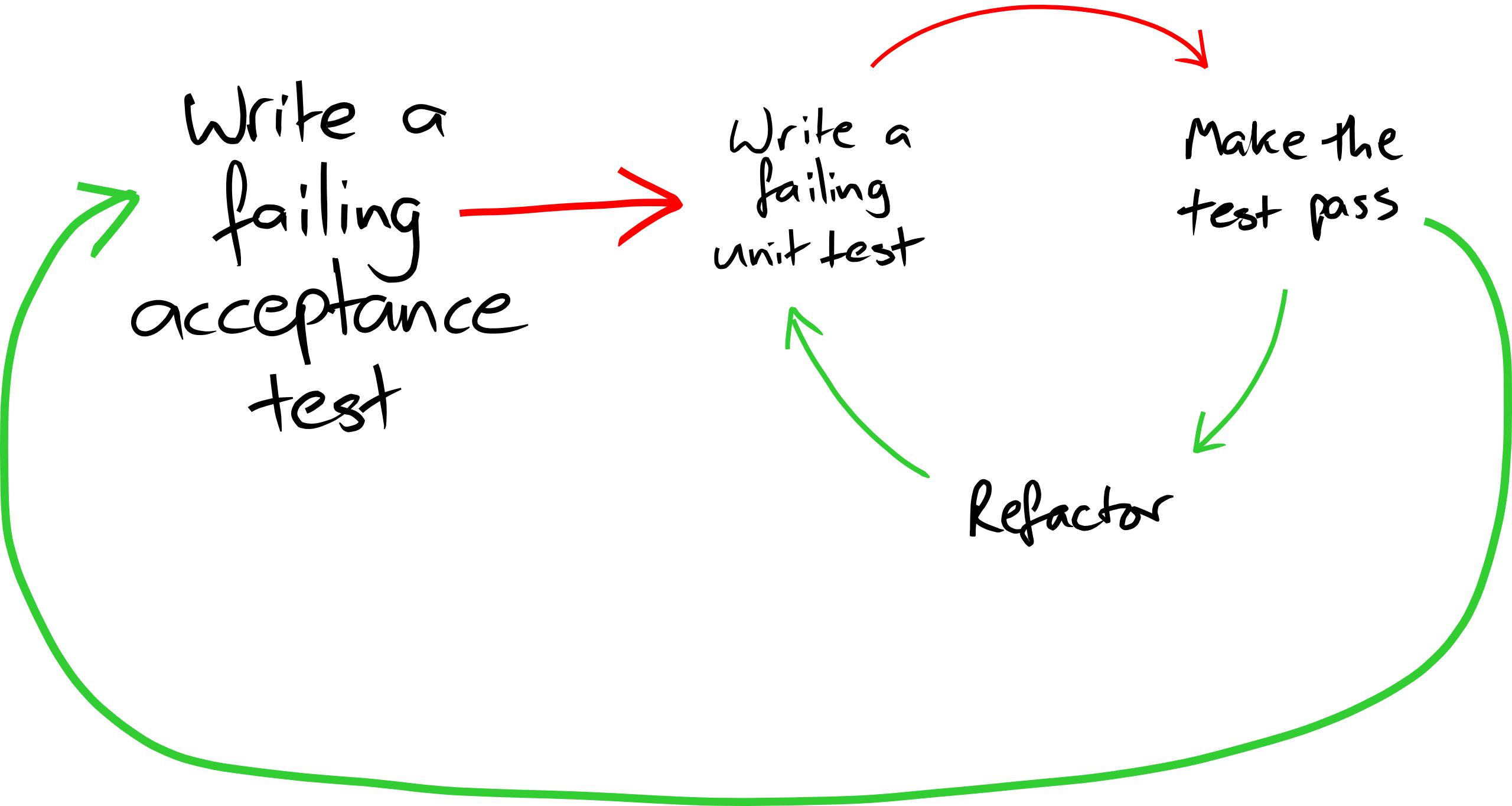


Write a failing acceptance test

Write a failing unit test

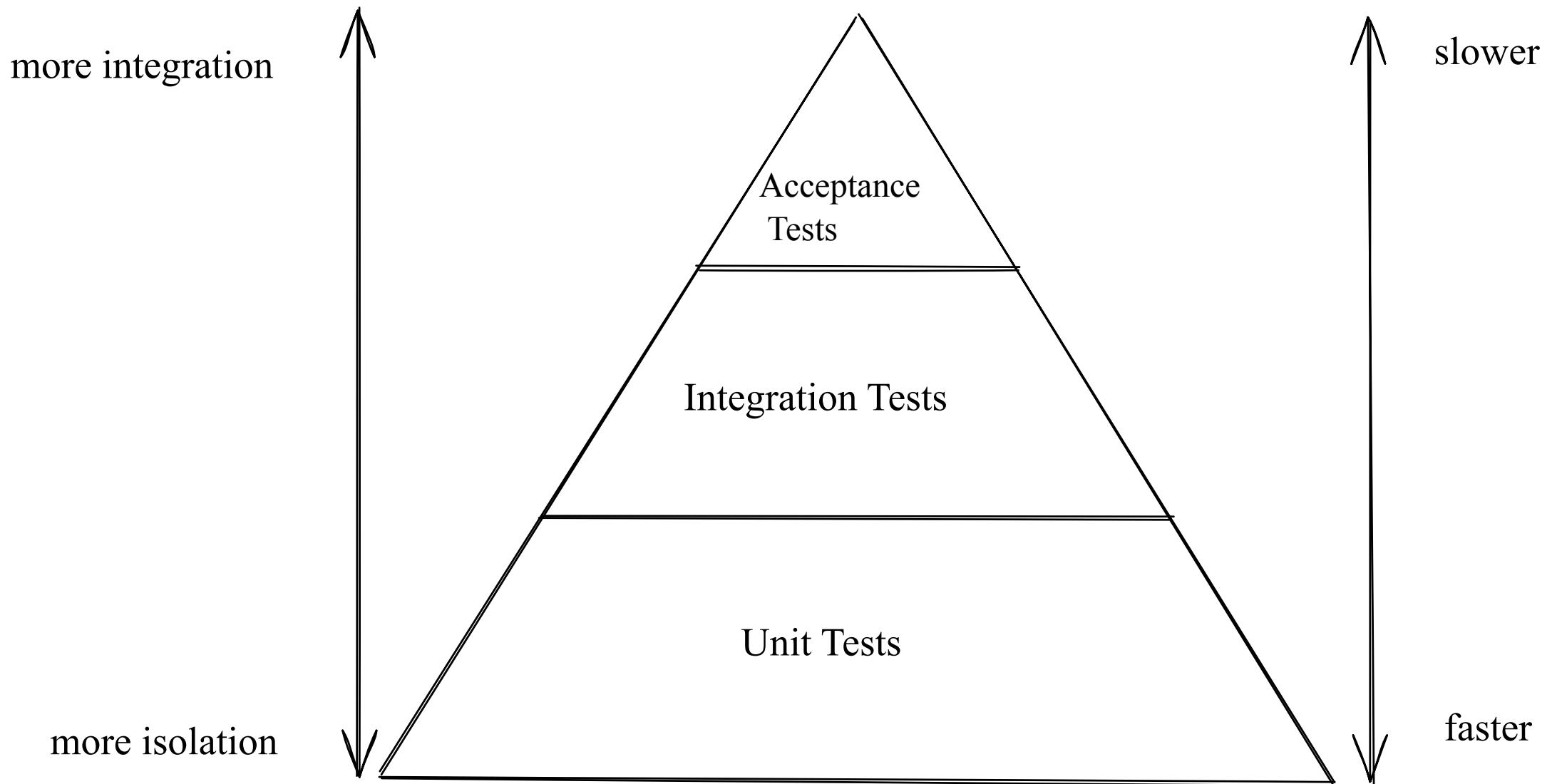
Make the test pass

Refactor



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

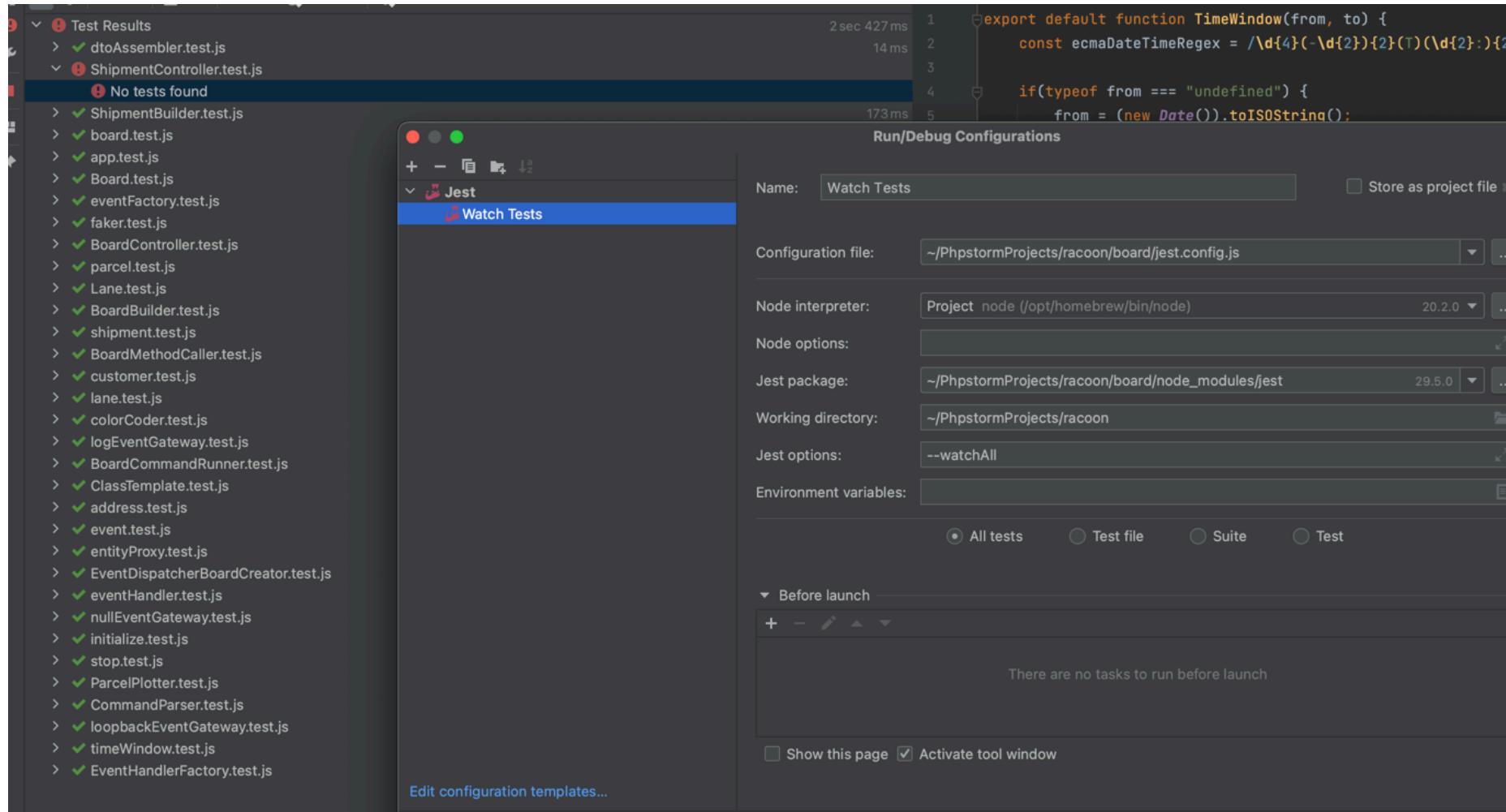
# Testpyramide



## **Testing: AAA**

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

# IDE Integration



## Testing: Further Reading

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

## 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

# 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

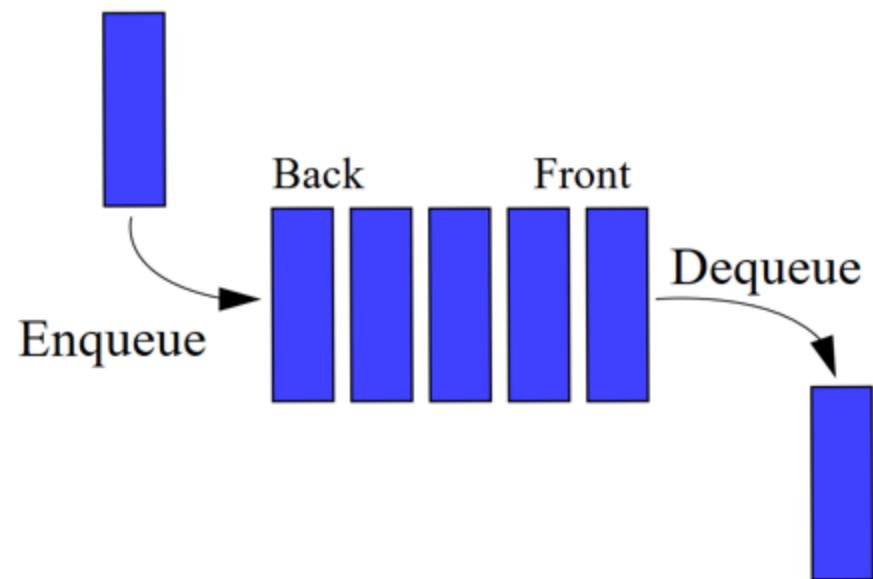
- Schlüssel / Wert Paare
- Hinzufügen, Entfernen, Ändern, Abrufen
- Assoziatives Array, Lookup Table, Dictionary

## Beispiel

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

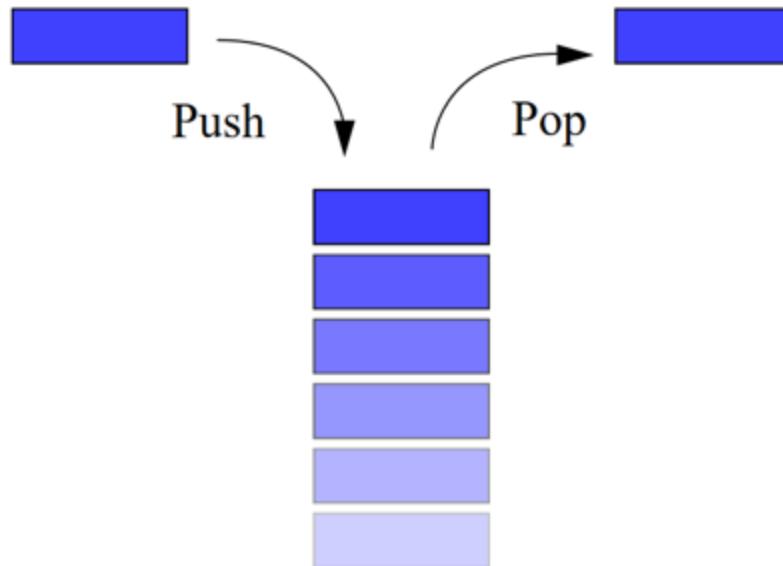
# Queue

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

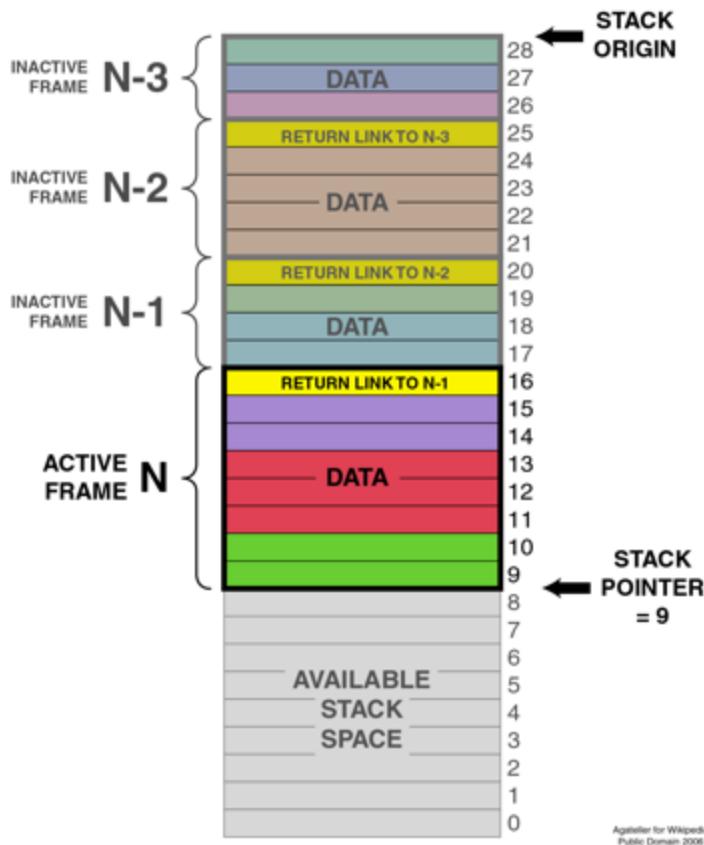


# Stack

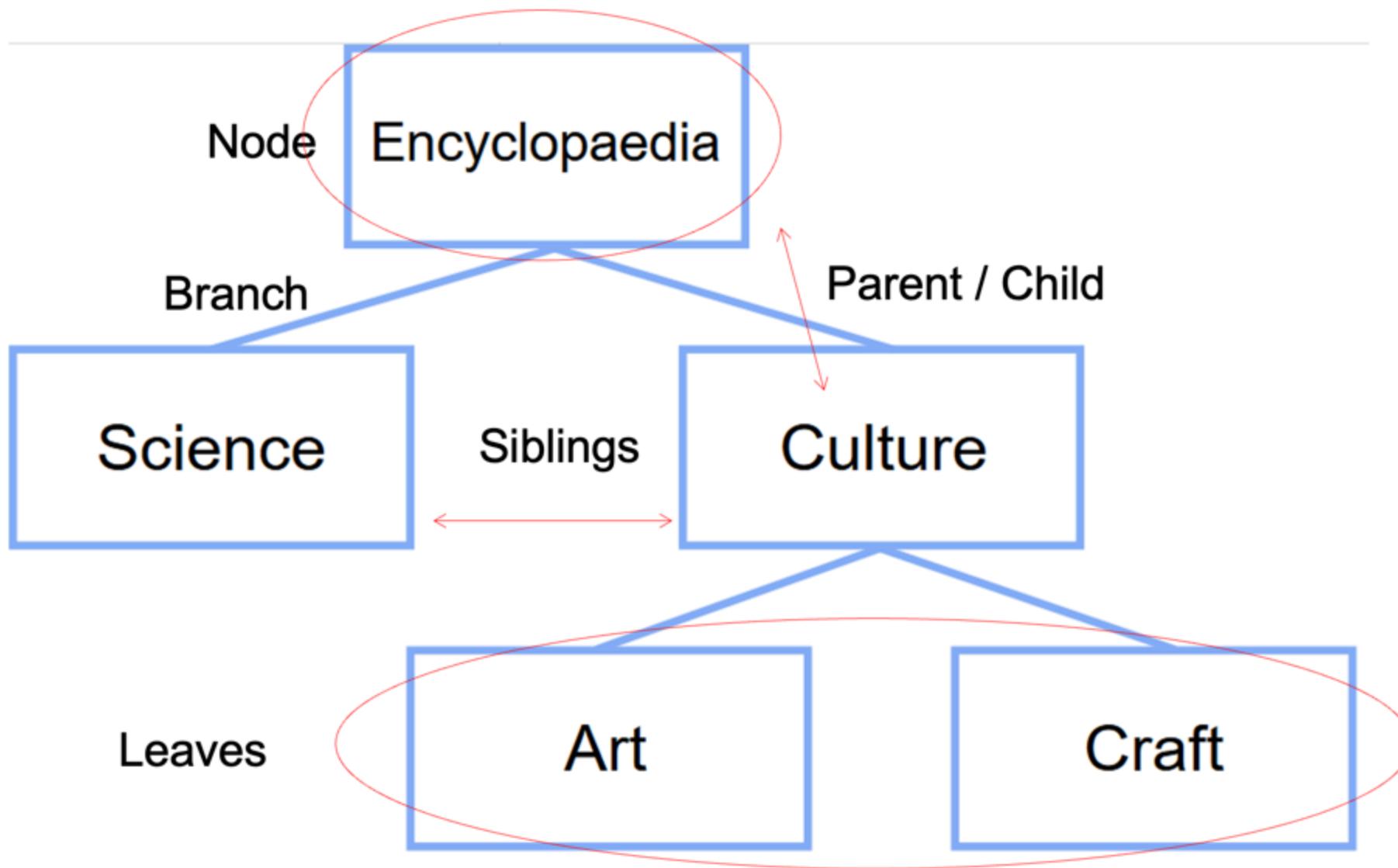
- LIFO: Last In, First Out
- push: Neues Element speichern
- pop: Letztes Element abrufen und entfernen
- Stapelspeicher, Kellerspeicher

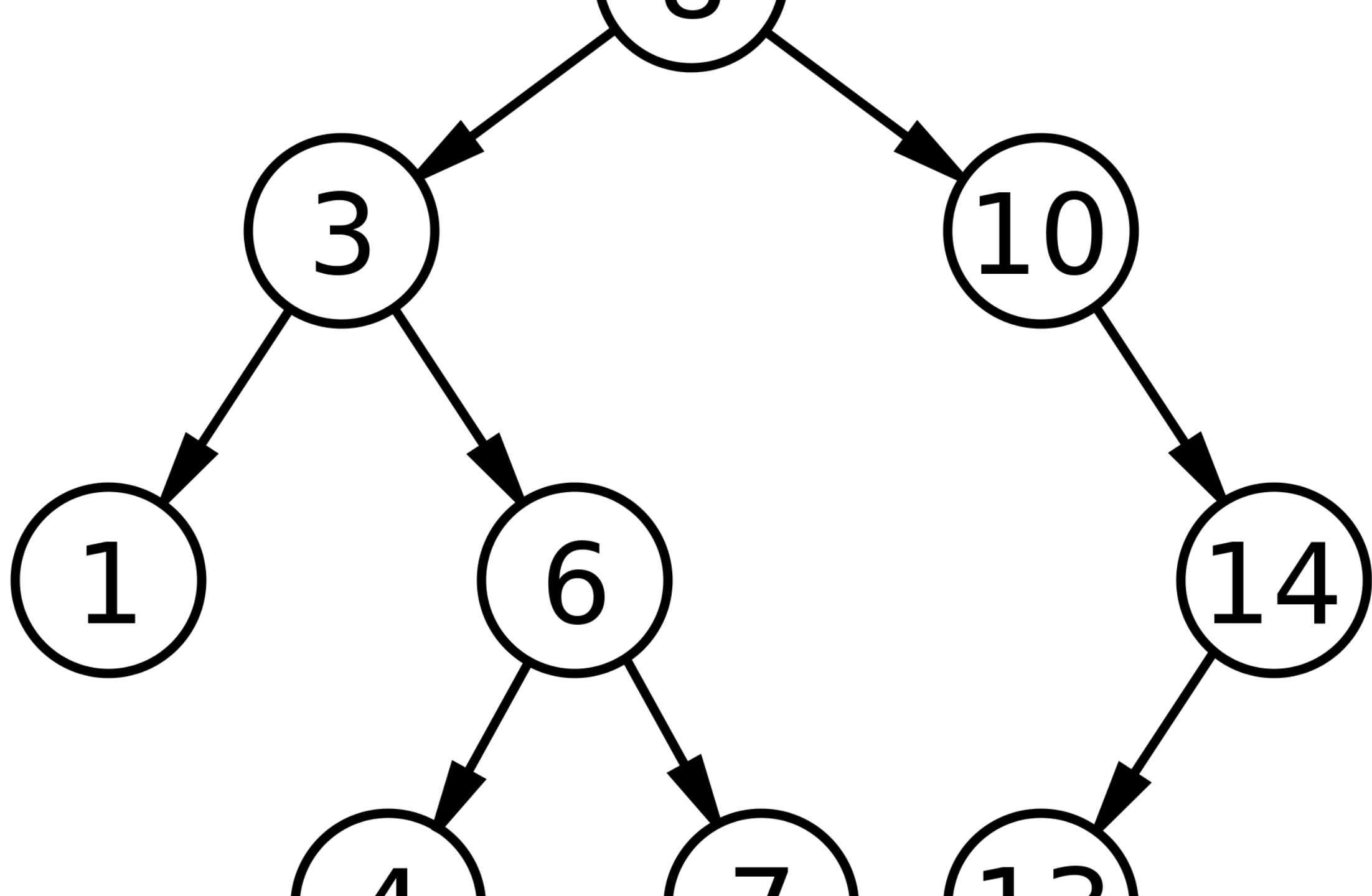


# Stack

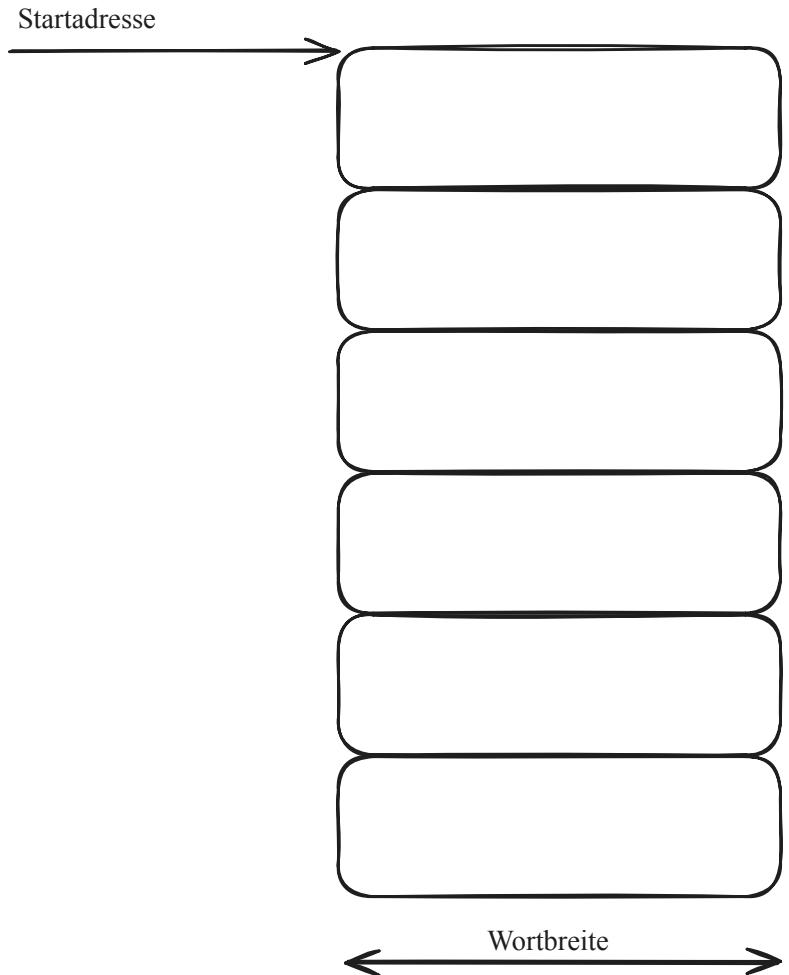


# Tree

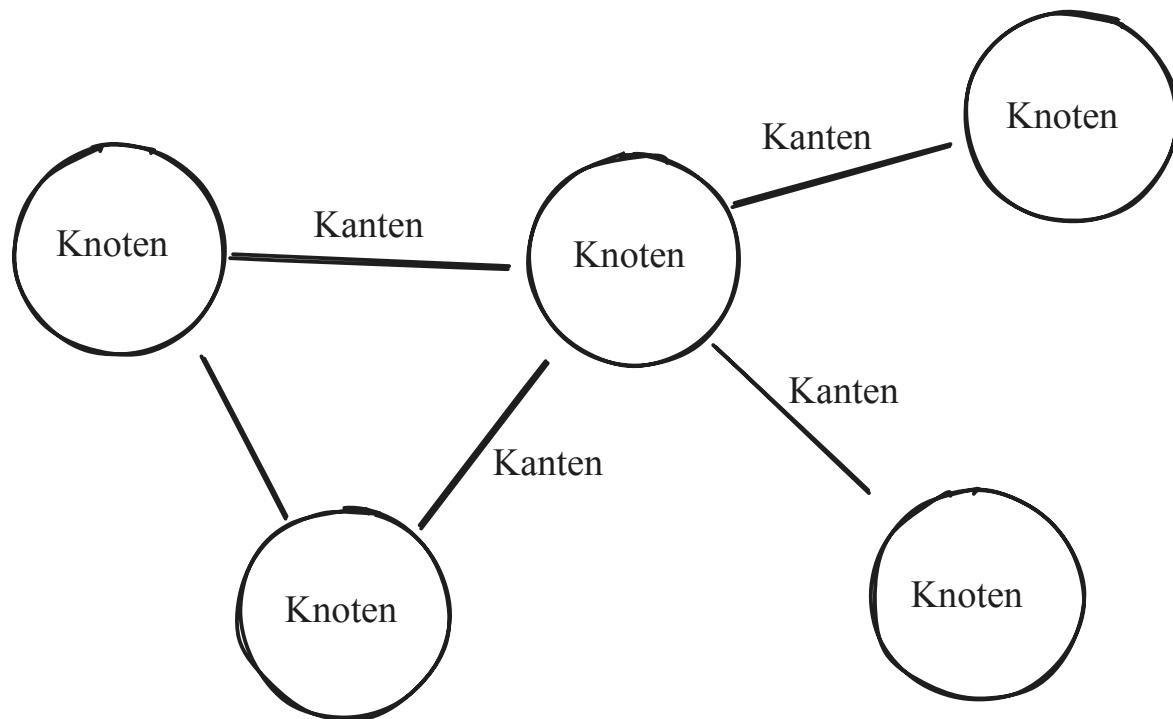




# Konkrete Datenstrukturen: Array



# Konkrete Datenstrukturen: Graph

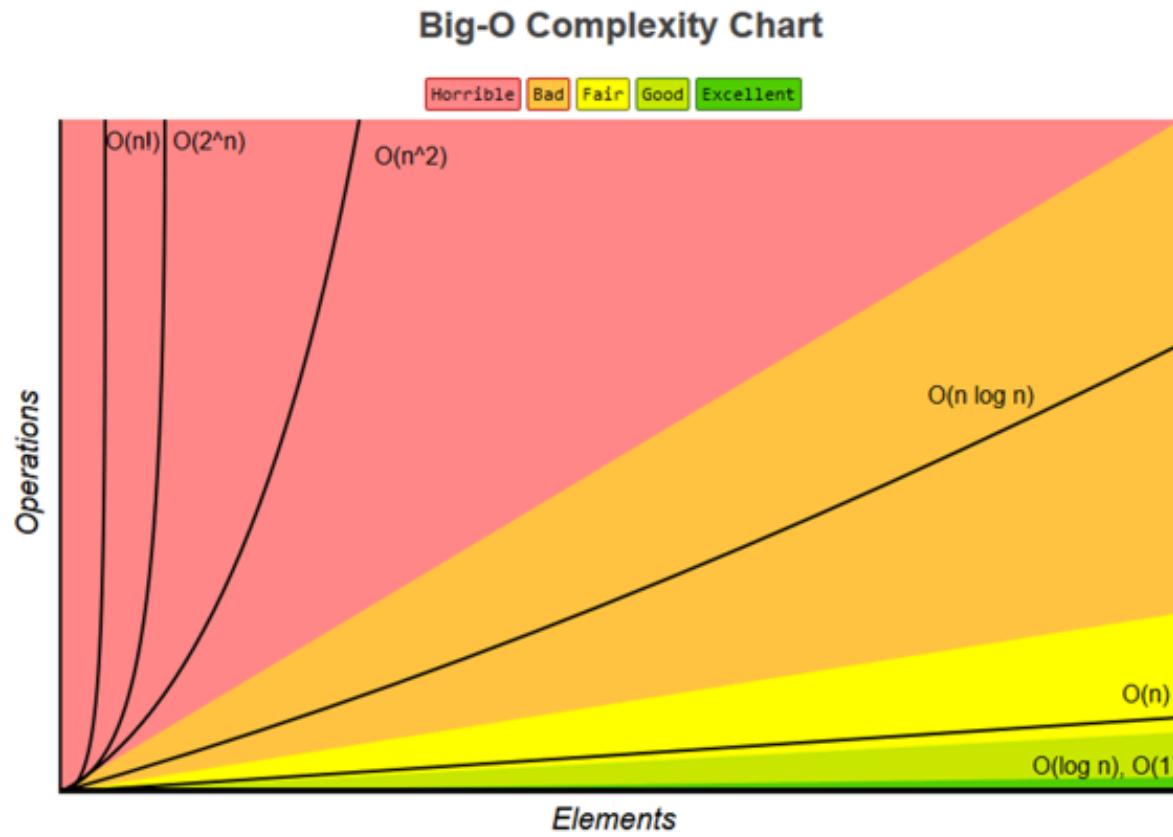


# Komplexität von Algorithmen

# Speicher und Rechenaufwand von Datenstrukturen

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

# Objektorientierte Prinzipien

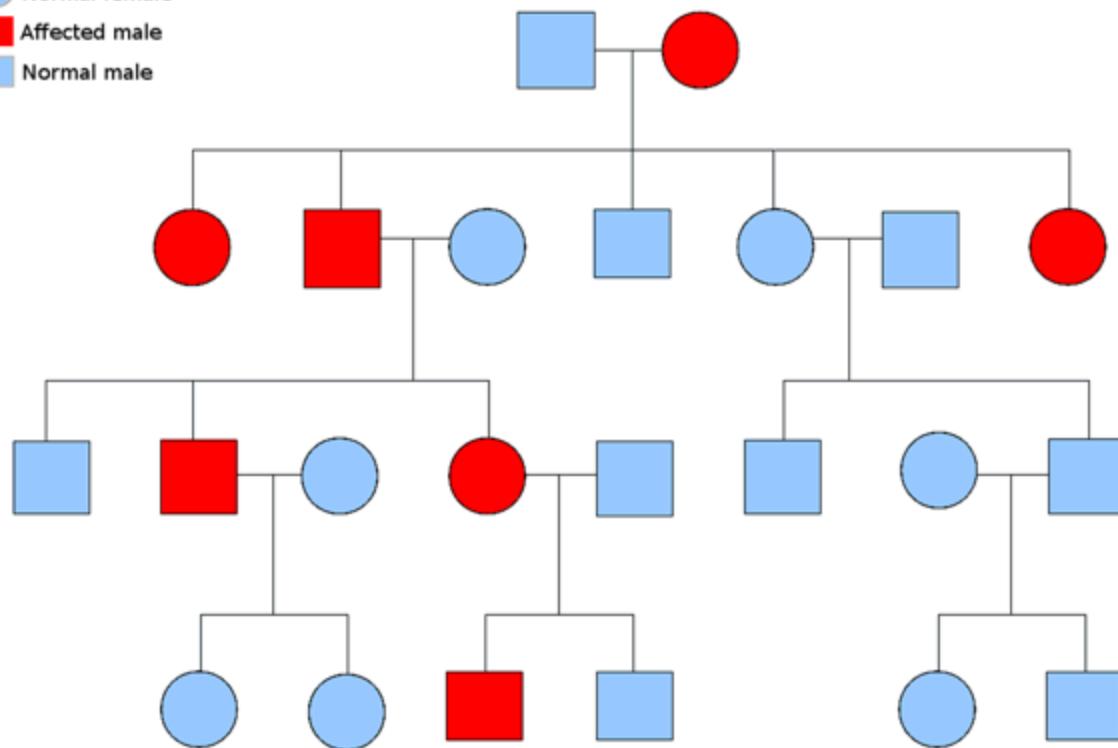
I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning – it took a while to see how to do messaging in a programming language efficiently enough to be useful).

OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.

beide Alan Kay, [http://userpage.fu-berlin.de/~ram/pub/pub\\_jf47ht81Ht/doc\\_kay\\_oop\\_en](http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en)

# 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)

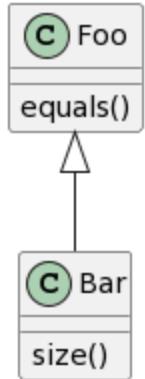
Eine Klasse ist ein Datentyp:

- 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

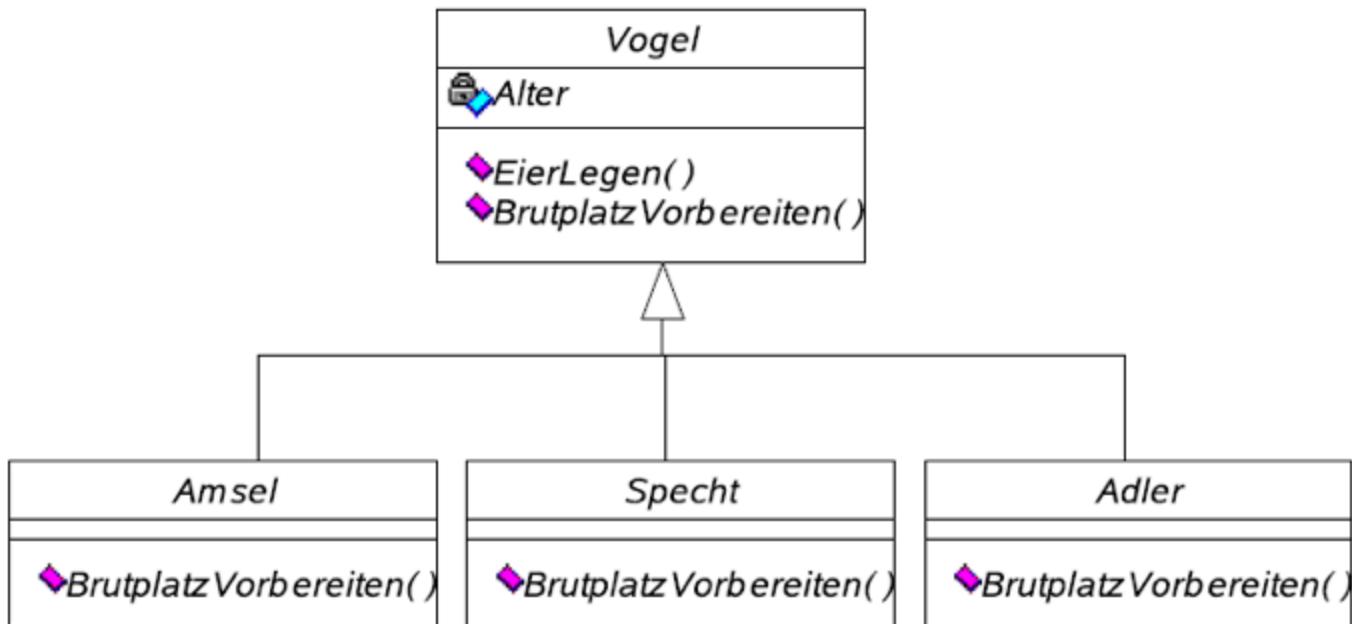
Bar erbt von Foo

- 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

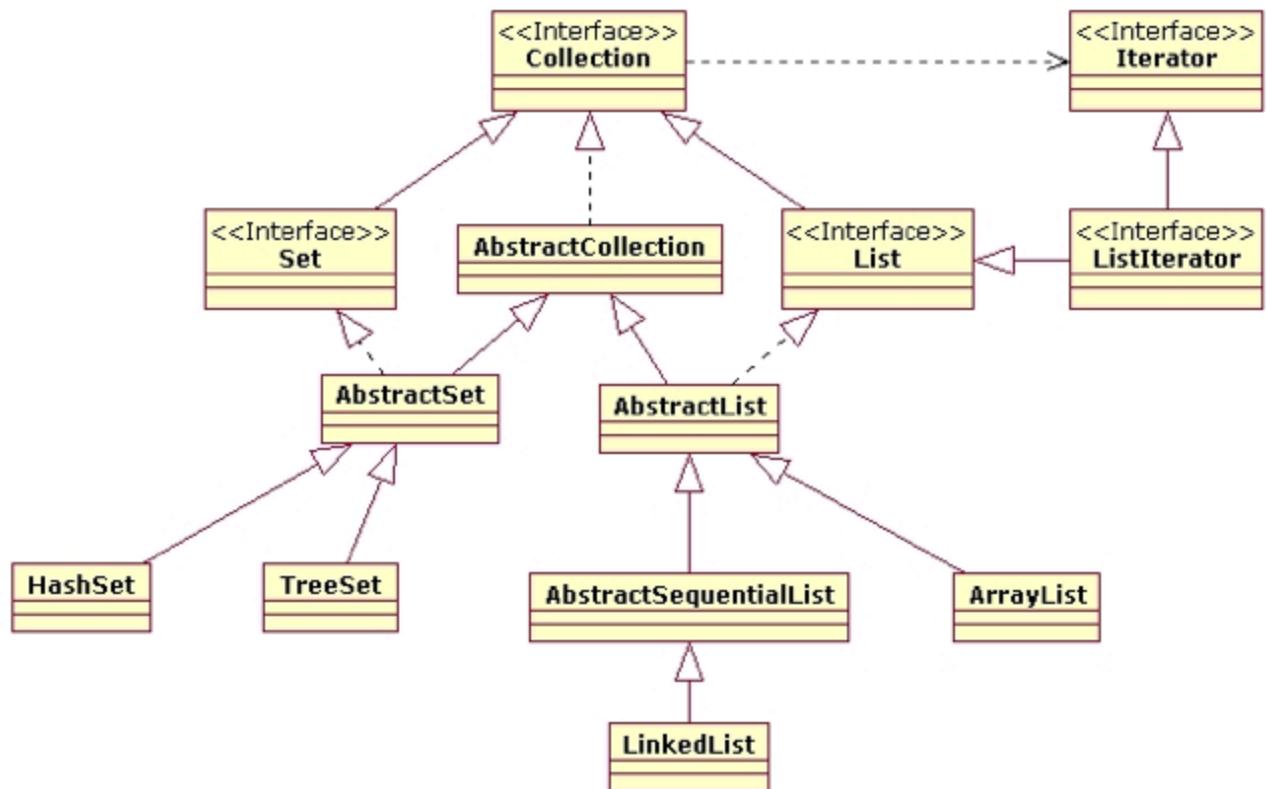
# Vererbung in Python

```
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

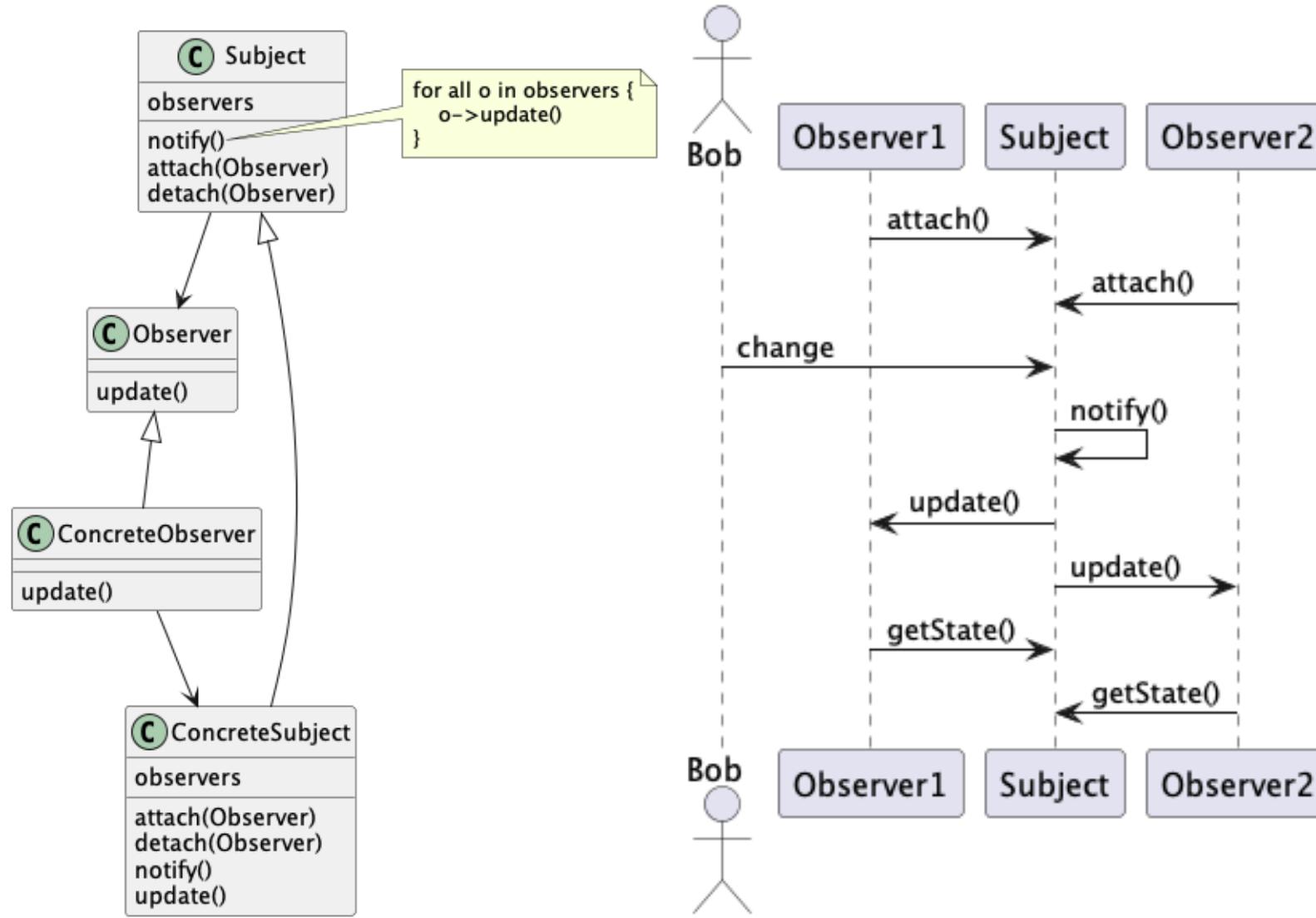
Gang of Four:

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

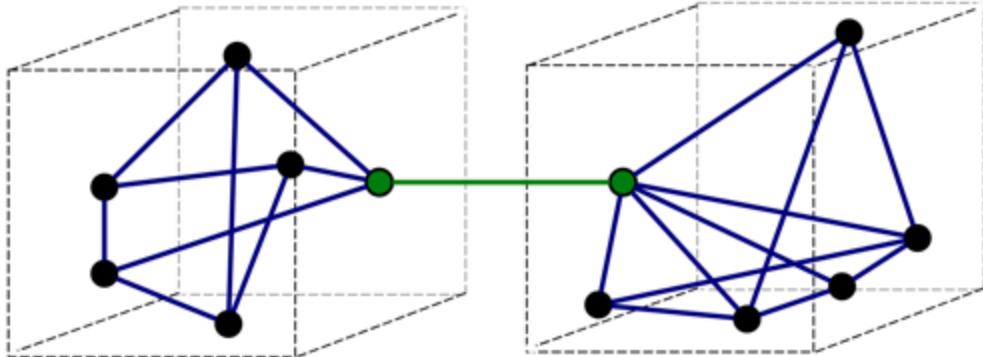
Hohpe, Gregor; Woolf, Bobby (2003): Enterprise Integration Patterns: Designing, Building,  
and Deploying Messaging  
Solutions, Addison-Wesley

# Observer

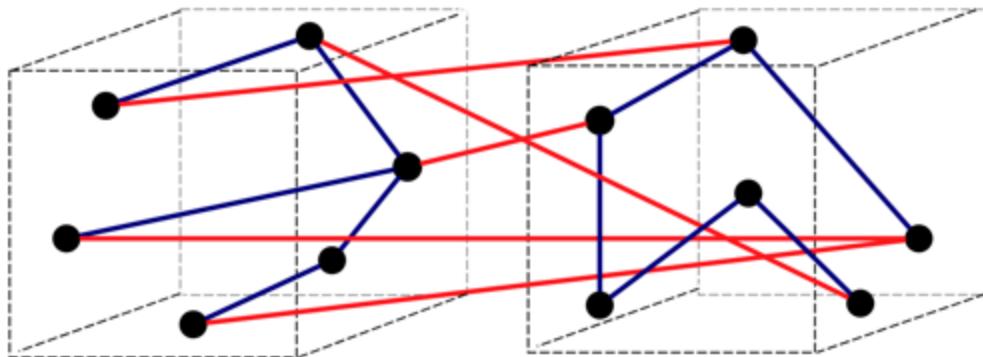


# Architekturen

# High Cohesion - Low Coupling

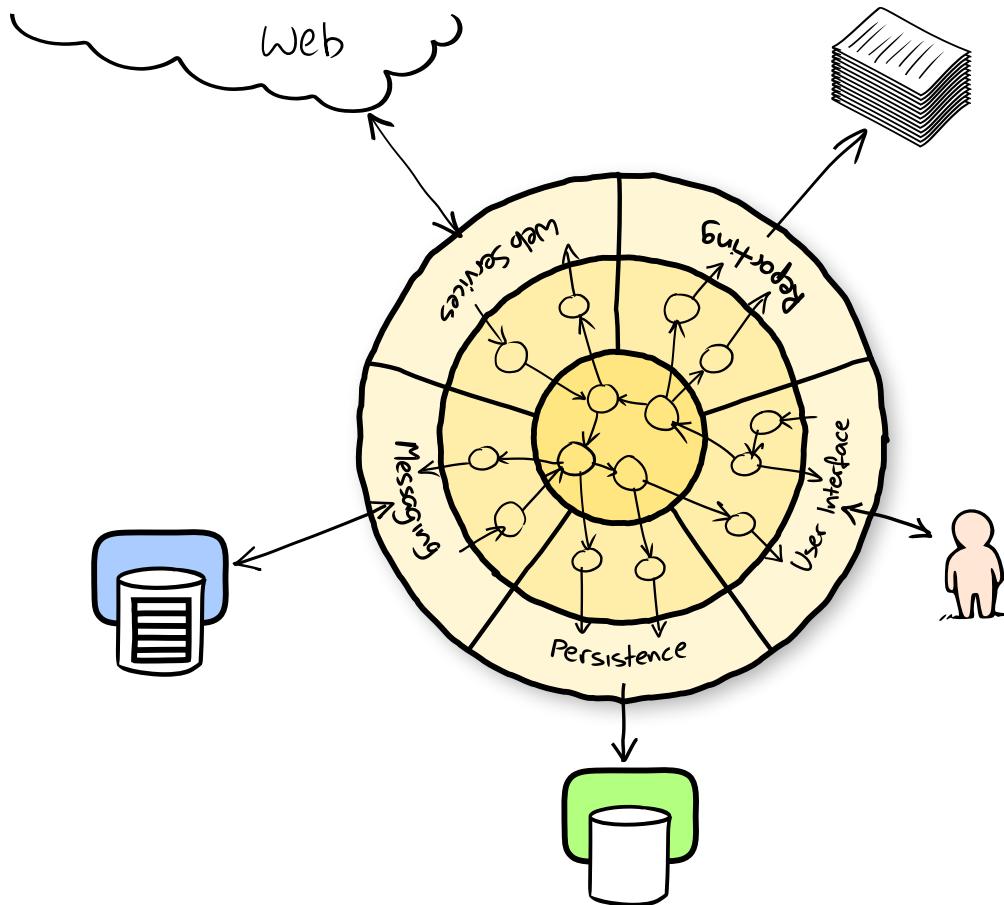


a) Good (loose coupling, high cohesion)

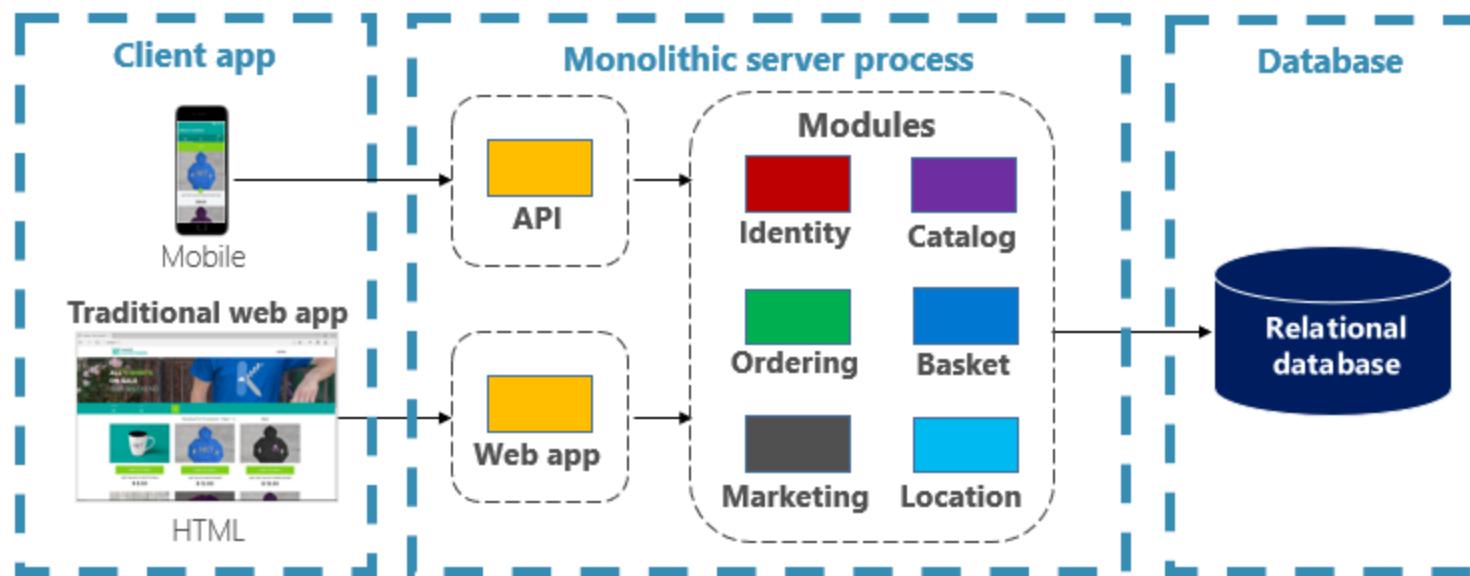


b) Bad (high coupling, low cohesion)

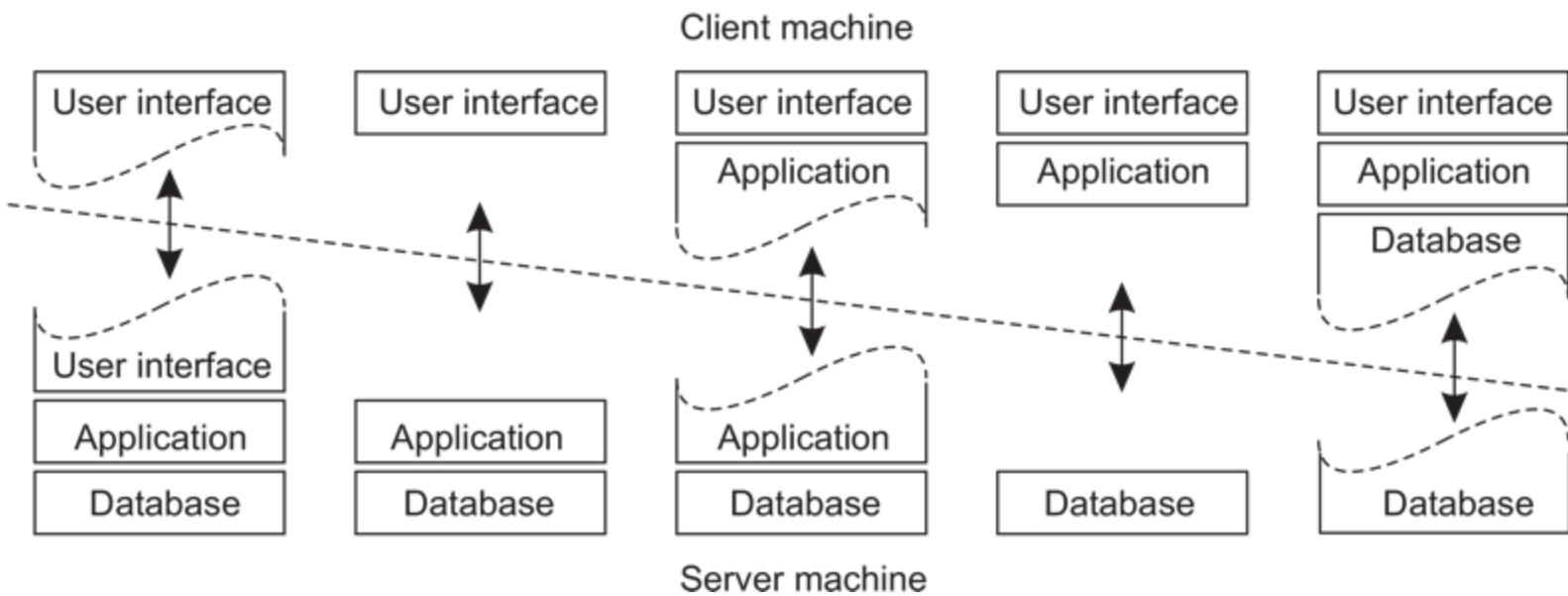
# Ports and Adapters



# Traditional Monolithic Design



# Schichtenarchitektur im Client Server Modell

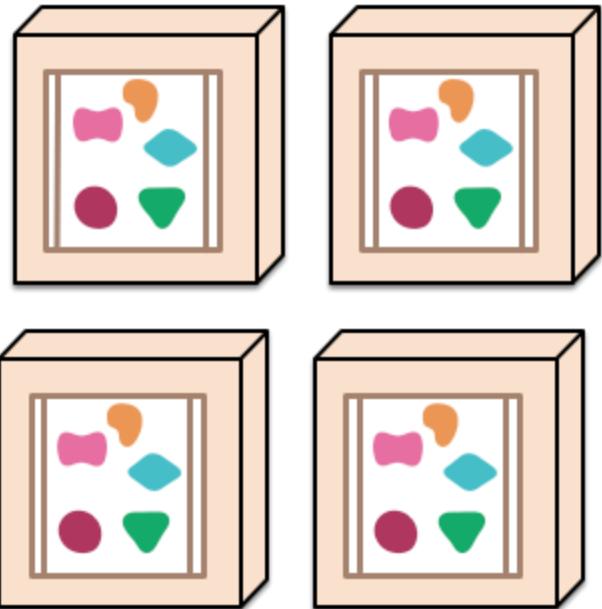


# Microservices

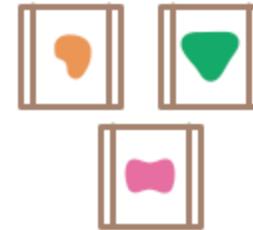
*A monolithic application puts all its functionality into a single process...*



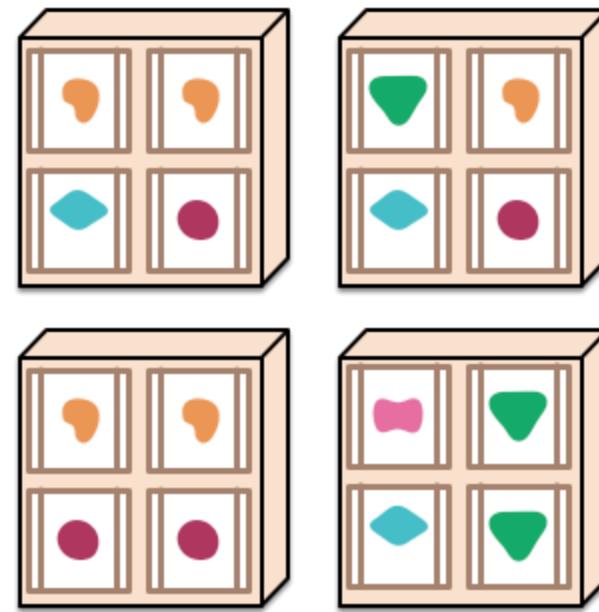
*... and scales by replicating the monolith on multiple servers*



*A microservices architecture puts each element of functionality into a separate service...*



*... and scales by distributing these services across servers, replicating as needed.*



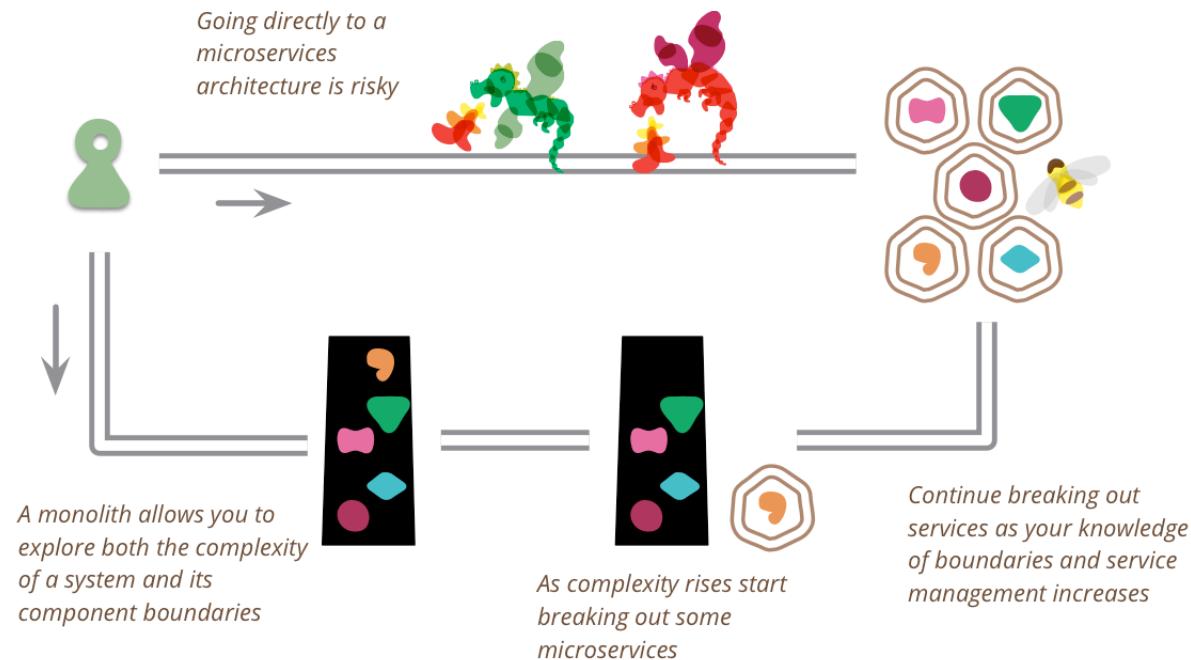
# Microservices

- Maximale Skalierbarkeit
- Einzelne Services können von **kleinen[^1]** Teams **unabhängig entwickelt und deployed** werden
- Bessere Wart- und Erweiterbarkeit
- Unterschiedliche Technologien können eingesetzt werden
- Kommunikation nicht trivial
- Höhere Wahrscheinlichkeit eines Ausfalls
- **Hohe Komplexität**

[^1]: "We try to create teams that are no larger than can be fed by two pizzas"

# Monolith First

- Vorsicht vor **Cargo-Kult**: Amazon, Google, Meta etc. haben heute andere Herausforderungen als Startups
- Technologien oder Architekturen wählen, "weil Google macht das auch so" ist ein schlechter Grund



[martinfowler.com/bliki/MonolithFirst.html](http://martinfowler.com/bliki/MonolithFirst.html)

# 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
}
```