

# Einstieg

# **Softwaredebakel**

## **VBS**

- Schweizer Armee ohne krisensichere Logistik bis 2035
- Armee-Debakel: 300-Millionen-Projekt seit Monaten suspendiert

## **Kantonsverwaltung**

- Wegen fehlerhafter Software braucht es mehr Haftplätze

## **Polizei**

- Berner Polizisten beklagen sich über die neue IT

## **Crowdstrike**

- Der Tag, an dem die IT weltweit verrückt spielte – ein Überblick

# Kundenorientierung

**Software soll den Kunden Mehrwert bringen**

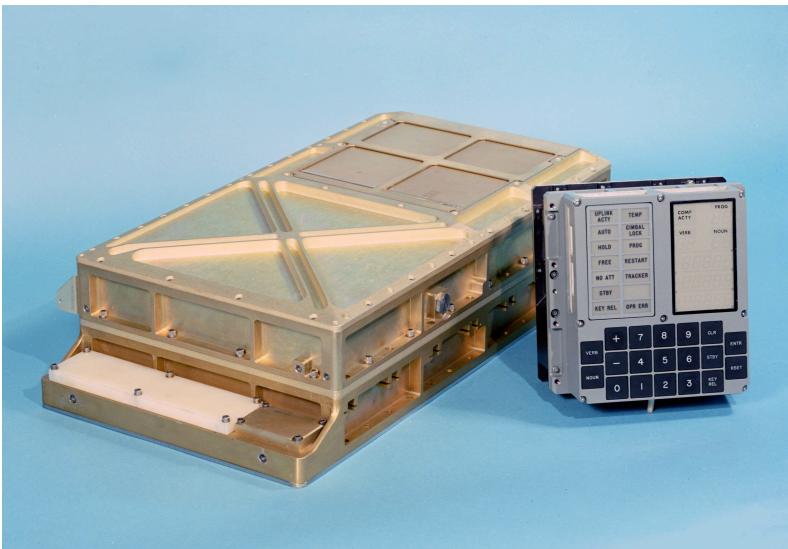
- Software soll stabil laufen
- Neue Features sollten schnell umgesetzt und nutzbar sein
- Softwaresysteme werden immer komplexer

# **Teamarbeit**

**Mehrere Personen arbeiten am selben Softwareprojekt**

- Versionsverwaltung wird verwendet (Git, SVN)
- Konflikte entstehen und sind aufwendig

# Ab 1961: Margaret Hamilton, Apollo Guidance Computer



## **2001: 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/>

# **Software Engineering / Software Architecture**

Software engineering is the application of an empirical, scientific approach to finding efficient, economic solutions to practical problems in software

(Farley, 2022, S.4)

The goal of software architecture is to minimize the human resources required to build and maintain the required system

(Martin, 2018)

Übergang zwischen Software Entwicklung und Software Architektur ist fliessend

# Learning

- Iteratives und inkrementelles Arbeiten
- Feedback
- Empirisches und experimentelles Arbeiten

(vgl. Farley, 2022, S.4)

# Managing Complexity

- Modularity & Separation of Concerns
- Cohesion & Coupling
- Abstraction

(vgl. Farley, 2022, S.5)

## Production Is Not Our Problem

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

# Space X Starship

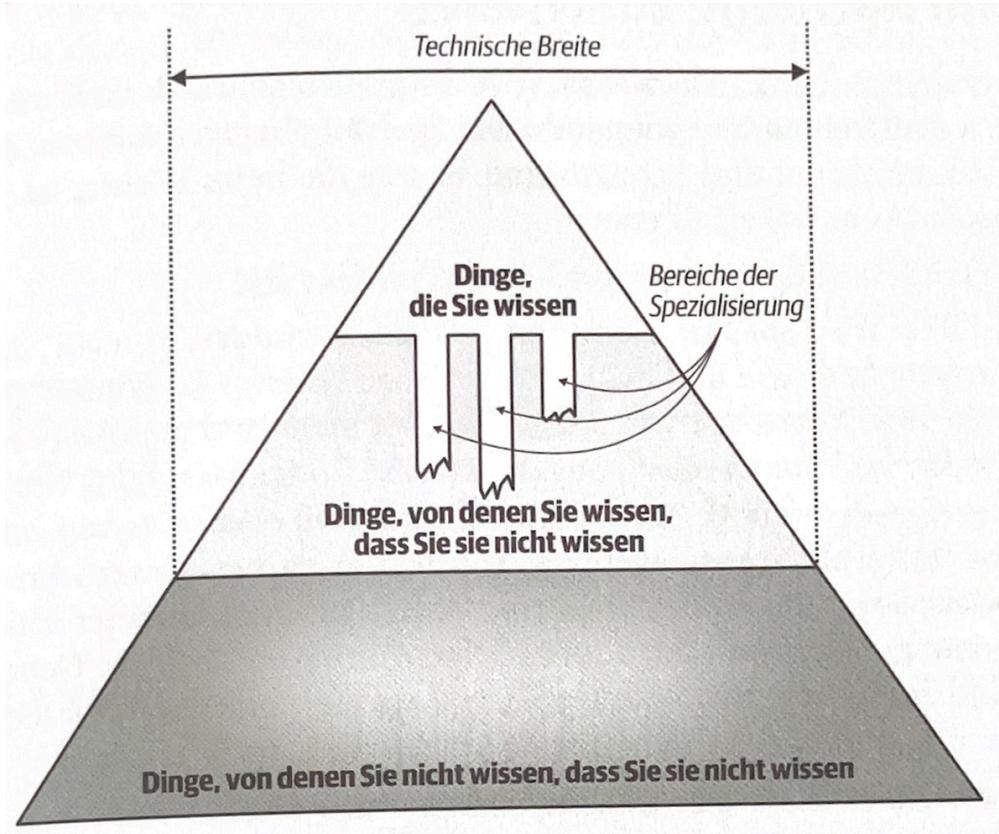
[How Not to Land an Orbital Rocket Booser, 2017](#)

[WOW! Watch SpaceX Catch A Starship Booster In Air, 2024](#)

Finanzierung: **ca 3 Mrd. Dollar**

Apollo-Programm: 1958 bis 1969, inflationsbereinigt: **163 Mrd. Dollar** (ohne Mercury und Gemini)

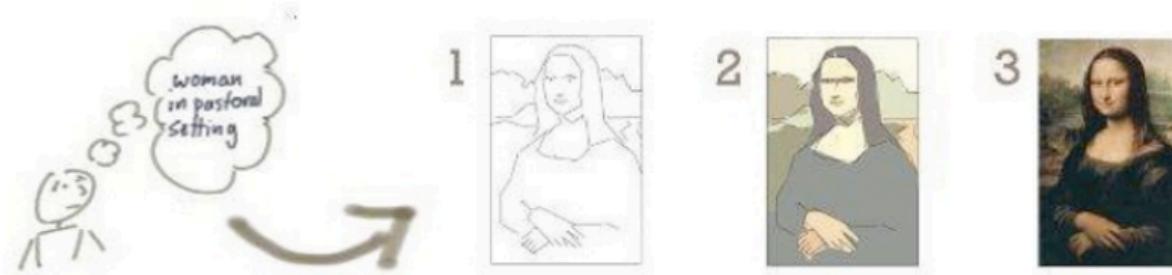
# Lernen



(Richards, 2021, S.29)

# Iteratives und inkrementelles Arbeiten

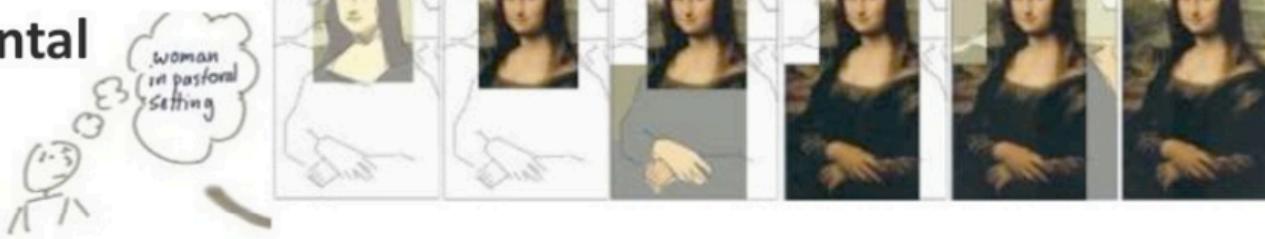
Iterative



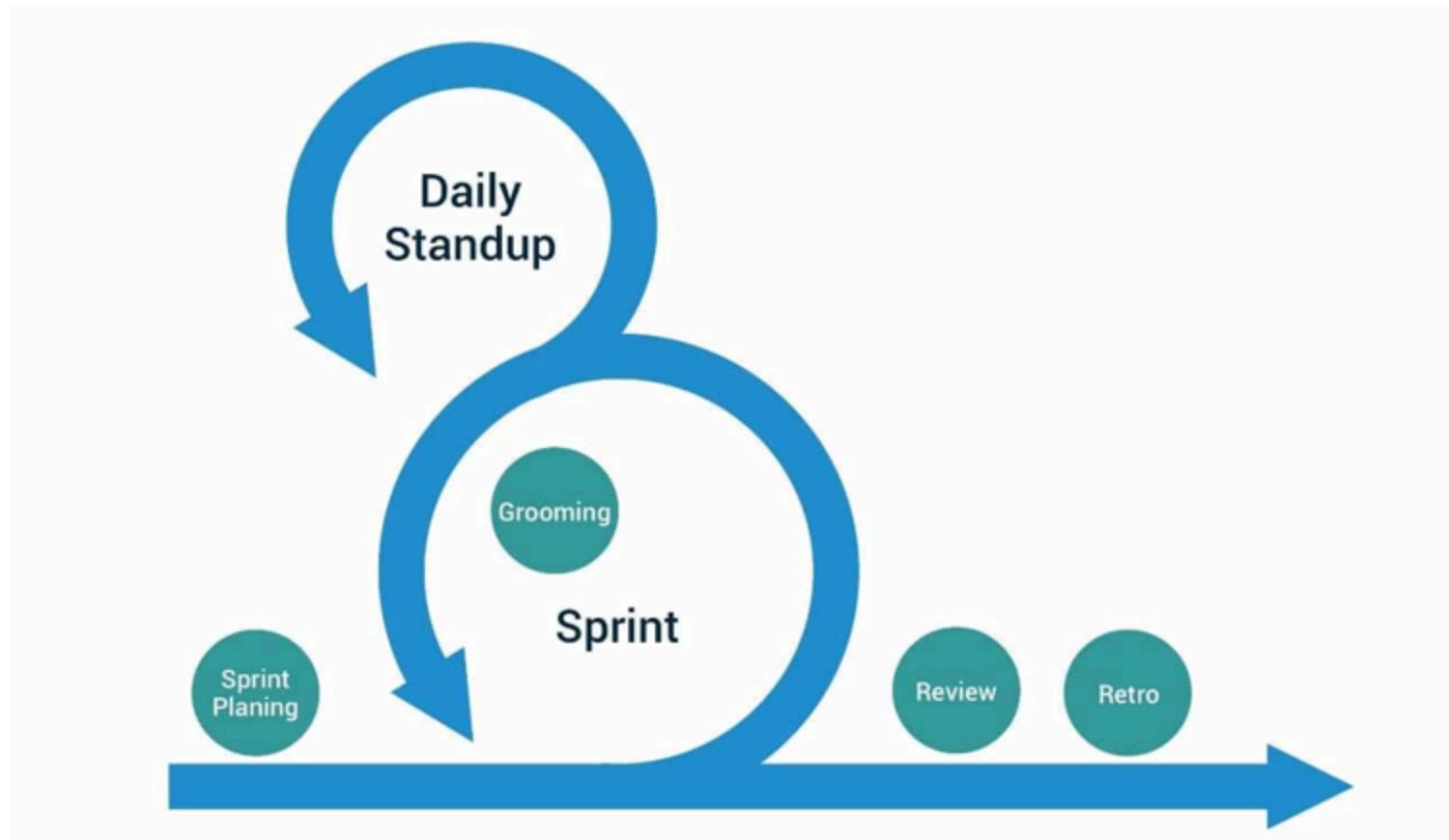
Incremental



Iterative &  
Incremental



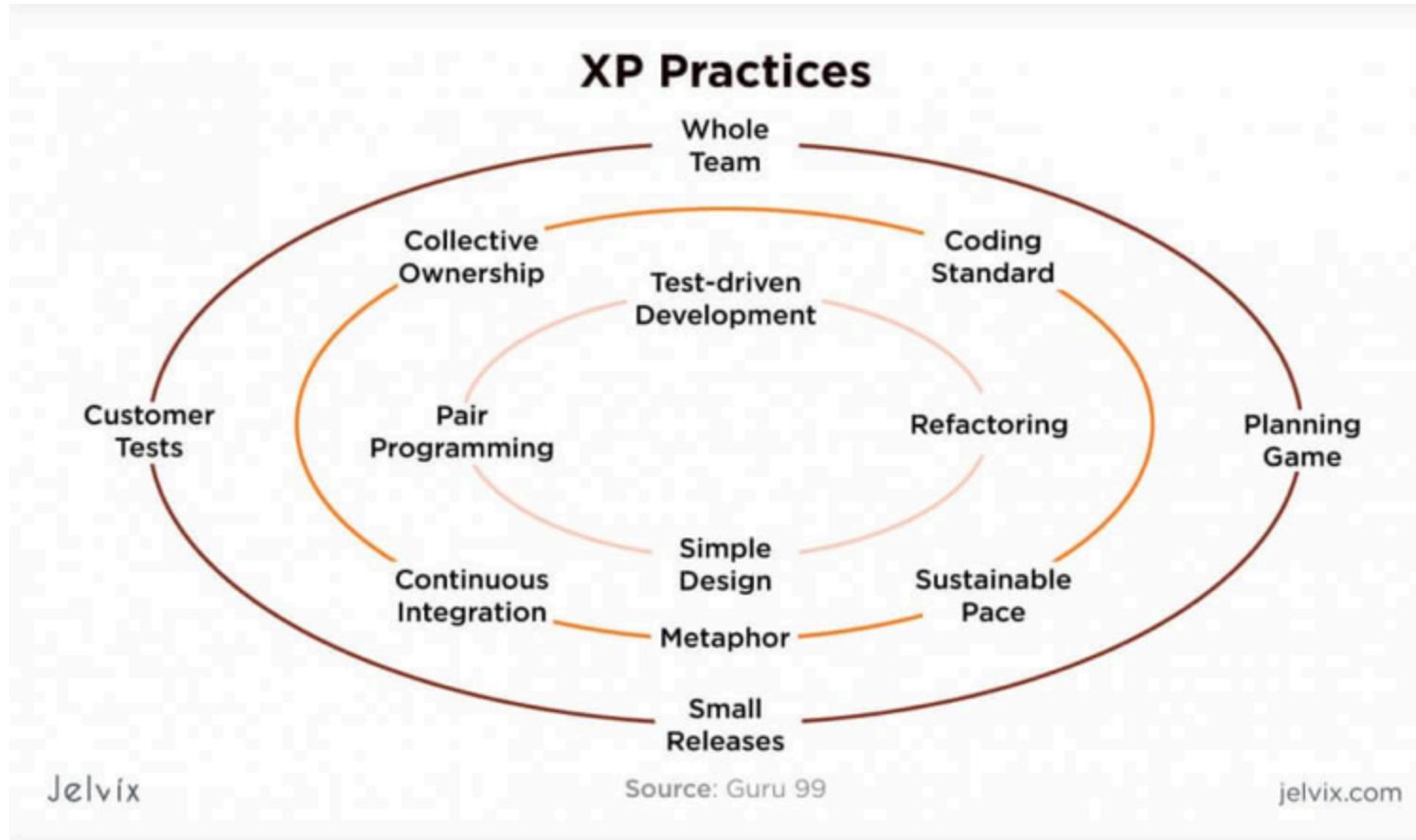
# Iterationen



# Embrace Change

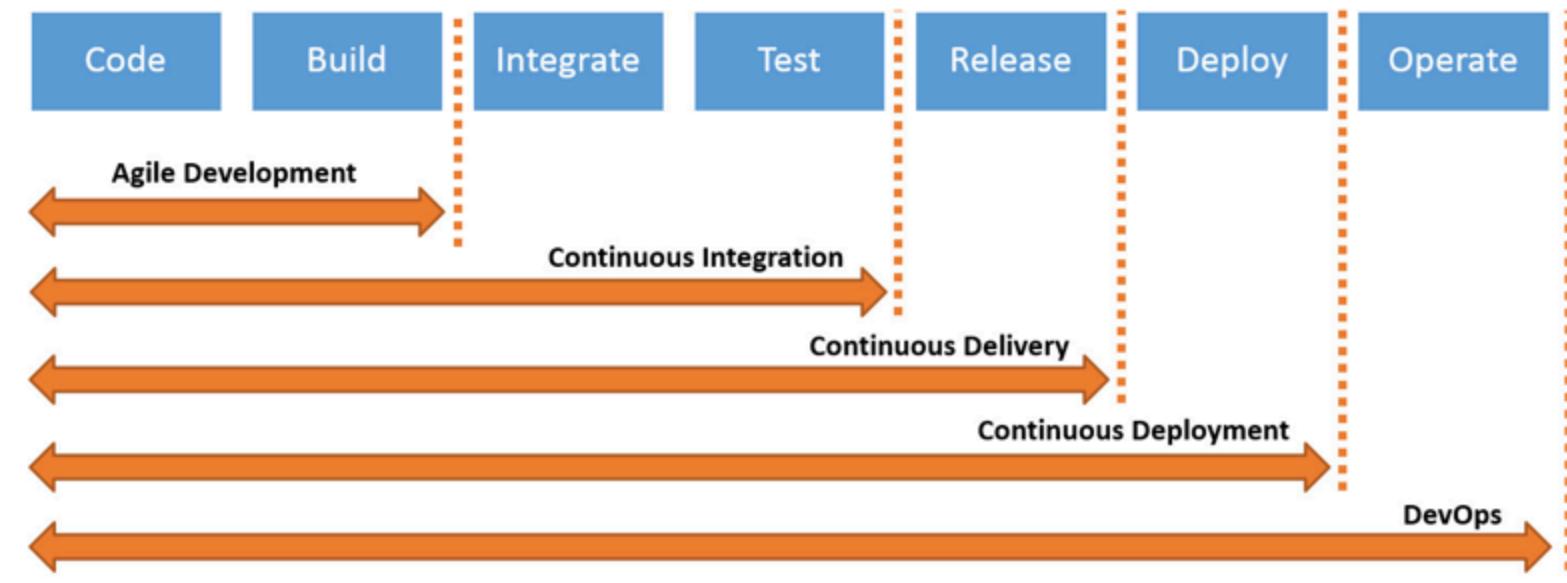


# Extreme Programming



# **Feedback**

# 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 vereinfacht**
- Eine sehr hohe, **automatische Testabdeckung** ist zwingend

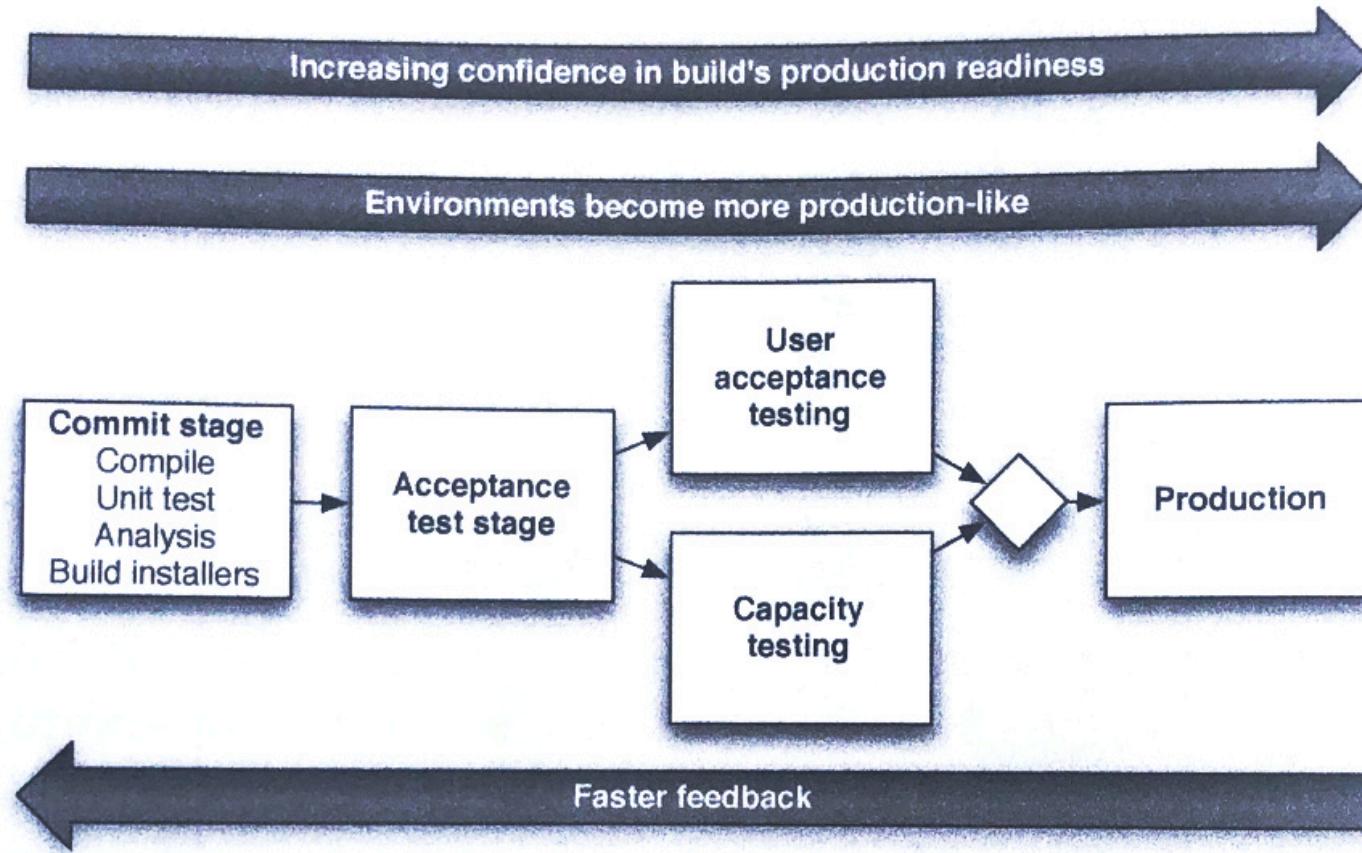
## Continuous Deployment

- 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, Better products**
- Kaum mehr Release-Stress, **Happier teams**

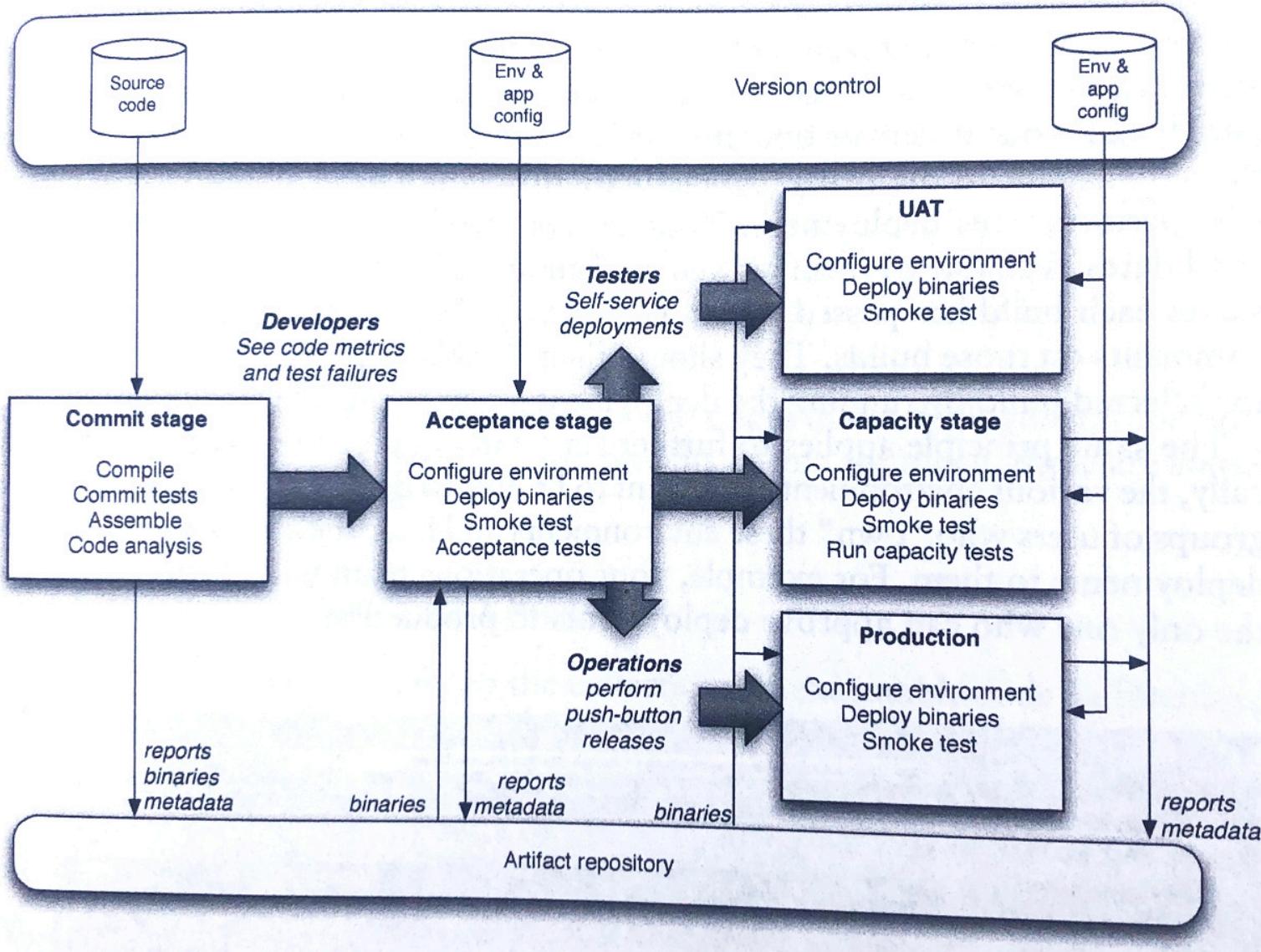
<https://www连续部署.com/>

Modern Software Engineering

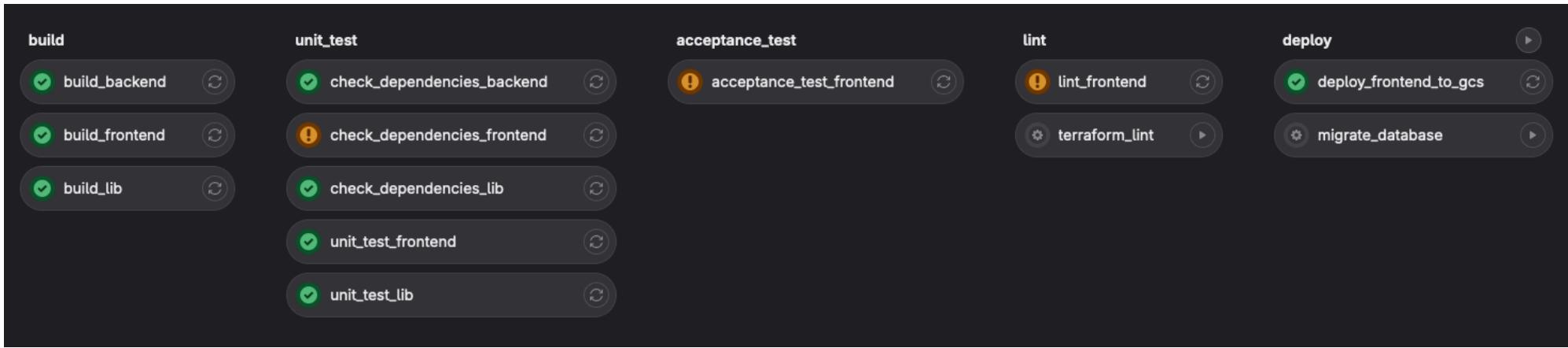
# Deployment Pipelines



**Figure 5.3** *Trade-offs in the deployment pipeline*



(Jez Humble, David Farley (2010): Continuous Delivery)



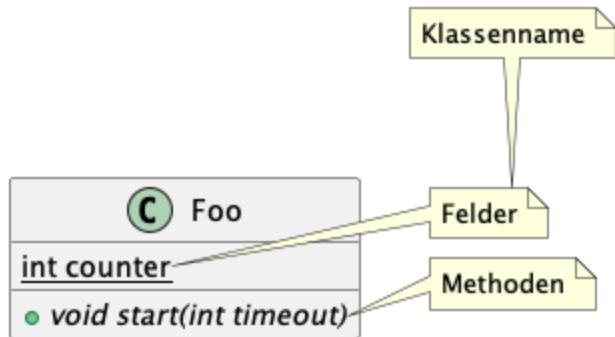
- [Youtube: Continuous Delivery - Deployment Pipelines](#)
- Jez Humble, David Farley (2010): Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley Signature Series (Fowler)

# **Empirisches und experimentelles Arbeiten**

# Kommunikation

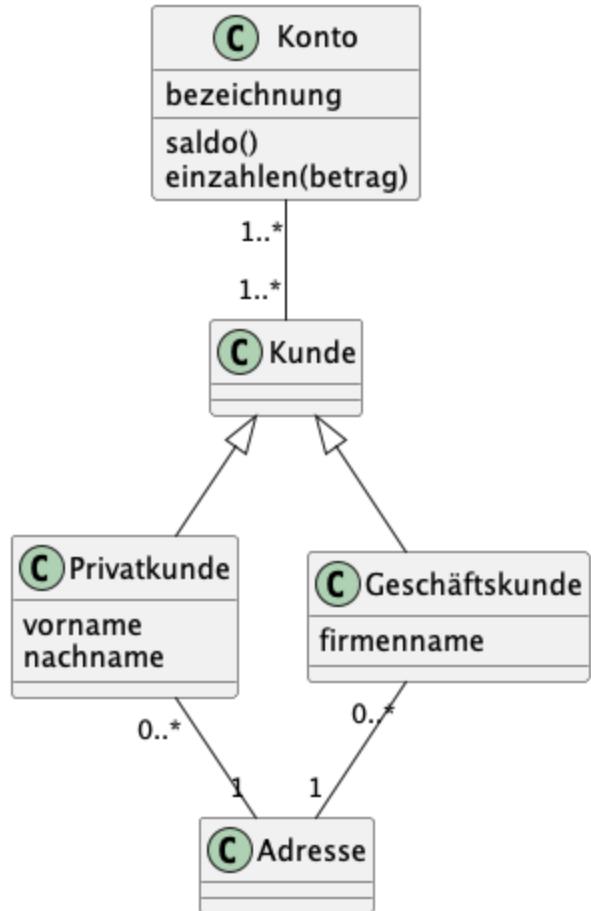
# **Domain Driven Design**

# UML Klassendiagramm



PlantUML

# UML Klassendiagramm



# PlantUML

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

class Kunde {}

class Privatkunde {
    vorname
    nachname
}

class Geschäftskunde {
    firmenname
}

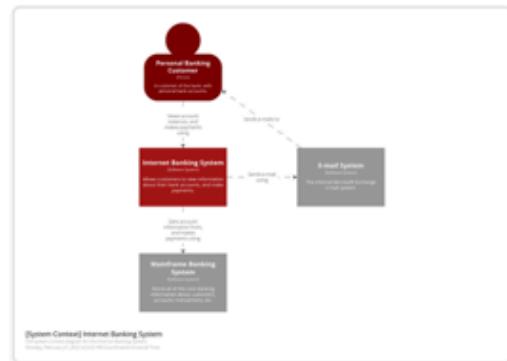
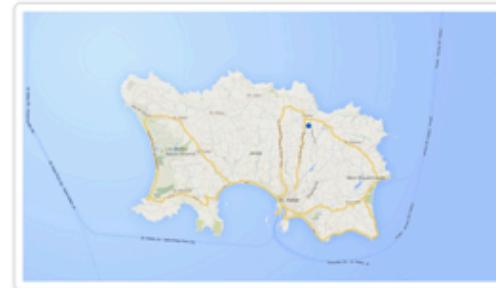
class Adresse {}

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

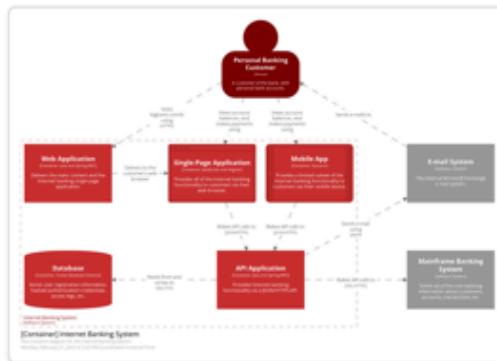
Privatkunde "0..*" -- "1" Adresse
Geschäftskunde "0..*" -- "1" Adresse

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

# C4 Model



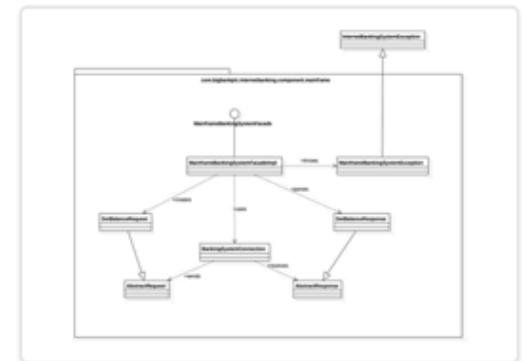
Level 1: A **System Context** diagram provides a starting point, showing how the software system in scope fits into the world around it.



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.

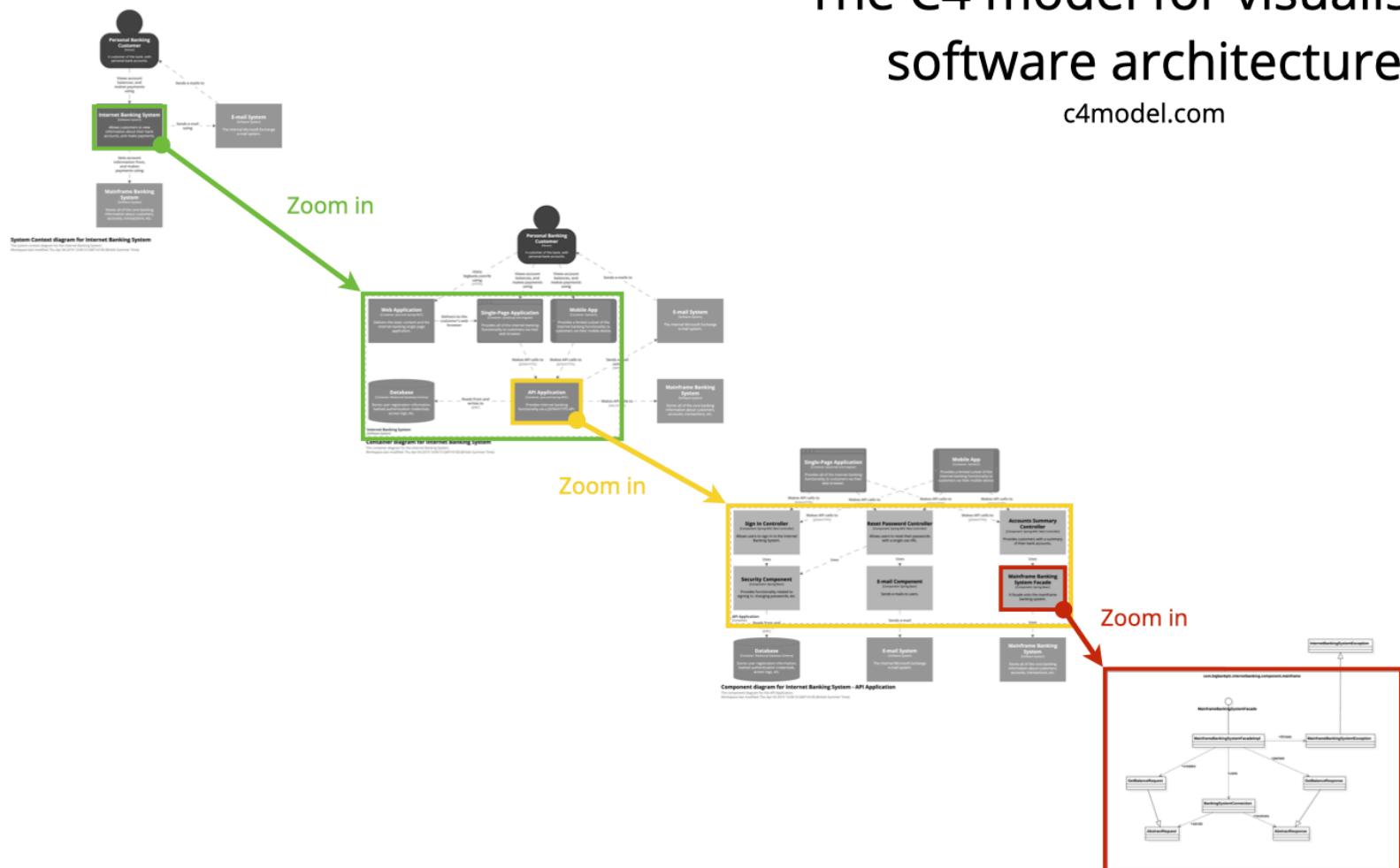


Level 4: A **code** (e.g. UML class) diagram can be used to zoom into an individual component, showing how that component is implemented.

<https://c4model.com/>

# The C4 model for visualising software architecture

c4model.com

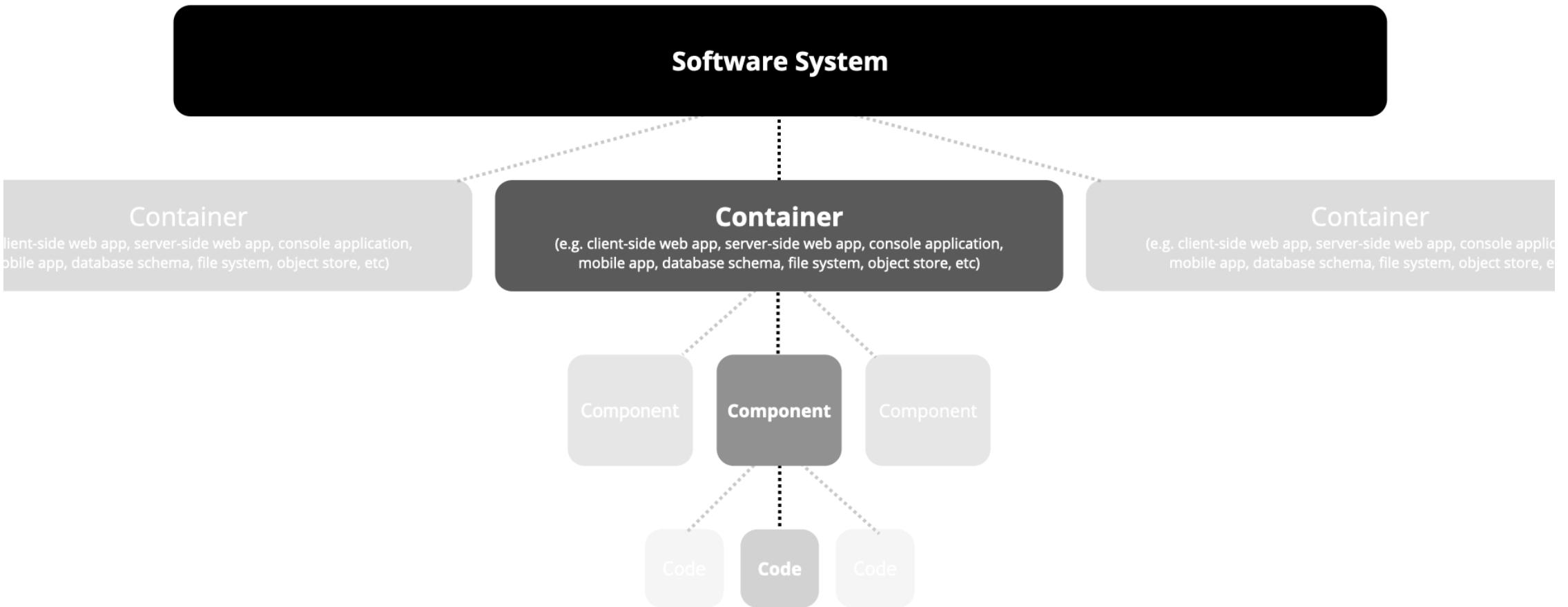


Level 1  
Context

Level 2  
Containers

Level 3  
Components

Level 4  
Code

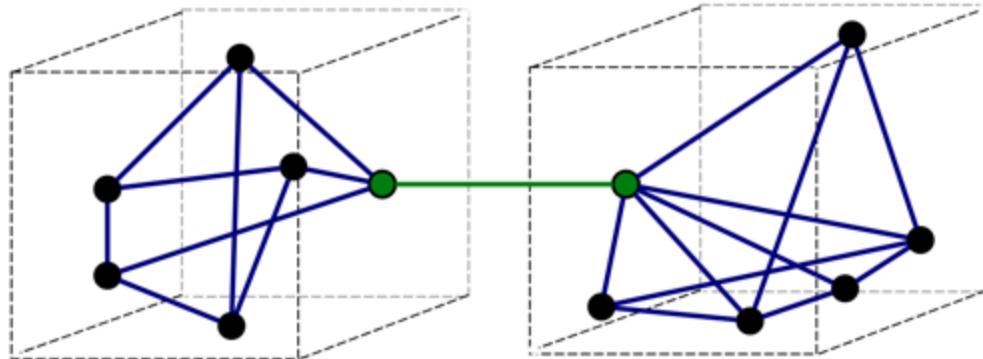


A **software system** is made up of one or more **containers** (applications and data stores), each of which contains one or more **components**, which in turn are implemented by one or more **code** elements (classes, interfaces, objects, functions, etc).

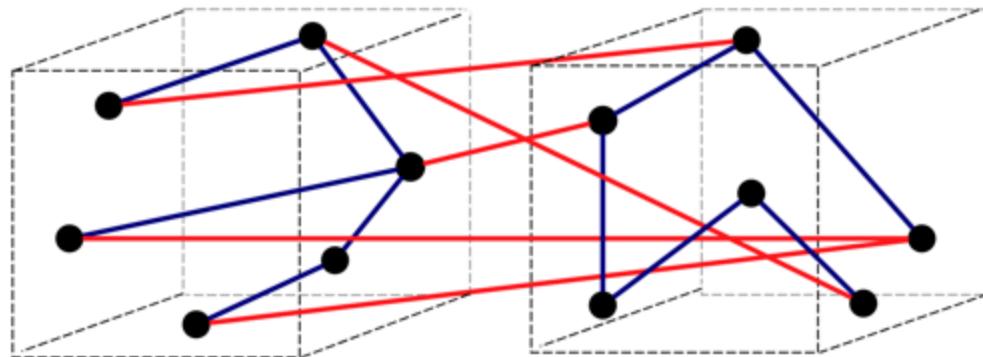
# Komplexität

# **Modularity & Separation of Concerns**

# Cohesion & Coupling



a) Good (loose coupling, high cohesion)

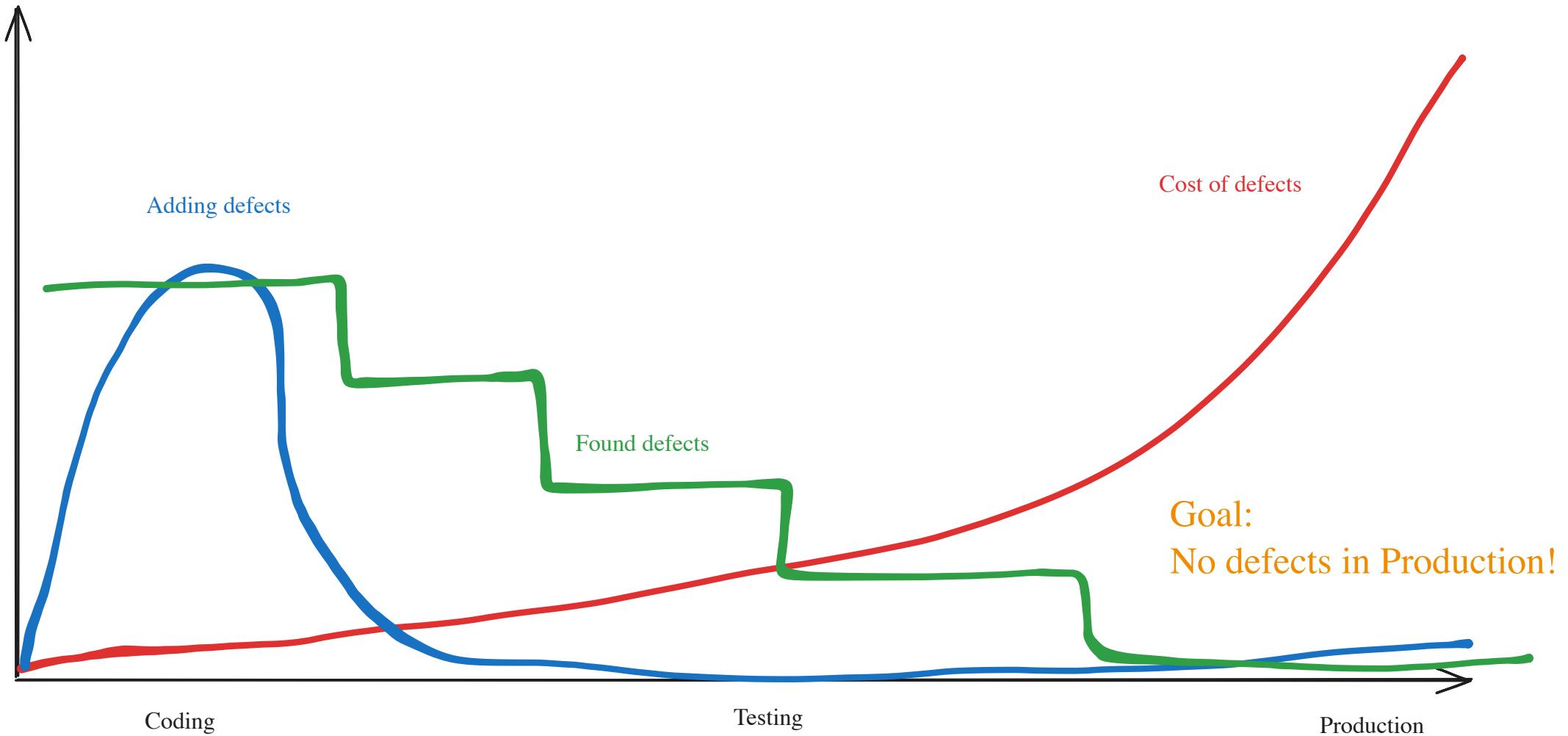


b) Bad (high coupling, low cohesion)

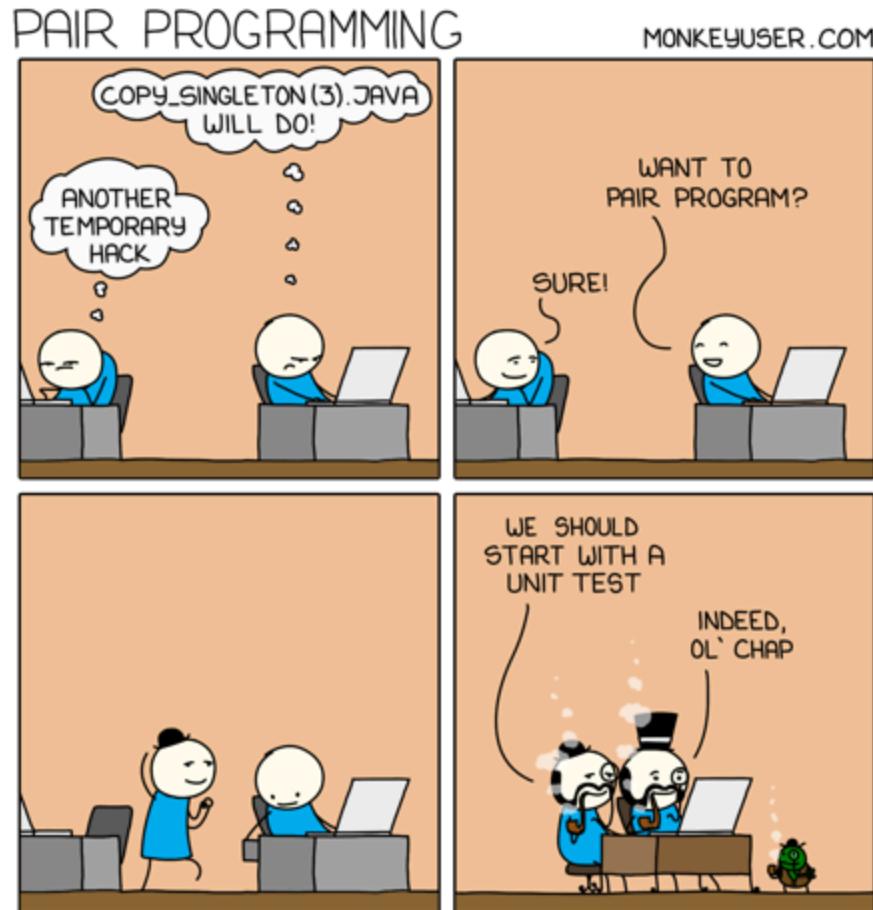
# Abstraction

# 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

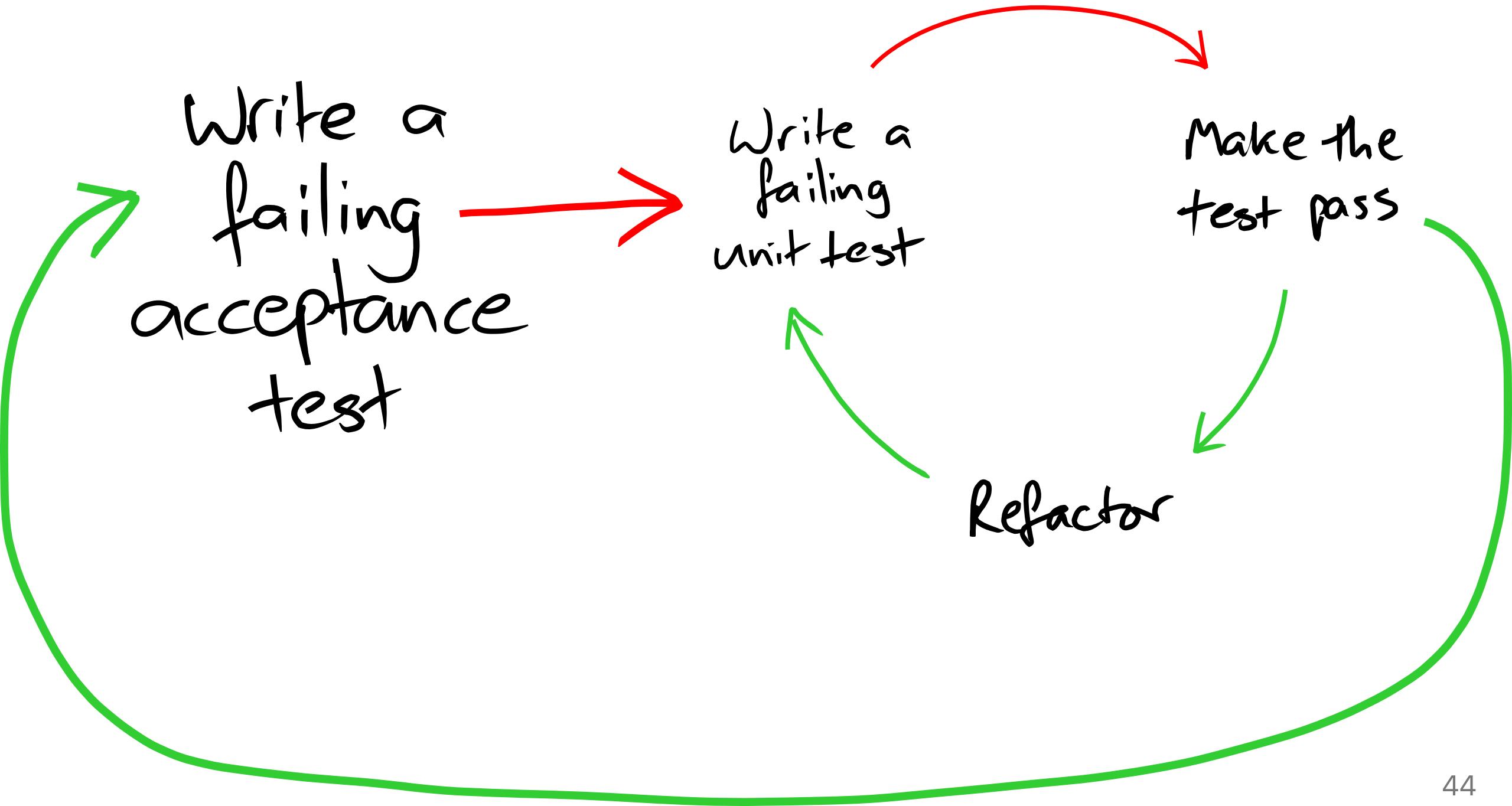
Hard to write a test?

Write a failing acceptance test

Write a failing unit test

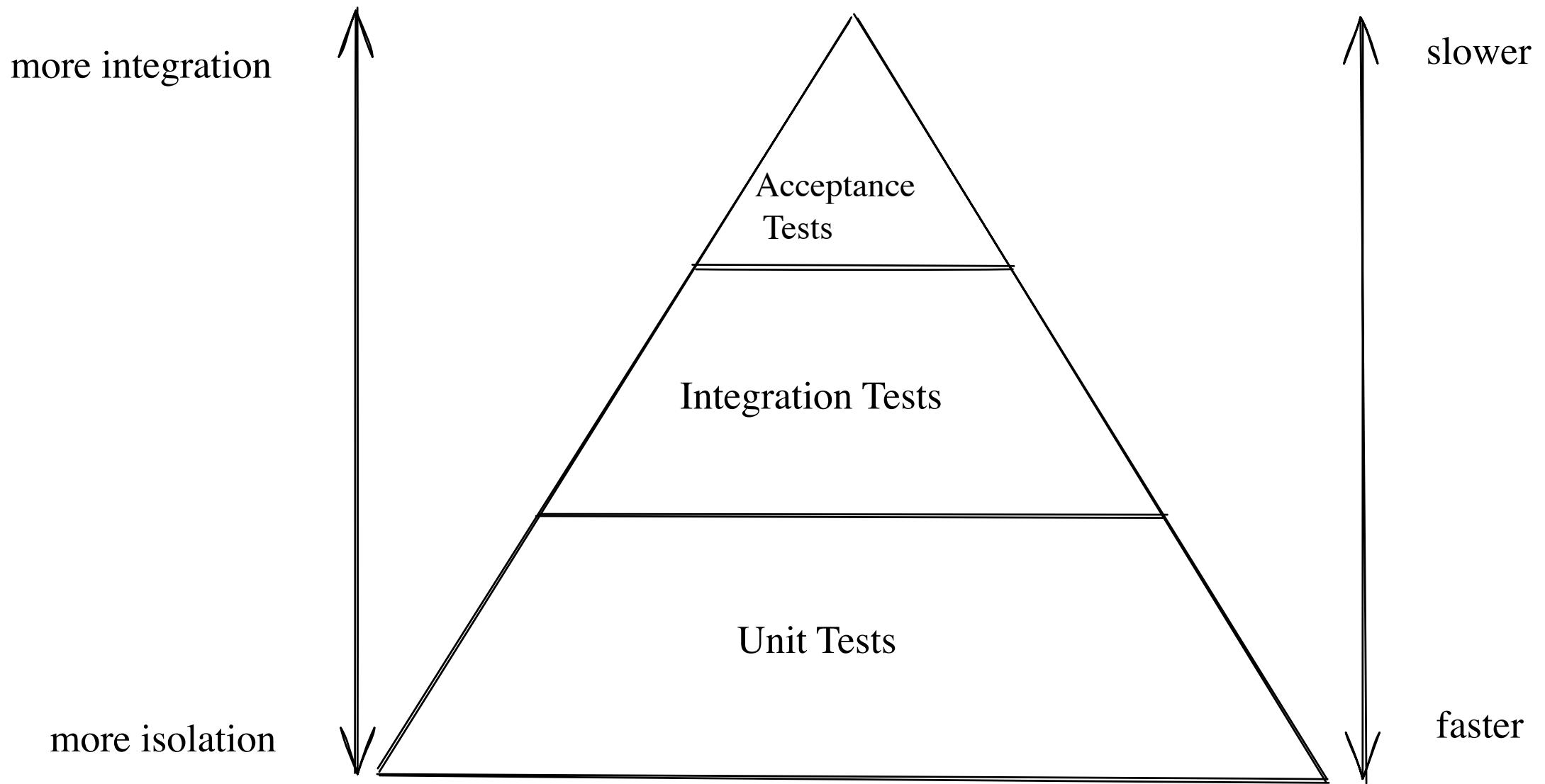
Make the test pass

Refactor



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

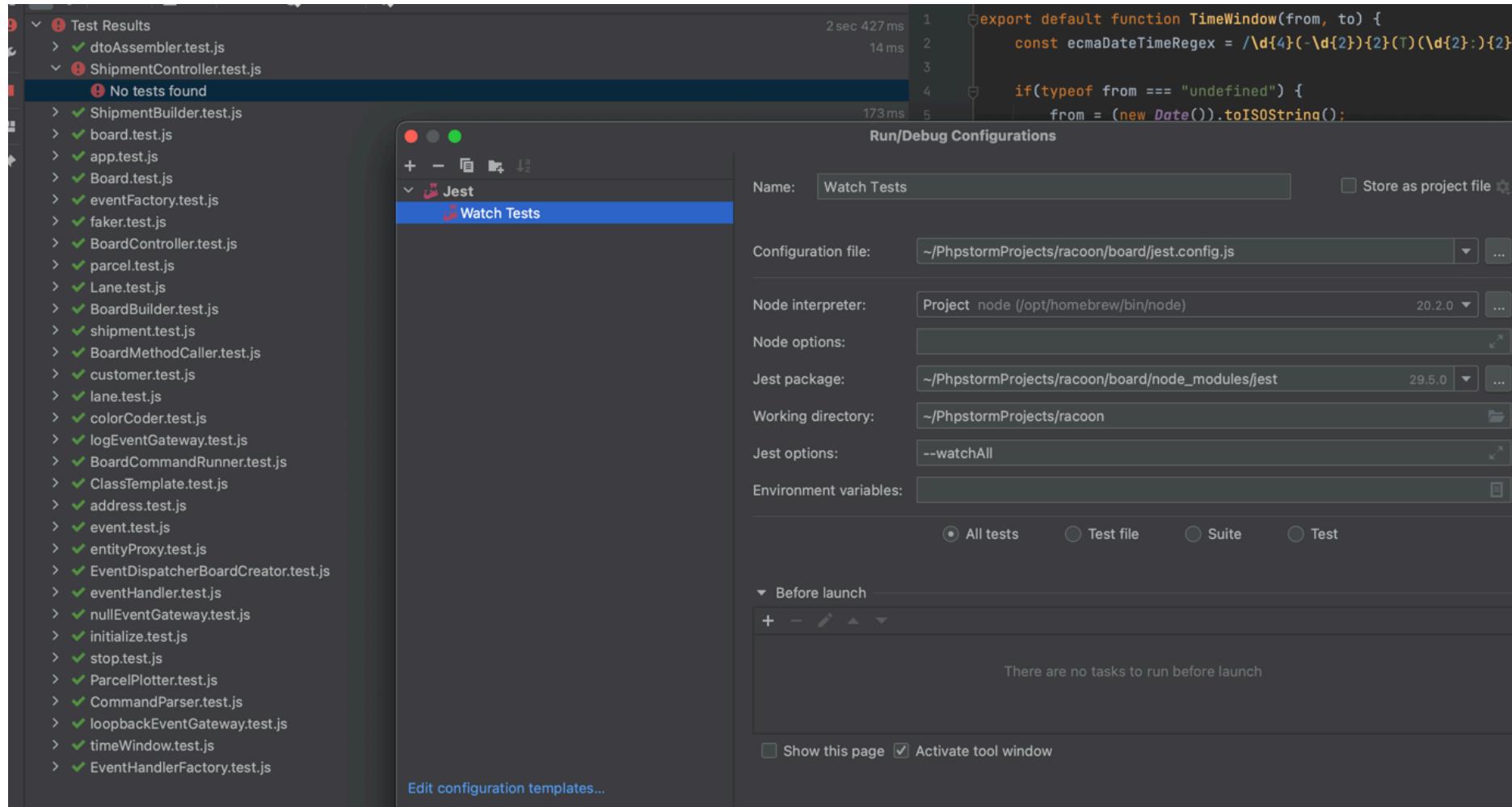
## Testpyramide



## Testing: AAA

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

# 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?

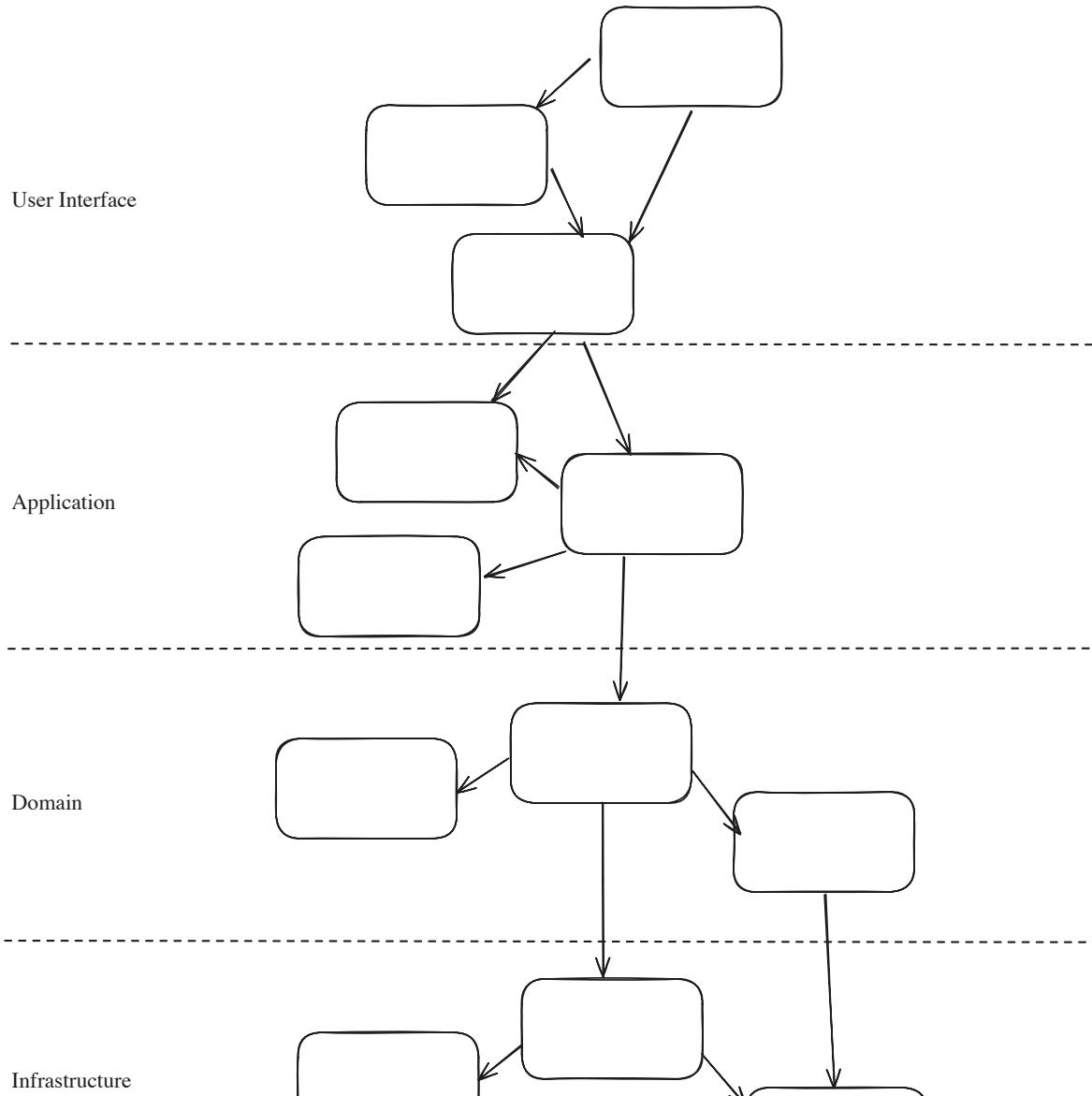
- Refactoring Improves the Design of Software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You Find Bugs
- Refactoring Helps You Program Faster

## When Should You Refactor?

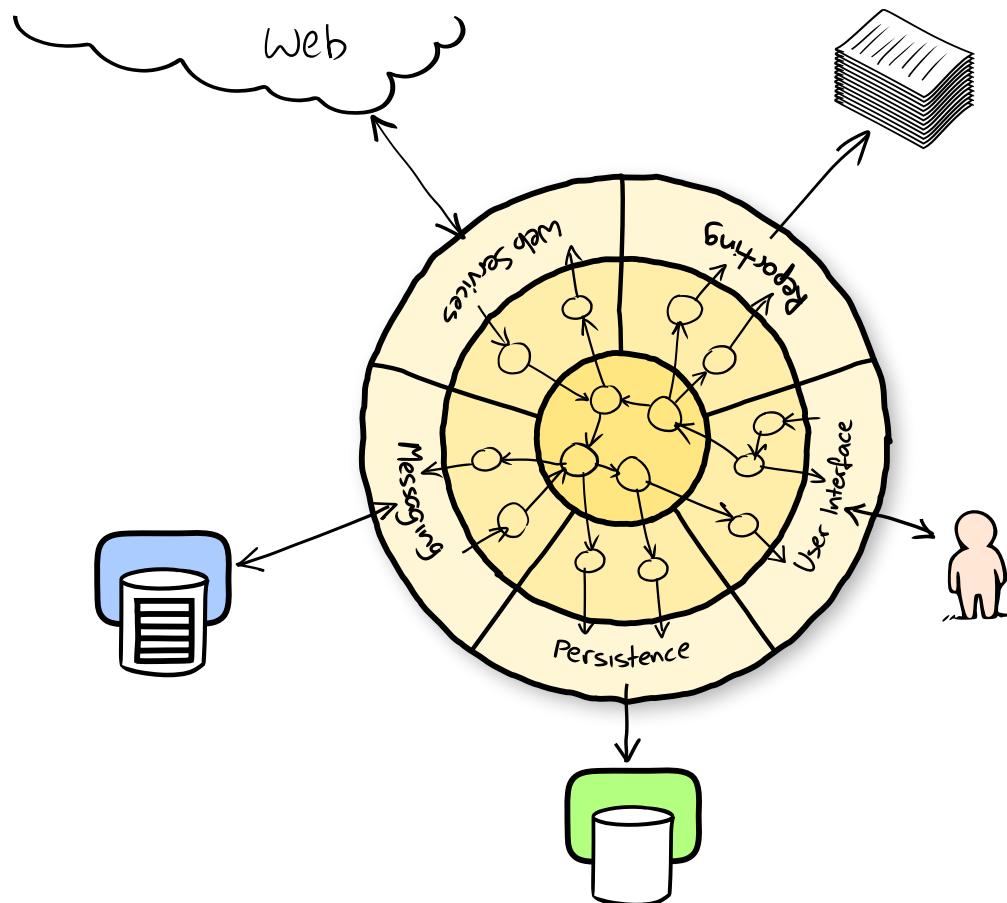
- [The Rule of Three](#)
- Refactor When You Add Functionality
- Refactor When You Need to Fix a Bug
- Refactor As You Do a Code Review

# **Architekturen**

# Schichtenarchitektur

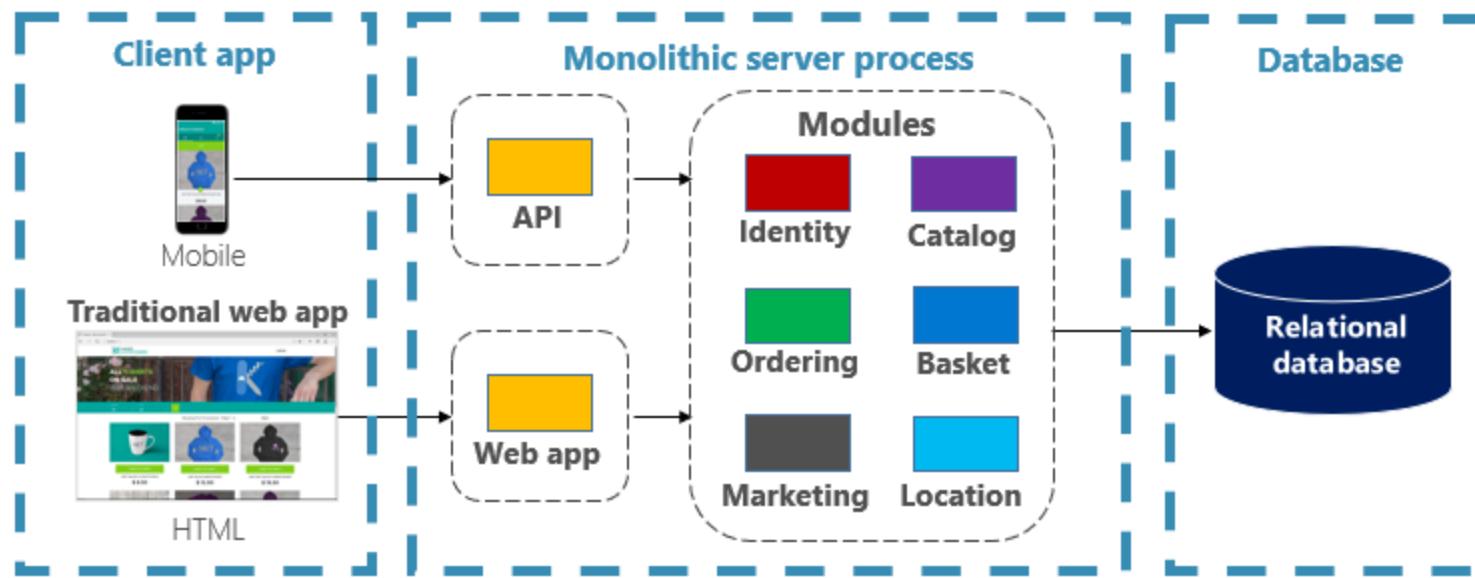


# Ports and Adapters

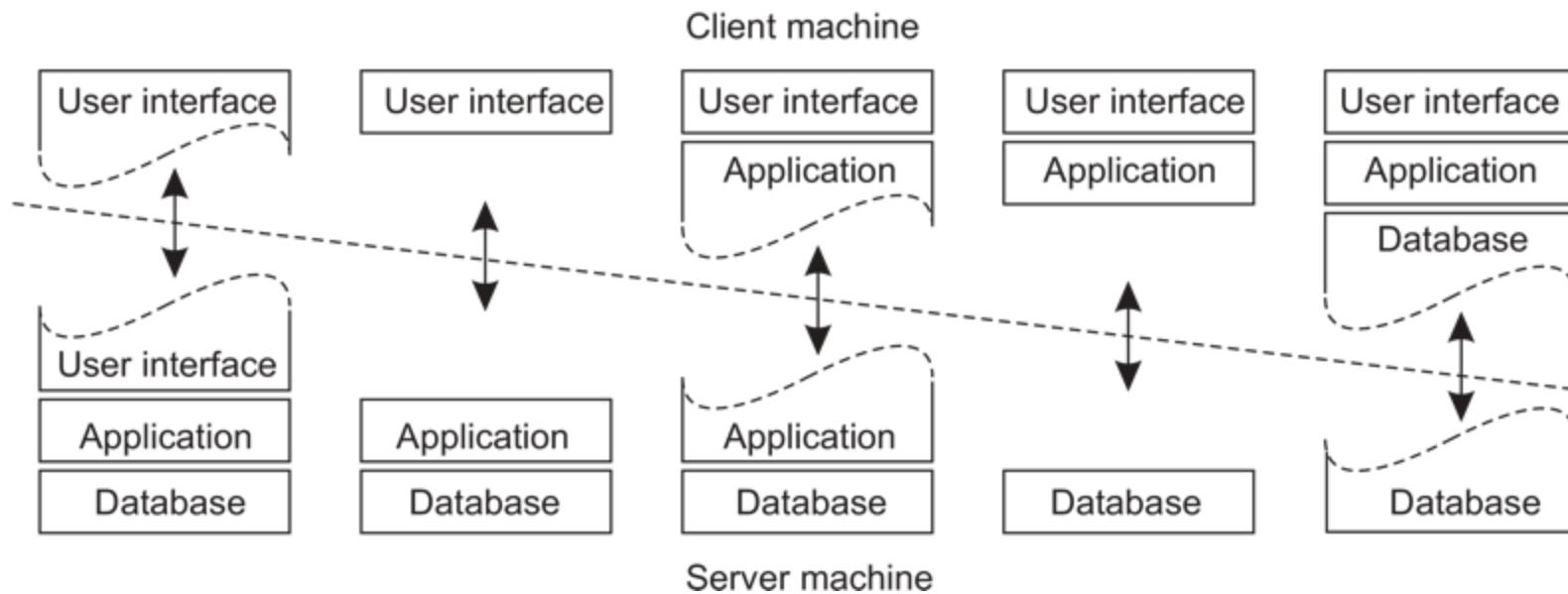


[growing-object-oriented-software.com](http://growing-object-oriented-software.com)

# Traditional Monolithic Design



# Schichtenarchitektur im Client Server Modell



# Microservices



[martinfowler.com/articles/microservices.html](http://martinfowler.com/articles/microservices.html)

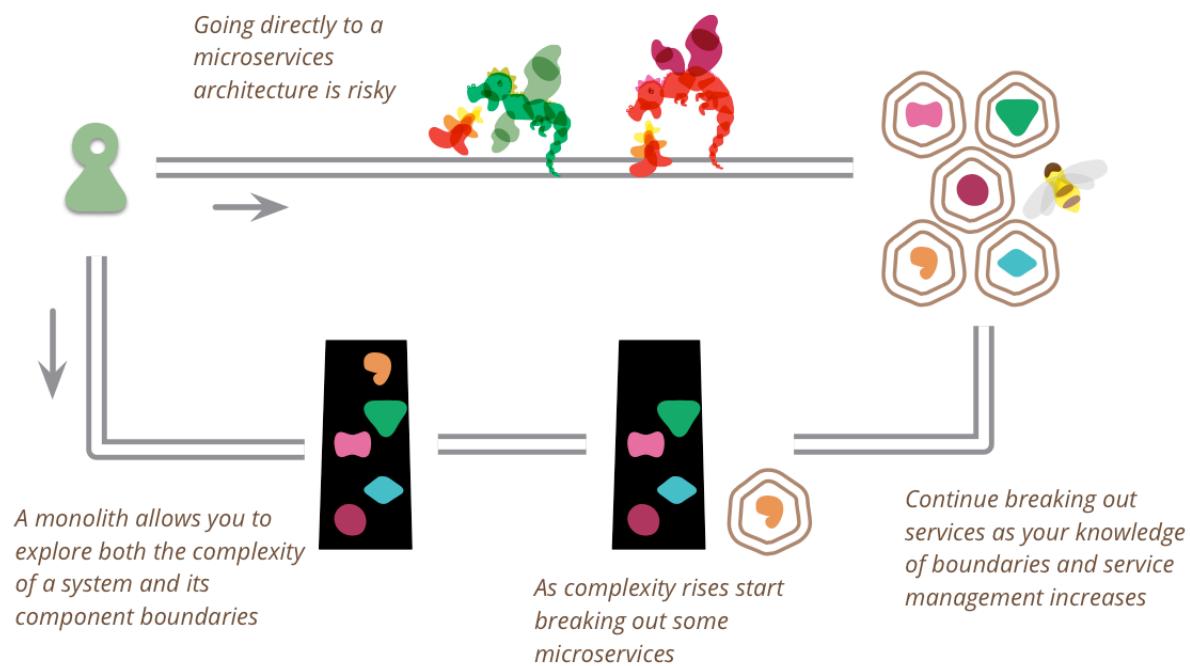
# Microservices

- 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



# Quellen

Farley, 2022

: David Farley (2022): Modern Software Engineering: Doing What Works to Build Better Software Faster, Addison-Wesley

Martin, 2018

: Robert C. Martin (2018): Clean Architecture: A Craftman's Guide to Software Structure and Design, Prentice Hall

Richards, 2021

: Mark Richards, Neal Ford (2021): Handbuch moderner Softwarearchitektur: Architekturstile, Patterns und Best Practices, O'Reilly, 978-3-96009-149-3